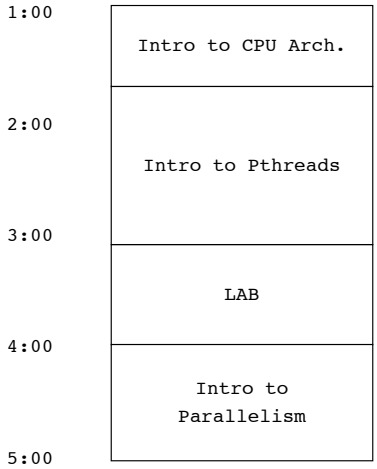
# Intro to Pthreads Programming

http://wiki. duke.edu/display/SCSC
scsc@duke.edu

John Pormann, Ph.D.
jbp1@duke.edu

## Schedule

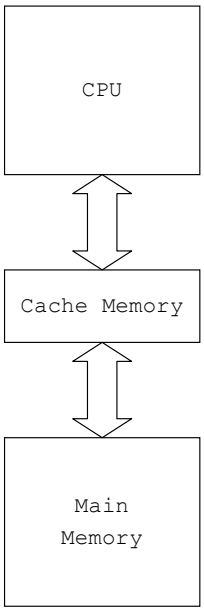
Wednesday



## Outline

- CPU Architecture
  - "Shared Memory" vs. "Distributed Memory"
- What are Pthreads?
  - Operational Model for Pthreads
- Pthread Library
  - Pthreads in 2 Function Calls
  - Thread "communication"
  - Thread synchronization
- Performance Issues/Hints

## Basic CPU Architecture



## Outline

- CPU Architecture
  - "Shared Memory" vs. "Distributed Memory"

⇨ What are Pthreads?
  - Operational Model for Pthreads

- Pthread Library
  - Pthreads in 2 Function Calls
  - Thread "communication"
  - Thread synchronization

- Performance Issues/Hints

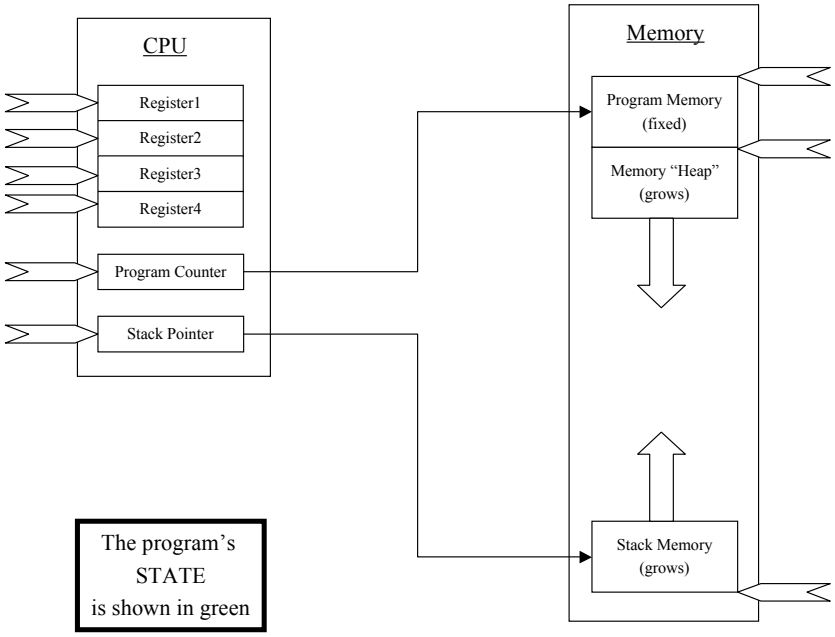
## What are Pthreads?

- The Pthread (POSIX Thread) Library is set of functions that enable C/C++ code to spawn multiple "threads" of execution to do multiple tasks simultaneously

  - It is fairly low-level
    - You create the thread, you tell it what to do, you wait for it to finish
  - "Communication" is through shared memory
    - So a Pthreads program cannot span multiple machines (like MPI)

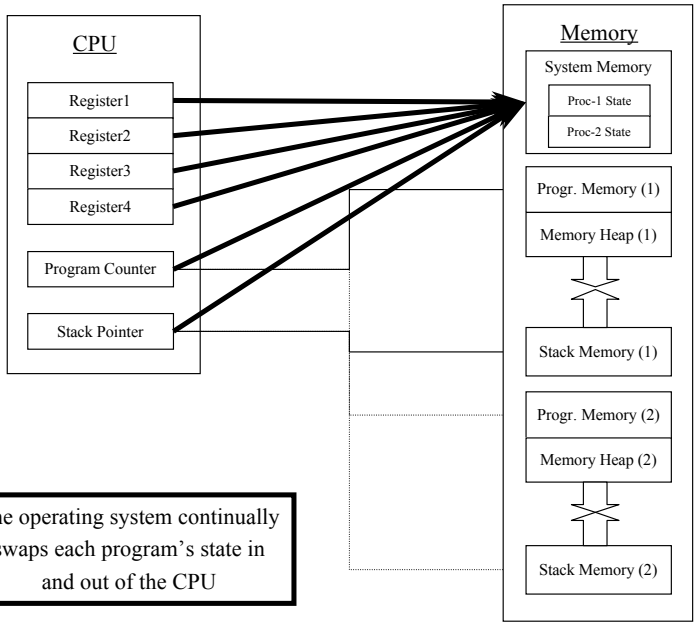
  - No explicit Fortran binding is specified, but you can write a "driver" program in C which makes calls to your Fortran routines

# The CPU's View of a Program




The program's STATE is shown in green

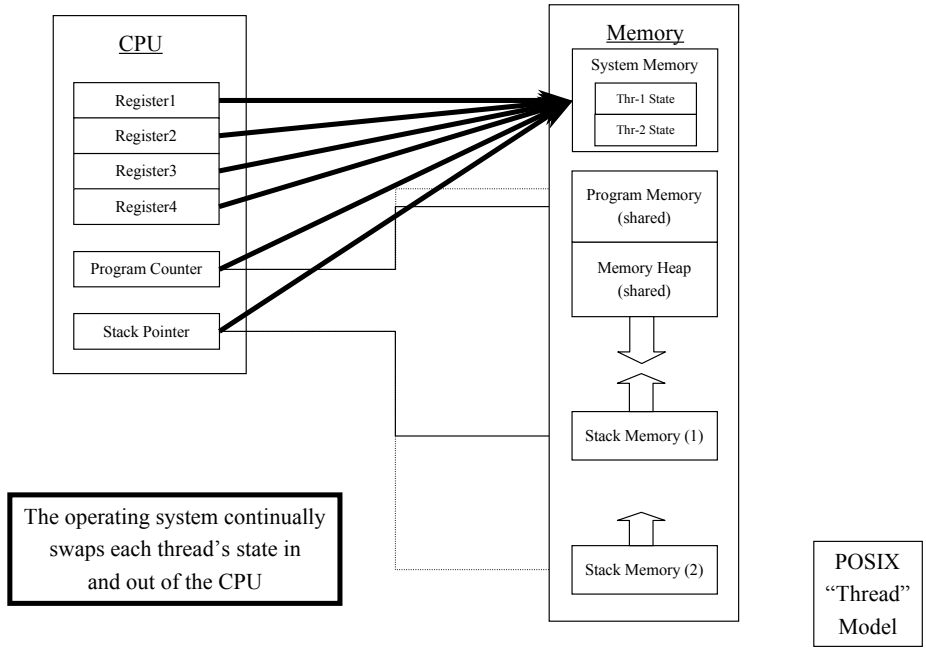
# Multiple Processes on a Single CPU




The operating system continually swaps each program's state in and out of the CPU

Unix "Process" Model

## Multiple Threads on a Single CPU




## Multiple {Processes|Threads} on Multiple CPUs

- The operating system handles swapping the {Process|Thread} state in and out of all CPUs according to some scheduling policy
  - Generally, the OS tries to be "smart" and "fair" about this
    - It may implement some kind of priority to allow system threads to jump ahead in the order to do some critical task
    - If CPU#2 is sitting idle, it will run a Prog-2 there instead of kicking Prog-1 out of CPU#1
    - On NUMA SMP systems, the OS may try to run a {Process|Thread} on a CPU that is close to the job's memory
  - The OS is not always as smart as you want it to be!

- Remember:
  - Multiple processes do NOT share memory
  - Multiple threads DO share memory

## Multiple Threads are Asynchronous

- Even if you create two identical threads (or processes) and have them run EXACTLY the same program on EXACTLY the same data, there is no guarantee that they will stay synchronized
  - OS may have other work to do, so Thread-2 may be stalled while Thread-1 is able to do computations
    - Much of this "OS work" is temporally random
  - Certain operations are inherently single threaded, so one thread will be done first
    - Disk I/O, Network I/O … there may be only one disk or one NIC
    - Older libraries may not be thread-aware and may only work if one thread uses the lib at a time (e.g. the C string lib)
  - This is somewhat counterintuitive given that we know computers are finite-state machines following a fixed program

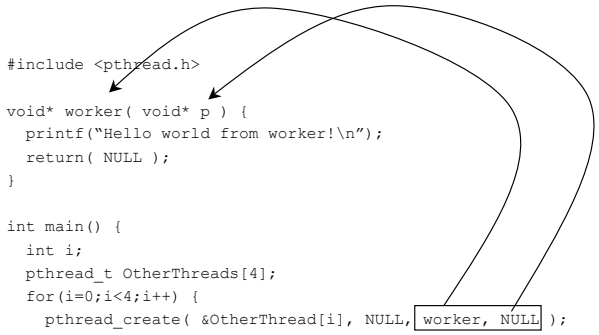
## Outline

- CPU Architecture
  - "Shared Memory" vs. "Distributed Memory"
- What are Pthreads?
  - Operational Model for Pthreads
- ⇨ Pthread Library
  - Pthreads in 2 Function Calls
  - Thread "communication"
  - Thread synchronization
- Performance Issues/Hints

## Pthreads in 2 Function Calls

- pthread_create()
  - Create a new thread of execution which executes an assigned function call
    - You can create different threads with different “work” functions
    - Generally, we create a pool of identical “workers”
  - This is not a hierarchy!!
    - There is a master thread, but everything else is equivalent
    - but threads may not be “visible” to other threads
  - If one thread exits, other threads keep going
- pthread_join()
  - The calling thread waits for the specified thread to exit

## “Hello World” Example



```
#include <pthread.h>

void* worker( void* p ) {
  printf(“Hello world from worker!\n”);
  return( NULL );
}

int main() {
  int i;
  pthread_t OtherThreads[4];
  for(i=0;i<4;i++) {
    pthread_create( &OtherThread[i], NULL, worker, NULL );
  }
  printf(“Hello world from main!\n”);
  for(i=0;i<4;i++) {
    pthread_join( OtherThread[i], NULL );
  }
  return( 0 );
}
```

```
Hello world from worker!
Hello world from worker!
Hello world from worker!
Hello world from worker!
Hello world from main!
```

## Building Pthread Code

- Most Unix/Linux systems come with the Pthread library already installed
  - Usually located in a standard location (/usr/lib)
  - Usually called libpthread.so or libpthread.a

```
% gcc -o myprog myprog.c -lpthread
```

  - All major compilers support it
    - All versions of gcc support it
  - You may need to link against "reentrant" versions of the standard libraries
    - E.g. the C string library is not multi-thread enabled -- simultaneous calls to strtok will collide and overwrite each other

## Outline

- CPU Architecture
  - "Shared Memory" vs. "Distributed Memory"
- What are Pthreads?
  - Operational Model for Pthreads
- Pthread Library
  - Pthreads in 2 Function Calls
  - ⇨ Thread "communication"
  - Thread synchronization
- Performance Issues/Hints

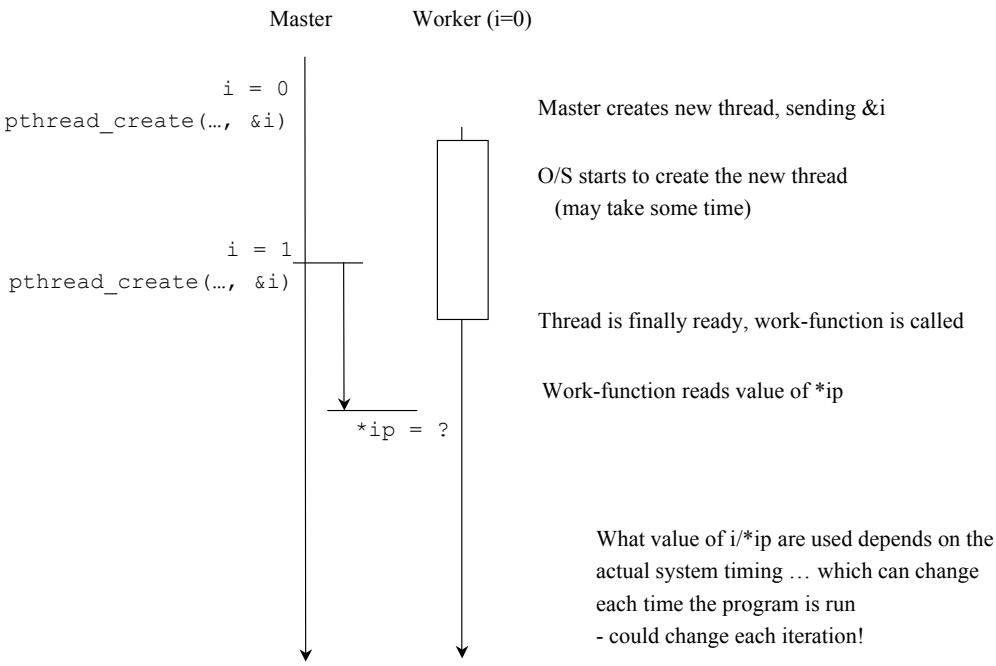
## Master-to-Worker Communication

- The Master passes a pointer as an argument to pthread_create()
  - This pointer is technically void*
  - Since you have to write both the Master and the Worker code, you can safely assume any pointer type you want
    - Even a pointer to a user-defined structure (or class)

```
#include <pthread.h>

void* worker( void* p ) {
  int* ip = (int*)p;
  printf("Hello world from worker %i!\n",*ip);
  . . .
}

int main() {
  pthread_t OtherThread[4];
  for(i=0;i<4;i++) {
    pthread_create( &OtherThread[i], NULL, worker, &i );
  }
}
```

Any potential problems?

## Master-to-Worker Communication, cont'd



# Master-to-Worker Communication, cont'd

```
#include <pthread.h>

void* worker( void* p ) {
  int* ip = (int*)p;
  printf("Hello world from worker %i!\n",*ip);
  . . .
}

int main() {
  pthread_t OtherThread[4];
  int ThreadNums[4];
  for(i=0;i<4;i++) {
    ThreadNums[i] = i;
    pthread_create( &OtherThread[i], NULL, worker, &ThreadNums[i] );
  }
}
```

ThreadNum data-area is stable even as 'i' increments

# Master-to-Worker Communication, cont'd

```
#include <pthread.h>

typedef struct {
  int self_num;
  int start,stop;
  double* data;
} workunit_t;

void* worker( void* p ) {
  workunit_t* wp = (workunit_t*)p;
  printf("Hello world from worker %i!\n",wp->self_num);
  printf("working on range: %i to %i\n",wp->start,wp->stop);
  . . .
}

int main() {
  workunit_t WorkunitList[4];
  double* maindata;
  . . .
    WorkunitList[i].self_num = i;
    WorkunitList[i].start = 1000*i;
    WorkunitList[i].stop  = 1000*(i+1) - 1;
    WorkunitList[i].data  = maindata;
    pthread_create( &OtherThread[i], NULL, worker, &WorkunitList[i] );
  . . .
}
```

(1) Define a structure with all the info needed by a single worker

(3) Each worker can unpack their own info from the arg

(2) Fill in an array of structures with each worker's real info

## Worker-to-Master Communication

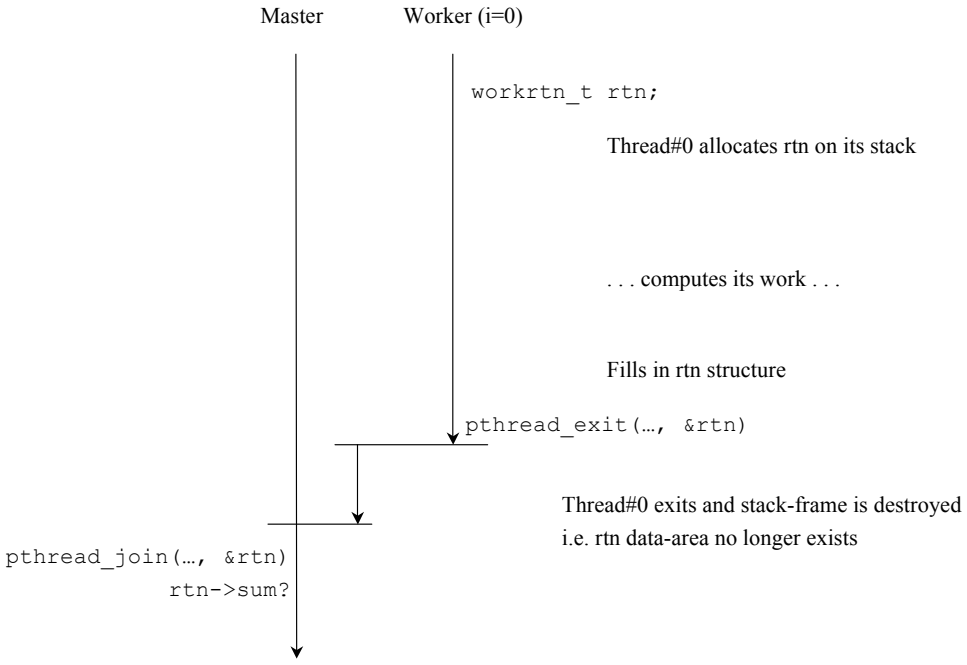
- You can use pthread_exit() or just the return() statement to return a single void* value
  - Again, you have to write both the Master and Workers so you can safely assume a pointer of some other type

```
#include <pthread.h>

typedef struct {
  double sum, sum2, min, max;
} workrtn_t;

void* worker( void* p ) {
  workrtn_t rtn;
  . . .
  rtn.sum = compute_local_sum();
  rtn.sum2 = compute_local_sum2();
  . . .
  return( &rtn );
}

int main() {
  workrtn_t* result;
  . . .
  pthread_join( OtherThread, &result );
}
```

You must return a pointer to a memory area

Yes, you send it the address of a pointer

## Worker-to-Master Communication, cont'd



## Worker-to-Master Communication, cont'd

```
#include <pthread.h>

typedef struct {
  double sum, sum2, min, max;
} workrtn_t;

void* worker( void* p ) {
  workrtn_t* rtn;
  rtn = malloc( sizeof(workrtn_t) );
  . . .
  rtn->sum = compute_local_sum();
  rtn->sum2 = compute_local_sum2();
  . . .
  return( rtn );
}

int main() {
  workrtn_t* result;
  . . .
  pthread_join( OtherThread, &result );
}
```

You must return a pointer to a HEAP memory area

## Worker-to-Master Communication, cont'd

- You could also just allocate "return-data" space somewhere in the global memory space
  - But you'll need to identify (and keep track of) what thread should write results to what block of the "return-data" memory area
- You could include a "return-data" field in the "work-unit" structure
  - This structure must live somewhere globally, or else in the Master's stack -- either way it is a stable location

```
typedef struct {
  int self_num
  int start,stop;
  double* data;
  double sum,sum2,min,max;
} workunit_t;

void* worker( void* p ) {
  workunit_t* wp = (workunit_t*)p;
  . . .
  wp->sum = compute_sum();
  . . .
  return( NULL );
}
```

Note that these values will be visible to the Master always, but will only be stable AFTER the pthread_join() call

## Why bother with these "Workunit" structures?

- Global variables are known to cause many headaches
  - Since every thread can read/write global variables, it is easy to have threads overwrite each other (and hence lose data)
  - Functions may have "side effects" (make changes to global variables) that are not apparent in their argument list
    - So changes to one function may have ripple effects, causing new bugs in other functions due to global variables being accessed differently
- Good OO or Modular Programming technique -- encapsulation

## Worker-to-Worker Communication

- Basic mechanism is to use shared memory
  - All threads can "see" the same global memory areas
  - They CANNOT see variables defined INSIDE a function

```
#include <pthread.h>

int global_var = 0;

void* worker( void* p ) {
  int local_var1 = 22;
  int* array;

  array = (int*)malloc( 100*sizeof(int) );

  return( NULL );
}

int main() {
  int local_var2 = 33;
  pthread_t OtherThread;
  pthread_create( &OtherThread, NULL, worker, NULL );
  /* thread-1 does work here */
  pthread_join( OtherThread, NULL );
  return( 0 );
}
```

Visible to all Workers

Only visible to THIS Worker

Allocated on the heap, but only known by THIS Worker

Only visible to Master

## Shared Memory Communication

```
#include <pthread.h>

typedef struct {
  int self_num
  int start,stop;
  double* data;
} workunit_t;

void* worker( void* p ) {
  workunit_t* wp = (workunit_t*)p;
  double* dataptr = wp->data;
  . . .
  data[i] = 0.5*( data[i-1] + data[i+1] );
  . . .
}

int main() {
  workunit_t WorkunitList[4];
  double* maindata;
  . . .
  maindata = (double*)malloc( 1000000*sizeof(double) );
  . . .
    WorkunitList[i].data  = maindata;
    pthread_create( &OtherThread[i], NULL, worker, &WorkunitList[i] );
  . . .
}
```

Workers can exchange data through the (heap-allocated) global memory area

It is up to the programmer to ensure that shared memory accesses "do the right thing"

Allocated on heap, potentially visible to all Workers

Pointer is passed to EVERY Worker

## Writing to Global Memory Areas

- Generally, multiple Threads reading from a global (shared) memory area is fine … they all read the same value
- Multiple Threads writing to a global (shared) memory area can be problematic … need to ensure proper synchronization to get consistent (correct) results
  - Usually best if only one Thread at a time has write-access
    - But you (the programmer) have to enforce this mutually exclusive write-access
  - Note that multiple Threads reading while one Thread writes can also be very tricky
    - Obviously some will read the "old" value, and some will read the "new" value
    - Be careful if the "new" value is supposed to trigger a different response in the workers

## Outline

- CPU Architecture
  - "Shared Memory" vs. "Distributed Memory"
- What are Pthreads?
  - Operational Model for Pthreads
- Pthread Library
  - Pthreads in 2 Function Calls
  - Thread "communication"
  - ⇨ Thread synchronization
- Performance Issues/Hints

## Synchronization -- Spin-Loops

- Spin-loops on global (or visible) shared variables seem obvious

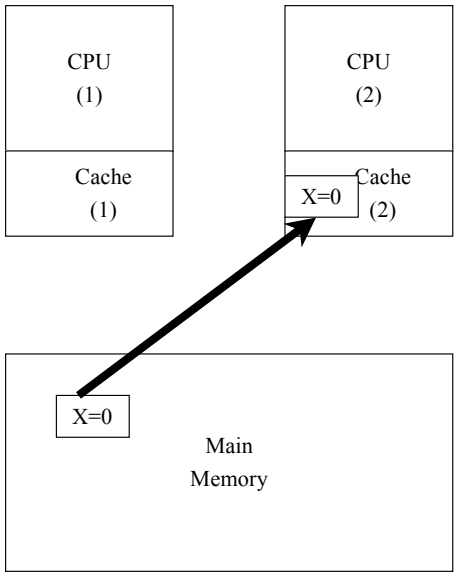
```
while( variable < 100 );
```

  - … but WILL NOT WORK !!
    - At least not all the time on all machines
    - Not even if you do:

```
while( (volatile)(variable) < 100 );
```

  - Cache effects make spin loops non-portable
    - If the machine has strict cache coherency, then spin loops will work
    - If the machine uses "relaxed" cache coherency (for better multi-processor performance), then the variable may not be written out to main memory … so other CPUs cannot see the correct value
    - A single dual-core chip may have different cache coherency behavior than two single-core chips
      - And two dual-core chips may have different cache behavior if the threads are in the same chip or on two different chips

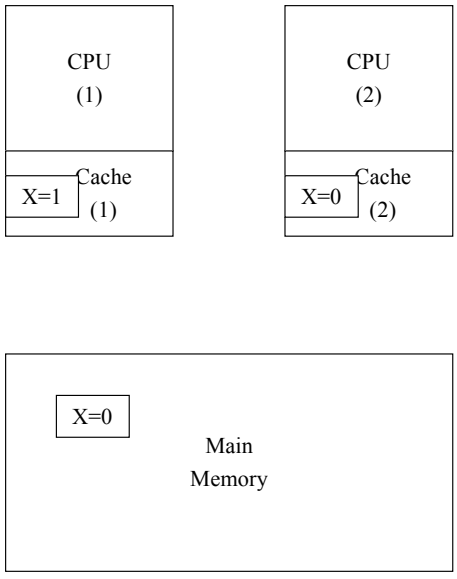
## Spin-Loop, details




```
read x
while( x == 0 );
```

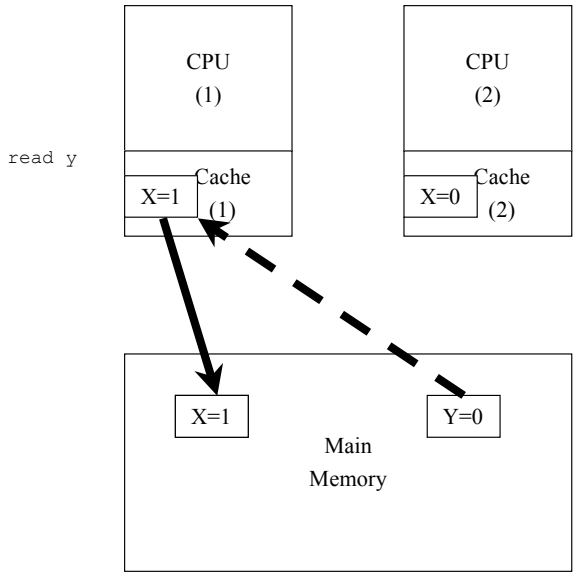
## Spin-Loop, details

```
/* hit found */
x = 1;
```



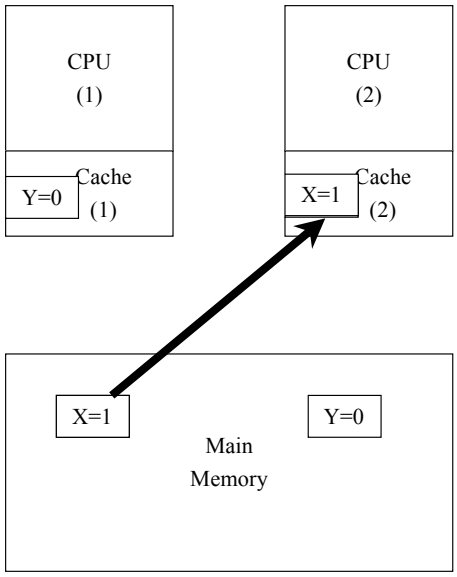

Caches and Memory are not coherent! (this can improve performance)

## Spin-Loop, details




At some later time, CPU-1 reads Y which conflicts with X's cache-line

## Spin-Loop, details




```
read x
while( x == 0 );
/* exit from loop */
```

Once the new value shows up in Main Memory, CPU-2 can finally see it

## Synchronization - Mutual-Exclusion

- A ‘mutex’ (mutual exclusion) is a region of code that only one thread at a time can enter
  - A thread ‘locks’ the mutex when it enters, and ‘unlocks’ the mutex when it leaves
  - All other threads attempting to gain the lock must wait until the ‘owner’ unlocks the mutex -- at which point one of the waiting threads is (essentially) chosen randomly
    - pthread_mutex_t
    - pthread_mutex_init()
    - pthread_mutex_lock()
    - pthread_mutex_unlock()
    - pthread_mutex_trylock()
- This is a single-thread region … obviously this will be bad for parallel performance

Note that we talk of mutex ‘regions’ but it is entirely programmer dependent. The locking/unlocking is done on the mutex variable.

## Mutex Example

```
#include <pthread.h>

int global_sum = 0;
pthread_mutex_t global_sum_mtx;

void* worker( void* p ) {
  /* compute local sum */
  . . .
  pthread_mutex_lock( &global_sum_mtx );
  global_sum += local_sum;
  pthread_mutex_unlock( &global_sum_mtx );
  . . .
}

int main() {
  . . .
  pthread_mutex_init( &global_sum_mtx, NULL );
  . . .
  pthread_create( &OtherThread[i], NULL, worker, &WorkunitList[i] );

}
```

What if we hadn’t locked the increment operation?

## Mutex Example, cont'd

- The increment operation is really several discrete steps:
  - Read 'global_sum' from shared memory into a CPU-register
  - Read 'local_sum' into a CPU-register
  - Add the two registers together
  - Store the sum back to the 'global_sum' shared memory location
  - There is a possible race condition: another Thread could have incremented 'global_sum' *just after* this Thread read the value
    - So our new 'global_sum' will be incorrect … but will overwrite any previous (correct) value

## Barriers

- A Barrier is a point in the code where all threads must arrive at that point before any one can leave that point
  - Sometimes split as two separate operations: enter-barrier and leave-barrier
  - The POSIX standard recently added barrier functionality
    - pthread_barrier_init()
    - pthread_barrier_wait()
    - pthread_barrier_destroy()
- A Barrier implies some number of Threads are WAITING for other Threads to arrive … obviously, this will not be good for performance
  - But often makes it easier to program … eliminates many race conditions since all threads must do "X" before any one of them can do "Y"

## POSIX Barriers

```
pthread_barrier_t barr1;

void* thread_work_routine( void* p ) {
   . . .
   DoWorkX();
   pthread_barrier_wait( &barr1 );
   DoWorkY();
   . . .
}

int main() {
   . . .
   pthread_barrier_init( &barr1, NULL, 4 );
   . . .
   pthread_create( . . . );
   . . .
}
```

Number of Threads expected for this Barrier

Minor change to compilation:

```
gcc -D_POSIX_C_SOURCE=200112 . . .
gcc -D_XOPEN_SOURCE=600 . . .
```

## Follow-Up to Cache Issues

- If you think about this for a minute, how can a Mutex or Barrier work if the caches and Main Memory can be incoherent?
  - What if the Mutex "lock" is held in someone's cache?
  - What if the Barrier counter is held in someone's cache?
- The PThreads Library calls will ensure that caches are properly flushed before and after lock/count data is updated
  - Recall that cache memory was intended to speed up calculations
  - Constantly flushing the entire cache will be bad for performance
- Unfortunately, there is no user-callable "cache-flush" routine

## Outline

- CPU Architecture
  - "Shared Memory" vs. "Distributed Memory"
- What are Pthreads?
  - Operational Model for Pthreads
- Pthread Library
  - Pthreads in 2 Function Calls
  - Thread "communication"
  - Thread synchronization

⇨ Performance Issues/Hints

## Basic Tips

- Use structures (or classes) as input/output arguments to your Thread-Work functions
  - Makes it easier to see what is/is not being shared
  - Keeps things a bit more modular
- Domain Decomposition (Iterative Parallelism) seems to be one of the easiest ways to "split" scientific computing problems up into multiple Threads
  - Look for large 'for' loops in your program
  - Put these loops into a work-function with 'start' and 'stop' bounds for the loop into a work-unit structure
    - Hint: don't forget about the "left-over" bits if the splitting isn't even
  - Use pthread_create() to start up multiple Threads

## One "Big" Parallel Region vs. Several Smaller Ones

- Starting up and shutting down Threads takes some time
  - So many, small work-functions can have significant overheads
  - Some loops may be too small to make Threading worthwhile
- Try to build one big work-function that includes all possible parallel regions you might need … i.e. include all your big loops into one work-function
  - If you need single-Thread regions within that parallel region, use a mutex with a simple flag/counter
  - In many cases, it is easier (and cheaper) to just allow Threads to do redundant computations

## One "Big" Parallel Region, cont'd

```
void* loop1( wkunit1* w1 ) {
   for(i=w1->start;i<w1->stop;i++) {
      data[i] = . . .
   }
}

void* loop2( wkunit2* w2 ) {
   for(i=w2->start;i<w2->stop;i++) {
      data[i] = . . .
   }
}

int main() {
   . . .
   pthread_create(...,loop1,...);
   . . .
   pthread_create(...,loop2,...);
   . . .
}
```

```
void* loop1n2( wkunit* wk ) {
   for(i=wk->start1;i<wk->stop1;i++) {
      data[i] = . . .
   }
   . . .
   for(i=wk->start2;i<wk->stop2;i++) {
      data[i] = . . .
   }
}

int main() {
   . . .
   pthread_create(...,loop1n2,...);
   . . .
}
```

## Timing PThreads Code

```
clock_t t0,t1;
t0 = clock();
/* do work */
t1 = clock();
dt = (double)(t1-t0)/(double)(CLOCKS_PER_SEC);
```

```
#include <time.h>

time_t t0,t1;
t0 = time(NULL);
/* do work */
t1 = time(NULL);
dt = (double)(t1-t0);
```

```
#include <sys/time.h>
struct timeval t0,t1;
gettimeofday(&t0,NULL);
/* do work */
gettimeofday(&t1,NULL);
dt = (double)(t1.tv_sec-t0.tv_sec)+(double)(t1.tv_usec-t0.tv_usec)/1.0e6;
```

POSIX-4

```
struct timespec t0,t1;
clock_gettime(CLOCK_REALTIME,&t0);
/* do work */
clock_gettime(CLOCK_REALTIME,&t1);
dt = (double)(t1.tv_sec-t0.tv_sec)+(double)(t1.tv_nsec-t0.tv_nsec)/1.0e9;
```

CLOCK_HIGHRES?

`times()` function is considered outdated

## Mutexes and Barriers

- A mutex is, by definition, a single-Thread region
  - Parallel performance will degrade if you have too many of them
  - But they may be required for correctness of your algorithm
- A barrier implies a significant amount of waiting
  - E.g. at some point 9 out of 10 Threads are sitting idle waiting for the last Thread to arrive
  - Parallel performance will degrade if you have too many of them
  - They can be helpful with debugging, especially race conditions
    - But they are a crutch!
    - You rarely NEED a barrier
      - If Thread#1 and Thread#2 need to be synchronized, then just synchronize those two, don't use a barrier just because it is easy

## Possible Alternatives to Mutexes

- In some cases, we can reduce the need for mutexes by doing local computations in each Thread, and THEN grabbing a mutex at the end to perform the global computation
  - E.g. compute the (global) sum of a large list of numbers
    - Each Thread computes its local sum
    - Grab mutex once to accumulate the local sums into the global sum
- Use a global (shared) array to store partial (per-Thread) information, then after some synchronization, all Threads can directly compute the sum across the global array
  - Perhaps the sync happens naturally, or is required for other reasons

```
int partial_sum[MAX_NUM_THREADS];

void* work( wkunit* wk ) {
   . . .
   partial_sum[self] = local_sum;
   . . .
}
```

## Race Conditions

- A "Race" is a situation where multiple Threads are making progress toward some shared or common goal where timing is critical
- Generally, this means you've ASSUMED something is synchronized, but you haven't FORCED it to be synchronized
- E.g. all Threads are trying to find a "hit" in a database
  - When a hit is detected, the Thread increments a global counter
  - Normally, hits are rare events and so incrementing the counter is "not a big deal"
  - But if two hits occur simultaneously, then we have a race condition
    - While one Thread is incrementing the counter, the other Thread may be reading it
    - Hence it may read the old value or new value depending on the exact timing of the two Threads

## Race Conditions, cont'd

- Some symptoms:
  - Running the code with 4 Threads works fine, running with 5 causes erroneous output
  - Running the code with 4 Threads works fine, running it with 4 Threads again produces different results
  - Sometimes the code works fine, sometimes it crashes after 4 hours, sometimes it crashes after 10 hours, …
  - Program sometimes runs fine, sometimes hits an infinite loop
  - Program output is sometimes jumbled (the usual line-3 appears before line-1)
  - Any kind of indeterminacy in the running of the code, or seemingly random changes to its output, could indicate a race condition

## Race Conditions, cont'd

- Race Conditions can be EXTREMELY hard to debug
  - Generally, race conditions don't cause crashes or even logic-errors
    - E.g. a race condition leads two Threads to both think they are in charge of data-item-X … nothing crashes, they just keep writing and overwriting X (maybe even doing duplicate computations)
  - Often, race conditions don't cause crashes at the time they actually occur … the crash occurs much later in the execution and for a totally unrelated reason
    - E.g. a race condition leads one Thread to think that the solver converged but the other Thread thinks we need another iteration … crash occurs because one Thread tried to compute the global residual
  - Sometimes race conditions can lead to deadlock
    - Again, this often shows up as seemingly random deadlock when the same code is run on the same input

## Deadlock

- A "Deadlock" condition occurs when all Threads are stopped at synchronization points and none of them are able to make progress
  - Sometimes this can be something simple:
    - Thread-1 locks Mutex-1 and waits for Variable-X to be incremented, and then it will release the lock
    - Thread-2 wants to increment X but needs to lock Mutex-1 in order to do that
  - Sometimes it can be more complex or cyclic:
    - Thread-1 locks Mutex-1 and needs Mutex-2
    - Thread-2 locks Mutex-2 and needs Mutex-3
    - Thread-3 locks Mutex-3 and …
- A related condition is "Starvation": when one Thread needs a resource (Mutex) that is never released by the current lock-holder

## Deadlock, cont'd

- Symptoms:
  - Program hangs (!)
- Debugging deadlock is often straightforward … unless there is also a race condition involved
  - Simple approach: put print statement in front of all Mutex calls

```
Thread-1 acquired lock on mutex-A
Thread-1 acquired lock on mutex-C
Thread-2 acquired lock on mutex-B
Thread-1 release lock on mutex-C
Thread-3 acquired lock on mutex-C
Thread-1 waiting for lock on mutex-B
Thread-2 waiting for lock on mutex-A
```

## What I left out

- Condition Variables -- a mechanism for one Thread to "signal" another
  - Some would argue this is a glaring omission
  - . . . it is fairly confusing for most new parallel programmers
- Threads and Unix Signals
  - Individual Threads can "mask" out certain signals so that other Threads will be used to service those signals
    - By default, a signal will be sent to a random Thread
    - If certain Threads are doing "important" work that shouldn't be interrupted, then they should mask out all/most signals
- Threads and Unix Semaphores
  - Semaphores are shared counters
  - Threads can increment and decrement semaphores, or wait for the semaphore to hit a certain value

## Where to go from here?

- http://www.csem.duke.edu . . . csem@duke.edu
- Get on our mailing lists:
  - Scientific Visualization seminars (Friday 12-1pm)
  - CSEM seminars
  - Workshop announcements
- Follow-up seminars on specific topics
  - Parallel programming methods (OpenMP, Pthreads, MPI)
  - Performance tuning methods (Vtune, Parallel tracing)
  - Visualization tools (Amira, AVS, OpenDX)
- Contacts:
  - Dr. Bill Rankin, wrankin@ee.duke.edu
  - Dr. Rachael Brady, rbrady@duke.edu
  - Dr. John Pormann, jbp1@duke.edu

## Mutex/Condition Variables

- A "Condition Variable" is a point in the program where one thread can "signal" a condition to another thread
  - It is ALWAYS associated with a Mutex (but perhaps a "smaller" mutex including just the two threads that need to communicate)
    - pthread_cond_init()
    - pthread_cond_signal(), pthread_cond_broadcast()
    - pthread_cond_wait(), pthread_cond_timedwait()
  - E.g. when a global counter hits some threshold, we want another Worker (or Master) to print a warning
    - Build a mutex/cond-var for the signal
    - The "Recv"-Thread waits for the "signal"
    - If a worker finds the counter crossed the threshold, they "signal" the "Recv"-Thread through the cond-var

## Condition Variable Example

```
int count = 0;
pthread_mutex_t count_mtx;
pthread_cond_t count_cv;

void* worker_incr( void *vp ) {
   . . .
   pthread_mutex_lock( &count_mtx );
   count++;
   if( count > threshold ) {
      pthread_cond_signal( &count_cv );
   }
   pthread_mutex_unlock( &count_mtx );
   . . .
}

void* worker_watch( void *ip ) {
   . . .
   pthread_mutex_lock( &count_mtx );
   pthread_cond_wait( &count_cv, &count_mtx );
   . . .
   pthread_mutex_unlock( &count_mtx );
   . . .
}
```

While the thread owns the lock, it can send a signal

While the thread owns the lock, it can wait for a signal

Waiting for a signal simultaneously unlocks the mutex.
Sending a signal releases the receiver, but it immediately has to compete to regain the mutex

## Barrier, cont'd

```
pthread_mutex_t barrier_mtx;
pthread_cond_t barrier_cv;
int num_workers;  /* set by master thread */
int num_arrived = 0;

void Barrier( void ) {
   pthread_mutex_lock( &barrier_mtx );
   num_arrived++;
   if( num_arrived < num_workers ) {
      pthread_cond_wait( &barrier_cv, &barrier_mtx );
   } else {
      num_arrived = 0;
      pthread_cond_broadcast( &cv );
   }
   pthread_mutex_unlock( &barrier_mtx );
}

void* thread_work_routine( void* p ) {
   . . .
   DoWorkX();
   Barrier();
   DoWorkY();
   . . .
}
```

## Unix Signals

- Signals are asynchronous events that trigger a function call inside the program
  - E.g. the operating system will tell your program when the user logs out via a SIGHUP (hang-up) signal
  - E.g. batch queuing systems will tell your program that it is running out of time via a SIGXCPU signal
- signal( sig_name, function )
  - when given signal arrives, the specified function will automatically be called
    - Your program will immediately jump to the function and execute it
    - Then your program will go back to whatever it was doing before

## Unix Signals, cont'd

- kill
  - Send a signal to a PROCESS
    - A thread will be selected at random to handle the interrupt
- pthread_kill
  - Send a signal to a specific THREAD
- pthread_sigmask
  - "Mask" out or block certain signals from affecting the current thread
    - Used to force only 1 thread to accept an incoming signal from 'kill'
- sigsuspend
  - Wait for a signal to arrive before continuing