Logistics

- **OpenMP homework should be started.**
  - Remember you have to use a batch system at NERSC!!
- **Access to isugw/osage/redwing**
  - Everyone able to get there??
- **Midterm Exam is less than 2 weeks from today**
  - 12 days
- **Homework issues**
  - PATH does not contain “.”
  - Timing functions (.cpp vs .c)
    - GCC is easy 😊
- **Any Questions??????????**
Destroying Threads

• There are 3 ways to terminate threads
  ➢ Return from the thread “start_fn”
  ➢ pthread_exit();
  ➢ Get cancelled by another thread
  ➢ (void)exit() is not one of the three ways!!!

• void pthread_exit(void **status);
  ➢ *status is a pointer to the completion status of the destroyed thread. Available to other threads.
  ➢ No return value from the function pthread_exit

• When a thread is destroyed by returning from the start routine, the thread’s completion status is set to the return value.
Joining Threads

- `int pthread_join(pthread_t tid, void **status)`
  - `tid` is the thread id (from the create); The thread to wait for.
  - `*status` is the exit status of the thread.
    - Thread completion status is copied to `*status`
    - If `*status` is NULL then it is not copied.
  - Return value 0 for success
- Once a thread is joined the thread ID is no longer valid.
  - It cannot be joined with any other thread 😊
Joining and Exiting Threads

• Thread code should not call exit( )
  ➢ Operates on the Process!!
• Main should not return before child threads are finished.
  ➢ Same as exit( ) in main. Kills the process
  ➢ The parent should always exit AFTER all other threads are joined (or completed).
int pthread_create

#include <pthread.h>

int pthread_create(
    pthread_t *new_thread_ID,
    const pthread_attr_t *attr,
    void * (*start_func)(void *),
    void *arg);
Thread Attributes: detachstate

- Controls whether the thread is created in the joinable state or in the detached state
- Default value: PTHREAD_CREATE_JOINABLE
- In the joinable state,
  - another thread can synchronize on the thread termination and recover its termination code using pthread_join
  - some of the thread resources are kept allocated after the thread terminates, and reclaimed only when another thread performs pthread_join on that thread.
- In the detached state,
  - the thread resources are immediately freed when it terminates
  - pthread_join cannot be used to synchronize on the thread termination.
- A thread created in the joinable state can later be put in the detached
Thread Attributes: schedpolicy

- The scheduling policy for threads can be one of:
  - **SCHED_OTHER**
    - regular,
    - non-realtime
  - **SCHED_RR**
    - realtime, round-robin
    - usually root only
  - **SCHED_FIFO**
    - realtime, first-in first-out
    - usually root only
- Default: **SCHED_OTHER**
Thread Attributes: schedparam

- Contain the scheduling parameters essentially, the scheduling priority for the thread.
- Default value: priority is 0.
  - Values and range of values system specific
- Most important for the realtime policies SCHED_RR and SCHED_FIFO
Thread Attributes: inheritsched

• Indicates whether the scheduling policy and scheduling parameters for the newly created thread are
  ➢ determined by the values of the specified attributes or
  ➢ are inherited from the parent thread.

• Default value:
  PTHREAD_EXPLICIT_SCHED
  ➢ E.g. by specified attributes
Thread Attributes: scope

• Defines the scheduling contention scope for the created thread.

• PTHREAD_SCOPE_SYSTEM
  - means that the threads contend for CPU time with all processes running on the machine.
  - Bound Threads

• PTHREAD_SCOPE_PROCESS
  - means that scheduling contention occurs only between the threads of the running process
  - thread priorities are interpreted relative to the priorities of the other threads of the process, regardless of the priorities of other processes
  - Unbound Threads

• Default values:
  - Linux PTHREAD_SCOPE_SYSTEM.
  - Solaris PTHREAD_SCOPE_PROCESS.
One Problem

• Setting the scheduling scope to PTHREADS_SCOPE_SYSTEM
  ➢ Breaks CPU_Time
  ➢ Even for 1 thread!!
    ▪ No resolution at this time.
  ➢ Use wall time!
Pthreads Pi

• Handout.
• Compute Pi with
  ➢ a fixed number of iterations
  ➢ A variable number of threads.
• Timing results
  ➢ 1 thread 177 sec
  ➢ 2 threads 90 sec
  ➢ 3 threads 90 sec
  ➢ 4 threads 90 sec
• How many processors do I have on the box?
• Efficiency is 98% on 2 threads.
argument structure and global data

- typedef struct
- {
  int my_id;
  int nproc;
  int my_start;
  int my_end;
} my_arg;

- double *pi_parts;
Thread function

void *pi_term(void *arg)
{
    my_arg *pp;  int i;  double part1, part2, pi_thread;
    pp = (my_arg *)arg;
    part1 = (double)0.0;
    for(i=pp->my_start;i<=pp->my_end;i+=2)
        part1 += (one/(two*(((double)i)-one)));
    part2 = (double)0.0;
    for(i=(pp->my_start+1);i<=pp->my_end;i+=2)
        part2 += (one/(two*(((double)i)-one)));
    if ((pp->my_start)%2)    pi_thread = part1 - part2;
    else                                   pi_thread = part2 - part1;
    pi_parts[pp->my_id] = pi_thread*four;}

2/18/2005               ComS 425               Spring 2005

Allocating memory

thread_ids =
    (pthread_t *)
malloc(num_thread*sizeof(pthread_t));

pt_arg = (my_arg *)
malloc(num_thread*sizeof(my_arg));

pi_parts = (double *)
malloc(num_thread*sizeof(double));
Starting the Threads

```c
pthread_attr_init(&thread_init_attr);
pthread_attr_setscope(&thread_init_attr,
PTREAD_SCOPE_SYSTEM);
for (i=0; i<num_thread; i++) {
    pt_arg[i].my_id=i;
    pt_arg[i].nproc = num_thread;
    pt_arg[i].my_start = i*block_size + 1;
    pt_arg[i].my_end   =
MIN(((i*block_size)+block_size),num_iters);
pthread_create(&thread_ids[i],
&thread_init_attr, pi_term, (void *)(pt_arg+i));
}
```
Pthreads

• What we have learned so far is powerful enough to do many kinds of algorithms

• The key to getting performance with threaded applications.
  ➢ Synchronization
  ➢ The lack of interference among threads

• Complexity comes from
  ➢ Dealing with synchronization and interference.
Example Matrix Multiply

/* Routine to multiply a row by a column and place element in resulting matrix. */
void mult(int size, int row, int column,
          matrix_t MA, matrix_t MB, matrix_t MC)
{
    int position;
    MC[row][column] = 0;
    for(position = 0; position < size; position++) {
        MC[row][column] +=
            ( MA[row][position] * MB[position][column] ) ;
    }
}
Example Matrix Multiply [2]

typedef struct { int id; int size; int Arow;
int Bcol; matrix_t *MA, *MB, *MC;} package_t;

void *mult_worker(void *arg)
{
    package_t *p=(package_t *)arg;
    mult(p->size, p->Arow, p->Bcol, *(p->MA), *(p->MB), *(p->MC));
    free(p);
    return(NULL);
}
Example Matrix Multiply [3]

```c
int main(int argc, char **argv) {
    int size, row, column, num_threads, i;
    pthread_t *threads; package_t *p;
    pthread_attr_t *pthread_attr_p, size = ARRAY_SIZE;
    /* one thread will be created for each element of the matrix. */
    threads = (pthread_t *)malloc(size*size*sizeof(pthread_t));
    /* Fill in matrix values, for A and B */
    pthread_attr_init(&pthread_custom_attr);
    pthread_attr_setscope(&pthread_custom_attr,
                         PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setscope...
```
Example Matrix Multiply [4]

/* Process Matrix, by row, column, Create a thread to process each element in the resulting matrix*/
for(row = 0; row < size; row++) {
    for (column = 0; column < size; column++) {
        p = (package_t *)malloc(sizeof(package_t));
        p->id = num_threads; p->size = size;
        p->Arow = row;      p->Bcol = column;
        (p->MA) = &MA;       (p->MB) = &MB;      (p->MC) = &MC;

        pthread_create(&threads[num_threads],
                       &pthread_custom_attr,    mult_worker, (void *) p);
    }
}
Example Matrix Multiply [5]

/* Synchronize on the completion of the element in each thread. */
for (i = 0; i < (size*size); i++) {pthread_join(threads[i], NULL);}

/* Print results */
printf("MATRIX MAIN THREAD: The resulting matrix C is:\n");
for(row = 0; row < size; row++) {
    for (column = 0; column < size; column++) {
        printf("%5d ",MC[row][column]);
    }
    printf("\n");
}
return 0;
}
Methods of Synchronization with threads

• **Pthread_join**
  - Essentially a barrier to execution until at least two threads are joined.
  - In examples so far this has been the “master” thread waiting on all other threads.

• **Mutex variables**
  - Mutual Exclusion

• **Condition variables**
  - Always in conjunction with mutexes
  - A way to make mutexes complex 😊

• **Semaphores**
The Next step beyond simple synchronization

- With these basic tools more complex synchronization tools can be built.
  - Read/write locks and unlocks
    - Many can read but only one can write.
  - Threadsafe data structures.
    - A linked list where the add/remove etc. functions have to acquire a lock to operate on the shared list data structure.
    - A queue system where the enqueue/dequeue functions must lock to operate on the shared queue data structure.
Race Conditions

• A race condition is when two or more threads try to modify the same memory location at the same time.

➢ Remember the critical sections?

• For example. Two threads are incrementing a global count variable.

➢ count++

   ▪ Ignoring the NUMA aspects there are at least three instructions
     ♦ Load count into register
     ♦ Increment register
     ♦ Store register to memory location for count.
   ▪ Without protection this operation is doomed.
Mutexes

• Has two states locked or unlocked
• Has two functional operations lock and unlock
• Only one thread can hold the lock
  ➢ Any other thread trying to lock it will wait until it is unlocked.
    ▪ Put into a “blocked” state
  ➢ The lock holding thread has exclusive right to the critical section of code that is controlled by the lock.
    ▪ This can be doing operations
    ▪ Accessing data (reads)
    ▪ Initializing setting data (writes).
• Can be either statically or dynamically allocated.
Statically Allocated Mutexes

- `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;`
  - This gives the lock the default attributes for locks.
  - Like threads there are attributes to locks that can be set.

- `pthread_mutex_lock(&lock);`
- `... critical section of code`
- `pthread_mutex_unlock(&lock);`
Dynamically allocated locks

```c
pthread_mutex_t *lock;
lock = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL); /*defaults*/
pthread_mutex_lock(lock);
... critical section of code
pthread_mutex_unlock(lock);
pthread_mutex_destroy(lock);
free(lock);
```
Now we can increment count properly

```c
int count;
pthread_mutex_t *lock;
lock = (pthread_mutex_t *)
    malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL); /*defaults*/
pthread_mutex_lock(lock);
    count++;
pthread_mutex_unlock(lock);
pthread_mutex_destroy(lock);
free(lock);
```
Or Can we??

• **This code will work perfectly:**
  - On a single processor workstation
  - In PTHREAD_SCOPE_PROCESS on multiple processor systems.
  - In PTHREAD_SCOPE_SYSTEM on multiple processor systems.

• **Performance will be acceptable:**
  - On a single processor workstation
  - In PTHREAD_SCOPE_SYSTEM on multiple processor systems.

• **Performance will NOT be acceptable:**
  - In PTHREAD_SCOPE_PROCESS on multiple processor systems.
  - Why?