Lecture 5:  Threads ... Pthreads (Part 2)

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides of this lecture are from:
• Peter S. Pacheco book
Issues

• **Busy-waiting**: enforces the order threads access a critical section but inefficiently uses resources.

• **Mutexes**: the order is left to chance and the system.

• There are applications where:
  – We need to control the order threads access the critical section.
  – We may need to have more than one thread in the critical section.
Problems with a mutex solution

```c
/* n and product_matrix are shared and initialized by the main thread */
/* product_matrix is initialized to be the identity matrix */
void* Thread_work(void* rank) {
    long my_rank = (long) rank;
    matrix_t my_mat = Allocate_matrix(n);
    Generate_matrix(my_mat);
    pthread_mutex_lock(&mutex);
    Multiply_matrix(product_mat, my_mat);
    pthread_mutex_unlock(&mutex);
    Free_matrix(&my_mat);
    return NULL;
}
/* Thread_work */
```
Syntax of the various semaphore functions

```
#include <semaphore.h>

int sem_init(
    sem_t* semaphore_p /* out */,
    int shared /* in */,
    unsigned initial_val /* in */);

int sem_destroy(sem_t* semaphore_p /* in/out */);
int sem_post(sem_t* semaphore_p /* in/out */);
int sem_wait(sem_t* semaphore_p /* in/out */);
```

Semaphores are not part of Pthreads; you need to add this.

Non-zero means semaphore shared among processes.
sem_post and sem_wait

- **sem_post**: increment the semaphore
- **sem_wait**: check semaphore value
  - if 0 then block
  - else: decrement and proceed
Barriers

• **What?** Synchronizing the threads to make sure that they all are at the same point in a program

• **Why?**
  – For correctness
  – For debugging
  – For timing

**Small challenge:** Many implementations of Pthreads do not provide barriers.
Steps

1. Declare a barrier variable (global):
   ```
   pthread_barrier_t barr;
   ```
2. Initialize the variable (by one thread):
   ```
   pthread_barrier_init(&barr, NULL, num_threads)
   ```
3. Use it (by all threads):
   ```
   pthread_barrier_wait(&barr);
   ```
4. Destroy it (by one thread):
   ```
   pthread_barrier_destroy(&barr)
   ```
Condition Variables

• **What?** A data object that allows a thread to suspend execution until a certain event or condition occurs.

• **How?** When the event or condition occurs another thread can signal the thread to “wake up.”

• **A condition variable is always associated with a mutex.**
**Condition Variables**

- `pthread_cond_t condv;
- `pthread_cond_init(&condv, NULL);
- `pthread_cond_destroy(&condv);
- `pthread_cond_signal(&condv);
  - unblock one of the blocked threads
- `pthread_cond_broadcast(&condv);
  - unblock all of the blocked threads
- `pthread_cond_wait(&condv, &mutexv);
  - unblock the mutex associated with mutexv
  - causes the calling thread to be blocked until unblocked by another thread
Implementing a barrier with condition variables

```c
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;

void* Thread_work(...) {
    ... 

    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    ... 
}
```
Let's see an example:
Multithreaded Linked List Implementation
struct list_node_s {
    int data;
    struct list_node_s* next;
}

Sorted Linked List
int Member(int value, struct list_node_s* head_p) {
    struct list_node_s* curr_p = head_p;

    while (curr_p != NULL && curr_p->data < value)
        curr_p = curr_p->next;

    if (curr_p == NULL || curr_p->data > value) {
        return 0;
    } else {
        return 1;
    }
} /* Member */
Inserting a new node into a list
Inserting a new node into a list

```c
int Insert(int value, struct list_node_s** head_pp) {
    struct list_node_s* curr_p = *head_pp;
    struct list_node_s* pred_p = NULL;
    struct list_node_s* temp_p;

    while (curr_p != NULL && curr_p->data < value) {
        pred_p = curr_p;
        curr_p = curr_p->next;
    }

    if (curr_p == NULL || curr_p->data > value) {
        temp_p = malloc(sizeof(struct list_node_s));
        temp_p->data = value;
        temp_p->next = curr_p;
        if (pred_p == NULL) /* New first node */
            *head_pp = temp_p;
        else
            pred_p->next = temp_p;
        return 1;
    } else { /* Value already in list */
        return 0;
    }
} /* Insert */
```
Deleting a node from a linked list
Deleting a node from a linked list

```c
int Delete(int value, struct list_node_s** head_pp) {
    struct list_node_s* curr_p = *head_pp;
    struct list_node_s* pred_p = NULL;

    while (curr_p != NULL && curr_p->data < value) {
        pred_p = curr_p;
        curr_p = curr_p->next;
    }

    if (curr_p != NULL && curr_p->data == value) {
        if (pred_p == NULL) { /* Deleting first node in list */
            *head_pp = curr_p->next;
            free(curr_p);
        } else {
            pred_p->next = curr_p->next;
            free(curr_p);
        }
    } else { /* Value isn't in list */
        return 0;
    }

    return 1;
}
```
A Multi-Threaded Linked List

• We can define `head_p` to be a global variable.

• This will simplify the function headers for `Member`, `Insert`, and `Delete`. 
Simultaneous access by two threads
Solution #1

• An obvious solution is to simply lock the list any time that a thread attempts to access it.

• A call to each of the three functions can be protected by a mutex.

```c
Pthread_mutex_lock(&list_mutex);
Member(value);
Pthread_mutex_unlock(&list_mutex);
```
Issues

• We’re serializing access to the list.

• If the vast majority of our operations are calls to Member, we’ll fail to exploit this opportunity for parallelism.

• On the other hand, if most of our operations are calls to Insert and Delete, then this may be the best solution since we’ll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.
Solution #2

- Instead of locking the entire list, we could try to lock individual nodes.
- A “finer-grained” approach.

```c
struct list_node_s {
    int data;
    struct list_node_s* next;
    pthread_mutex_t mutex;
}
```
```c
int Member(int value) {
    struct list_node_s* temp_p;

    pthread_mutex_lock(&head_p_mutex);
    temp_p = head_p;
    while (temp_p != NULL && temp_p->data < value) {
        if (temp_p->next != NULL)
            pthread_mutex_lock(&temp_p->next->mutex);
        if (temp_p == head_p)
            pthread_mutex_unlock(&head_p_mutex);
        pthread_mutex_unlock(&temp_p->mutex);
        temp_p = temp_p->next;
    }

    pthread_mutex_unlock(&head_p_mutex);
    return temp_p;
}
```
if (temp_p == NULL || temp_p->data > value) {
    if (temp_p == head_p)
        pthread_mutex_unlock(&head_p_mutex);
    if (temp_p != NULL)
        pthread_mutex_unlock(&(temp_p->mutex));
    return 0;
} else {
    if (temp_p == head_p)
        pthread_mutex_unlock(&head_p_mutex);
    pthread_mutex_unlock(&(temp_p->mutex));
    return 1;
}
/* Member */
Issues

- This is much more complex than the original Member function.
- It is also much slower, since, in general, each time a node is accessed, a mutex must be locked and unlocked.
- The addition of a mutex field to each node will substantially increase the amount of storage needed for the list.
Pthreads Read-Write Locks

• Neither of our multi-threaded linked lists exploits the potential for simultaneous access to any node by threads that are executing Member.

• The first solution only allows one thread to access the entire list at any instant.

• The second only allows one thread to access any given node at any instant.
Pthreads Read-Write Locks

• A read-write lock is somewhat like a mutex except that it provides two lock functions.

• The first lock function locks the read-write lock for reading, while the second locks it for writing.
Pthreads Read-Write Locks

• So multiple threads can simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function.

• If any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function.
Pthreads Read-Write Locks

• If any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.
Protecting our linked list functions

```c
pthread_rwlock_t rwlock;
pthread_rwlock_init(&rwlock, NULL);

pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);

...  
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);

...  
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);

pthread_rwlock_destroy(&rwlock);
```
## Linked List Performance

Time in seconds

<table>
<thead>
<tr>
<th>Implementation</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Write Locks</td>
<td>0.213</td>
<td>0.123</td>
<td>0.098</td>
<td>0.115</td>
</tr>
<tr>
<td>One Mutex for Entire List</td>
<td>0.211</td>
<td>0.450</td>
<td>0.385</td>
<td>0.457</td>
</tr>
<tr>
<td>One Mutex per Node</td>
<td>1.680</td>
<td>5.700</td>
<td>3.450</td>
<td>2.700</td>
</tr>
</tbody>
</table>

100,000 ops/thread
99.9% Member
0.05% Insert
0.05% Delete
# Linked List Performance

Time in seconds

<table>
<thead>
<tr>
<th>Implementation</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Write Locks</td>
<td>2.48</td>
<td>4.97</td>
<td>4.69</td>
<td>4.71</td>
</tr>
<tr>
<td>One Mutex for Entire List</td>
<td>2.50</td>
<td>5.13</td>
<td>5.04</td>
<td>5.11</td>
</tr>
<tr>
<td>One Mutex per Node</td>
<td>12.00</td>
<td>29.60</td>
<td>17.00</td>
<td>12.00</td>
</tr>
</tbody>
</table>

- 100,000 ops/thread
- 80% Member
- 10% Insert
- 10% Delete
Let’s Have A Summary of Widely used APIs
More than 100 subroutines!
**Thread Management**

* pthread_create
  - * pthread_t *
  - const pthread_attr_t *
  - void *(*start_routine)(void*),
  - void * arg

- NULL: keep the default
- specified only at thread creation time

- The main steps in setting attributes:
  - pthread_attr_t tattr
  - pthread_attr_init(&tattr)
  - pthread_attr_*(&tattr,SOME_ATTRIBUTE_VALUE_PARAMETER)
Threads Cheaper than Processes

<table>
<thead>
<tr>
<th>Platform</th>
<th>fork()</th>
<th></th>
<th>pthread_create()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>user</td>
<td>sys</td>
<td>real</td>
</tr>
<tr>
<td>AMD 2.4 GHz Opteron (8cpus/node)</td>
<td>41.07</td>
<td>60.08</td>
<td>9.01</td>
<td>0.66</td>
</tr>
<tr>
<td>IBM 1.9 GHz POWER5 p5-575 (8cpus/node)</td>
<td>64.24</td>
<td>30.78</td>
<td>27.68</td>
<td>1.75</td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4 (8cpus/node)</td>
<td>104.05</td>
<td>48.64</td>
<td>47.21</td>
<td>2.01</td>
</tr>
<tr>
<td>INTEL 2.4 GHz Xeon (2 cpus/node)</td>
<td>54.95</td>
<td>1.54</td>
<td>20.78</td>
<td>1.64</td>
</tr>
<tr>
<td>INTEL 1.4 GHz Itanium2 (4 cpus/node)</td>
<td>54.54</td>
<td>1.07</td>
<td>22.22</td>
<td>2.03</td>
</tr>
</tbody>
</table>
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

    iret1 = pthread_create( &thread1, NULL,
                          print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, 
                          print_message_function, (void*) message2);

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Threads terminate by:
- explicitly calling `pthread_exit`
- letting the function return
- a call to the function exit which will terminate the process including any threads.
- canceled by another thread via the `pthread_cancel` routine
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

void *PrintHello(void *threadid) {
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;

    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\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);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
What is wrong about the following code?

```c
int rc;
long t;
for(t=0; t<NUM_THREADS; t++) {
    printf("Creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
    ... }
```
The `pthread_join()` subroutine blocks the calling thread until the specified thread id thread terminates.
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4

void *BusyWork(void *t)
{
    int i;
    long tid;
    double result = 0.0;
    tid = (long)t;
    printf("Thread %ld starting...\n", tid);
    for (i = 0; i < 1000000; i++)
    {
        result = result + sin(i) * tan(i);
    }
    printf("Thread %ld done. Result = %e\n", tid, result);
    pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc;
    long t;
    void *status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for (t = 0; t < NUM_THREADS; t++)
    {
        printf("Main: creating thread %ld\n", t);
        rc = pthread_create(&thread[t], &attr, BusyWork, (void*) t);
        if (rc)
        {
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    /* Free attribute and wait for the other threads */
    pthread_attr_destroy(&attr);
    for (t = 0; t < NUM_THREADS; t++)
    {
        rc = pthread_join(thread[t], &status);
    }
}
```
Important!

- Call `pthread_join()` or `pthread_detach()` for every thread that is created joinable
  - so that the system can reclaim all resources associated with the thread
  - `int pthread_detach(pthread_t threadid);` indicates that system resources for the specified thread should be reclaimed when the thread ends

- Failure to join or to detach threads → memory and other resource leaks until the process ends
How about the stack?

- Default thread stack size varies greatly.
- Safe and portable programs do not depend upon the default stack limit.
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;

void *dowork(void *threadid)
{
    double A[N][N];
    int i,j;
    long tid;
    size_t mystacksize;

    tid = (long)threadid;
    pthread_attr_getstacksize (&attr, &mystacksize);
    printf("Thread %ld: stack size = %li bytes \n", tid, mystacksize);
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            A[i][j] = ((i*j)/3.452) + (N-i);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t threads[NTHREADS];
    size_t stacksize;
    int rc;
    long t;

    pthread_attr_init(&attr);
    pthread_attr_getstacksize (&attr, &stacksize);
    printf("Default stack size = %li\n", stacksize);
    stacksize = sizeof(double)*N*N+MEGEXTRA;
    printf("Amount of stack needed per thread = %li\n", stacksize);
    pthread_attr_setstacksize (&attr, stacksize);
    printf("Creating threads with stack size = %li bytes\n", stacksize);
    for (t=0; t<NTHREADS; t++)
        rc = pthread_create(&threads[t], &attr, dowork, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }

    printf("Created %ld threads.\n", t);
    pthread_exit(NULL);
}
• Mutex = Mutual Exclusion
• One of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
• acts like a lock protecting access to a shared data resource
• only one thread can lock (or own) a mutex variable at any given time.
A typical sequence in the use of a mutex is as follows:

• Create and initialize a mutex variable
• Several threads attempt to lock the mutex
• Only one succeeds and that thread owns the mutex
• The owner thread performs some set of actions
• The owner unlocks the mutex
• Another thread acquires the mutex and repeats the process
• Finally the mutex is destroyed

It is up to the code writer to insure that the necessary threads all make the mutex lock and unlock calls correctly.
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/*
The following structure contains the necessary information to allow the function "dotprod" to access its input data and place its output into the structure.
*/

typedef struct
{
    double    *a;
    double    *b;
    double    sum;
    int        veclen;
} DOTDATA;

/* Define globally accessible variables and a mutex */

#define NUMTHRDS 4
#define VECLLEN 100

DOTDATA dotstr;

pthread_t callThd[NUMTHRDS];

void *dotprod(void *arg)
{

    Lock a mutex prior to updating the value in the shared structure, and unlock it upon updating.
 *
    pthread_mutex_lock (&mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock (&mutexsum);

    pthread_exit((void*) 0);
}

int main (int argc, char *argv[])
{
    long i;
    double  *a, *b;
    void   *status;
    pthread_attr_t attr;

    /* Assign storage and initialize values */
    a = (double*) malloc (NUMTHRDS*VECLLEN*sizeof(double));
    b = (double*) malloc (NUMTHRDS*VECLLEN*sizeof(double));

    for (i=0; i<VECLLEN*NUMTHRDS; i++)
    {
        a[i]=1.0;
        b[i]=a[i];
    }

    dotstr.veclen = VECLLEN;
    dotstr.a = a;
    dotstr.b = b;
    dotstr.sum=0;

    pthread_mutex_init(&mutexsum, NULL);

    /*
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
     *
• While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.

• A condition variable is always used in conjunction with a mutex lock.

• Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This is very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
Steps for Using Condition Variables

- **Main thread:**
  - Declare and initialize global data/variables which require synchronization
  - Declare and initialize a condition variable
  - Declare and initialize an associated mutex
  - Create threads A and B to do work

- **Thread A**
  - Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
  - Lock associated mutex and check value of a global variable
  - Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B.
    - a call to `pthread_cond_wait()` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
  - When signalled, wake up. Mutex is automatically and atomically locked.
  - Explicitly unlock mutex
  - Continue

- **Thread B**
  - Do work
  - Lock associated mutex
  - Change the value of the global variable that Thread-A is waiting upon.
  - Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
  - Unlock mutex.
  - Continue
• Condition variables must be declared with type `pthread_cond_t`  
• must be initialized before they can be used.  
• There are two ways to initialize a condition variable:  
  • Statically, when it is declared. For example:  
    ```c
    pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
    ```  
  • Dynamically, with the `pthread_cond_init()` routine.  
• `pthread_cond_destroy()` should be used to free a condition variable that is no longer needed.
int count = 0;
int thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *watch_count(void *t) {
    long my_id = (long)t;
    pthread_mutex_lock(&count_mutex);

    while (count<COUNT_LIMIT) {
        pthread_cond_wait(&count_threshold_cv, &count_mutex);
        count += 125;
    }
    pthread_mutex_unlock(&count_mutex);
    pthread_exit(NULL);
}

void *inc_count(void *t) {
    int i;
    long my_id = (long)t;

    for (i=0; i<TCOUNT; i++) {
        pthread_mutex_lock(&count_mutex);
        count++;
        if (count == COUNT_LIMIT)
            pthread_cond_signal(&count_threshold_cv);
        pthread_mutex_unlock(&count_mutex);
    }
    /* Do some "work" so threads can alternate on mutex lock */
    sleep(1); }
    pthread_exit(NULL);
}
Definition: Synchronization is an enforcing mechanism used to impose constraints on the order of execution of threads, in order to coordinate thread execution and manage shared data.

By now you must have realized that we have 3 synchronization mechanisms:

- Mutexes
- Condition variables
- Joins
Semaphores

• permit a limited number of threads to execute a section of the code
• similar to mutexes
• should include the `semaphore.h` header file
• semaphore functions have `sem_` prefixes
Basic Semaphore functions

• creating a semaphore:
  – int sem_init(sem_t *sem, int pshared, unsigned int value)
  – initializes a semaphore object pointed to by sem
  – pshared is a sharing option; a value of 0 means the semaphore is local to the calling process
  – gives an initial value value to the semaphore

• terminating a semaphore:
  – int sem_destroy(sem_t *sem)
  – frees the resources allocated to the semaphore
  – usually called after pthread_join()
Basic Semaphore functions

• int `sem_post(sem_t *sem)`
  - atomically increases the value of a semaphore by 1, i.e., when 2 threads call `sem_post` simultaneously, the semaphore's value will also be increased by 2.

• int `sem_wait(sem_t *sem)`
  - atomically decreases the value of a semaphore by 1
  - If the value is 0 then the thread will block waiting it to become 1
Semaphores

- Only positive values (or 0)
- Its operations are atomic
- Cannot read it except at initializations
- Main usage
  - mutual exclusion
  - synchronization
```c
#include <pthread.h>
#include <semaphore.h>
...

void *thread_function( void *arg )
{
    sem_wait( &semaphore );
    perform_task_when_sem_open();
    ...
    pthread_exit( NULL );
}

int main()
{
    int tmp;
    ...
    // initialize the semaphore
    tmp = sem_init( &semaphore, 0, 0 );
    ...
    // create threads
    pthread_create( &thread[i], NULL, thread_function, NULL );
    ...
    while ( still_has_something_to_do() )
    {
        sem_post( &semaphore );
        ...
    }
    ...
    pthread_join( thread[i], NULL );
    sem_destroy( &semaphore );
    return 0;
}
```

The Problem With Threads

• Paper by Edward Lee, 2006
• The author argues:
  – “From a fundamental perspective, threads are seriously flawed as a computation model”
  – “Achieving reliability and predictability using threads is essentially impossible for many applications”
• The main points:
  – Our abstraction for concurrency does not even vaguely resemble the physical world.
  – Threads are dominating but not the best approach in every situation
  – Yet threads are suitable for embarrassingly parallel applications
The Problem With Threads

• The logic of the paper:
  – Threads are nondeterministic
  – Why shall we use nondeterministic mechanisms to achieve deterministic aims??
  – The job of the programmer is to prune this nondeterminism.
  – This leads to poor results

Do you agree or disagree with the author?
Conclusions

• Processes $\rightarrow$ threads $\rightarrow$ processors
• Pthreads assume shared memory.
• There are many more APIs for Pthreads. We just studied the main ones here.