COMP/CS 605: Introduction to Parallel Computing
Lecture : Controlling Access & Synchronization

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Presented: 04/06/17
Updated: 04/03/17

Table of Contents

Busy-Waiting

- A thread repeatedly tests a condition

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

flag initialized to 0 by main thread

- Thread 1 cannot enter critical section until Thread 0 has finished.
- Beware of optimizing compilers:
  They can optimize code and rearrange order of code affecting busy-wait cycle.

Pthreads: global sum with busy-waiting

```
*------------------------------------------------------------------
 * Function:      Thread_sum
 * Purpose:       Add in the terms computed by the thread running this
 * In arg:        rank
 * Ret val:       ignored
 * Globals in:    n, thread_count
 * Global in/out: sum
 */
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }
   return NULL;
}  /* Thread_sum */
```

Thread 1 spins until Thread 0 finishes - could waste resources.
Add in logic for last thread to reset flag

```
[tuckoo]  mthomas% ./pth_pi_busy1 8 100000
With n = 100000 terms,
   Multi-threaded estimate of pi  = 3.141582653589717
   Elapsed time = 1.306486e-02 seconds
   Single-threaded estimate of pi = 3.141582653589720
   Elapsed time = 4.179478e-04 seconds
   Math library estimate of pi    = 3.141592653589793
```

```
[tuckoo] mthomas% ./pth_pi_busy1 8 10000000
With n = 10000000 terms,
   Multi-threaded estimate of pi  = 3.141592553589788
   Elapsed time = 9.265280e-01 seconds
   Single-threaded estimate of pi = 3.141592553589792
   Elapsed time = 4.049492e-02 seconds
   Math library estimate of pi    = 3.141592653589793
```

Note: Serial version is faster than threaded version!

Pthreads: Controlling Access to Shared Variable

```
*-------------------------------------------------------------------
 * Function:      Thread_sum
 * Purpose:       Add in the terms computed by the thread running this
 * In arg:        rank
 * Ret val:       ignored
 * Globals in:    n, thread_count
 * Global in/out: sum
 */
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

Define local sum, then update global sum in a critical section after loop

Output after using local sum var; moving critical section to after loop.

```
[mthomas@tuckoo ch4]$ ./pth_pi_busy2 8 1000000
With n = 1000000 terms,
   Multi-threaded estimate of pi  = 3.141591653589728
   Elapsed time = 1.039195e-02 seconds
   Single-threaded estimate of pi = 3.141591653589774
   Elapsed time = 1.185608e-02 seconds
   Math library estimate of pi    = 3.141592653589793

```

```

[mthomas@tuckoo ch4]$ ./pth_pi_busy2 8 10000000
With n = 10000000 terms,
   Multi-threaded estimate of pi  = 3.141592553589832
   Elapsed time = 3.278208e-02 seconds
   Single-threaded estimate of pi = 3.141592553589792
   Elapsed time = 1.130030e-01 seconds
   Math library estimate of pi    = 3.141592653589793
```

Note: Serial and threaded timings are closer

Mutexes

- A thread that is busy-waiting may continually use the CPU accomplishing nothing.
- Mutex (mutual exclusion) is a special type of variable that can be used to restrict access to a critical section to a single thread at a time.
- Used to guarantee that one thread "excluded" all other threads while it executes the critical section.
- The Pthreads standard includes a special type for mutexes: *pthread_mutex_t* .

```
int pthread_mutex_init (
pthread_mutex_t *        mutex_p /* out */
pthread_mutexattr_t *    attr_p /* out */ );
```

Mutexes

- When a thread is finished executing the code in a critical section, it should call

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p /* in/out */);
```

- calling thread waits until no other thread is in critical section
- steps:
  - declare global mutex variable
  - have main thread init variable
  - use pthread_mutex_lock work use pthread_mutex_unlock pair
  - this is a **blocking** call

main defines global mutex variable, inits and destroys

```
pthread_mutex_t mutex;       /*declare global mutex variable */

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;
   double start, finish, elapsed;

   /* Get number of threads from command line */
   Get_args(argc, argv);
   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));

  /**********************************************/
   pthread_mutex_init(&mutex, NULL);

   sum = 0.0;
   GET_TIME(start);
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,Thread_sum, (void*)thread);

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);
   GET_TIME(finish);
   elapsed = finish - start;
   sum = 4.0*sum;

   GET_TIME(start);     sum = Serial_pi(n);       GET_TIME(finish);
   elapsed = finish - start;

  /**********************************************/
  pthread_mutex_destroy(&mutex);

   free(thread_handles);
   return 0;      }  /* end main */
```

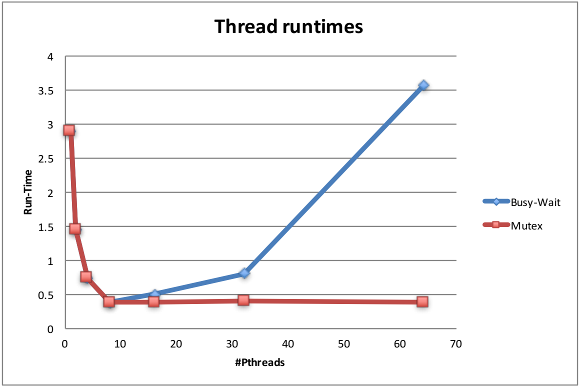
function computes local my_sum, then uses mutex_lock for control

```
/*------------------------------------------------------------------*/
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \texttt{thread\_count}$$

Run-times (in seconds) of π programs using n = 108 terms on a system with two four-core processors.


Thread runtimes
Run-Time
#Pthreads
4
3.5
3
2.5
2
1.5
1
0.5
0
0
10
20
30
40
50
60
70
Busy-Wait
Mutex

A few observations

- Results on OS X are similar to text. What would happen on tuckoo?
- The order in which threads execute is random
- This is effectively a barrier, so you expect mutex performance to degrade ($Nthreads > Ncores$)
- if $T\frac{T_{serial}}{T_{parallel}} \approx \quad threadcount$ then you have *Speedup*

Time	flag	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
0	0	crit sect	busy wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy wait	susp
2	2	—	terminate	susp	busy wait	busy wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy wait

Possible sequence of events with busy-waiting and more threads than cores.

Producer-Consumer Model, Synchronization and Semaphores

- Busy-waiting enforces the order in which threads access a critical section.
- Using 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.
- Trade-off between safety (mutex) and control (busy-wait) and performance.

Global sum function that uses a mutex.

```
/* 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 */
```

Problem: Matrix-Matrix multiplication is not commutative.

First attempt at sending messages using Pthreads

```c
/* messages has type char**. It's allocated in main. */
/* Each entry is set to NULL in main.                  */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + thread_count - 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;

   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n", my_rank, source);

   return NULL;
}  /* Send_msg */
```

$$[P_{source}] \rightarrow [P_{myrank}] \rightarrow [P_{destination}]$$

pth_msg.c

```
/* File:     pth_msg.c
 *
 * Purpose:  Illustrate a synchronization problem with pthreads:  create
 *           some threads, each of which creates and prints a message.
 *
 * Input:    none
 * Output:   message from each thread
 *
 * Compile:  gcc -g -Wall -o pth_msg pth_msg.c -lpthread
 * Usage:    pth_msg <thread_count>
 *
 * IPP:      Section 4.7 (pp. 172 and ff.)
 */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

const int MAX_THREADS = 1024;
const int MSG_MAX = 100;

/* Global variables:  accessible to all threads */
int thread_count;
char** messages;

void Usage(char* prog_name);
void *Send_msg(void* rank);  /* Thread function */
```

pth_msg.c

```
/*-------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;

   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   if (thread_count <= 0 || thread_count > MAX_THREADS) Usage(argv[0]);

   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
   messages = (char**) malloc(thread_count*sizeof(char*));
   for (thread = 0; thread < thread_count; thread++)
      messages[thread] = NULL;

   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], (pthread_attr_t*) NULL,
          Send_msg, (void*) thread);

   for (thread = 0; thread < thread_count; thread++) {
      pthread_join(thread_handles[thread], NULL);
   }

   for (thread = 0; thread < thread_count; thread++)
      free(messages[thread]);
   free(messages);

   free(thread_handles);
   return 0;
}  /* main */
```

pth_msg.c

```
/*-------------------------------------------------------------------
 * Function:    Usage
 * Purpose:     Print command line for function and terminate
 * In arg:      prog_name
 */
void Usage(char* prog_name) {

   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   exit(0);
}  /* Usage */


/*------------------------------------------------------------------
 * Function:       Send_msg
 * Purpose:        Create a message and ``send'' it by copying it
 *                 into the global messages array.  Receive a message
 *                 and print it.
 * In arg:         rank
 * Global in:      thread_count
 * Global in/out:  messages
 * Return val:     Ignored
 * Note:           The my_msg buffer is freed in main
 */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + thread_count - 1) % thread_count;
   char* my_msg = (char*) malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;

   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n", my_rank, source);
```

Sending Messages Using Pthreads: mutex does not control when messages are sent so some get lost.

```
[gidget:intro-par-pgming-pacheco/ipp-source/ch4] mthomas% ./pth_msg 4
Thread 0 > No message from 3
Thread 1 > Hello to 1 from 0
Thread 3 > No message from 2
Thread 2 > Hello to 2 from 1
[gidget:intro-par-pgming-pacheco/ipp-source/ch4] mthomas% ./pth_msg 10
Thread 0 > No message from 9
Thread 3 > No message from 2
Thread 2 > No message from 1
Thread 1 > Hello to 1 from 0
Thread 5 > No message from 4
Thread 4 > Hello to 4 from 3
Thread 6 > Hello to 6 from 5
Thread 7 > Hello to 7 from 6
Thread 9 > No message from 8
Thread 8 > Hello to 8 from 7
```

Possible Solutions

- Try busy-wait, but we will waste cpu time.
  $while\,(messages\,[my\_rank] == NULL)$
  $printf\,(\text{"}Thread\quad \%d\ >\ \%s\text{"}, my\_rank, messages\,[my\_rank])$
- There is no MPI style send/recv pairs
- Find way to notify destination thread, not easy to do with mutexes
  $messages\,[dest] = my\_msg;$
  Notify thread $[P_{dest}]$ to enter block
  ...
  Await notification from thread $[P_{source}]$
  $printf\,(\text{"}Thread\,\%d\ >\ \%s\text{"}, my\_rank, messages\,[my\_rank])$
- Solution: Semaphores

What is a semaphore?

```
Ask.com:
semaphore
Noun:
A system of sending messages by holding the arms or two flags or poles
positions according to an alphabetic code.
Verb:
Send (a message) by semaphore or by signals resembling semaphore.
Synonyms:
noun.  traffic light - traffic lights - signal
verb.  signal

Wikepedia:
In computer science, a semaphore is a variable or abstract data
type that provides a simple but useful abstraction for controlling
access by multiple multiple processes to a common  resource in
a parallel programming environment.
```

Possible Solutions

- unsigned int
- binary semaphore = 0,1 == locked,unlocked
- usage:
  1. *init* semaphore to 1 (unlocked)
  2. before critical block, thread places call to *sem_wait*
  3. if *semaphore* > 1 , decrement semaphore and enter critical block
  4. when done, call *sem_post*, which increments semaphore for next thread
- semaphores have no ownership: any thread can modify them
- semaphores are not part of Pthreads, so need to include *semaphore.h*

Syntax of the various semaphore functions

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

```
#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 */);
```

Send_msg using semaphore

```
/*-------------------------------------------------------------------
 * Function:       Send_msg
 * Purpose:        Create a message and ‘‘send’’ it by copying it
 *                 into the global messages array.  Receive a message
 *                 and print it.
 * In arg:         rank
 * Global in:      thread_count
 * Global in/out:  messages, semaphores
 * Return val:     Ignored
 * Note:           The my_msg buffer is freed in main
 */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) \% thread_count;
   char* my_msg = (char*) malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to \%ld from \%ld", dest, my_rank);
   messages[dest] = my_msg;
   sem_post(&semaphores[dest]);  /* "Unlock" the semaphore of dest */

   sem_wait(&semaphores[my_rank]);  /* Wait for our semaphore to be unlocked */
   printf("Thread \%ld > \%s\n", my_rank, messages[my_rank]);

   return NULL;
}  /* Send_msg */
```

Send_msg output on tuckoo using PBS node

```
[mthomas@tuckoo ch4]$ cat pth_msg_sem.o63124
Thread 1 > Hello to 1 from 0
Thread 2 > Hello to 2 from 1
Thread 5 > Hello to 5 from 4
Thread 3 > Hello to 3 from 2
Thread 4 > Hello to 4 from 3
Thread 6 > Hello to 6 from 5
Thread 7 > Hello to 7 from 6
Thread 8 > Hello to 8 from 7
Thread 9 > Hello to 9 from 8
Thread 10 > Hello to 10 from 9
Thread 11 > Hello to 11 from 10
Thread 12 > Hello to 12 from 11
Thread 13 > Hello to 13 from 12
Thread 14 > Hello to 14 from 13
Thread 15 > Hello to 15 from 14
Thread 16 > Hello to 16 from 15
Thread 17 > Hello to 17 from 16
Thread 18 > Hello to 18 from 17
Thread 19 > Hello to 19 from 18
Thread 20 > Hello to 20 from 19
Thread 21 > Hello to 21 from 20
Thread 22 > Hello to 22 from 21
Thread 23 > Hello to 23 from 22
Thread 24 > Hello to 24 from 23
Thread 25 > Hello to 25 from 24
Thread 26 > Hello to 26 from 25
Thread 27 > Hello to 27 from 26
Thread 28 > Hello to 28 from 27
Thread 29 > Hello to 29 from 28
Thread 0 > Hello to 0 from 29
```

Send_msg output on OS Mountain Lion

```
[gidget] mthomas\% ./pth_msg_sem 30
Thread 0 > (null)
Thread 2 > (null)
Thread 1 > Hello to 1 from 0
Thread 3 > Hello to 3 from 2
Thread 4 > Hello to 4 from 3
Thread 5 > Hello to 5 from 4
Thread 6 > Hello to 6 from 5
Thread 7 > Hello to 7 from 6
Thread 8 > Hello to 8 from 7
Thread 11 > Hello to 11 from 10
Thread 10 > (null)
Thread 9 > Hello to 9 from 8
Thread 12 > Hello to 12 from 11
Thread 13 > Hello to 13 from 12
Thread 14 > Hello to 14 from 13
Thread 15 > Hello to 15 from 14
Thread 16 > Hello to 16 from 15
Thread 17 > Hello to 17 from 16
Thread 19 > (null)
Thread 18 > Hello to 18 from 17
Thread 20 > Hello to 20 from 19
Thread 21 > Hello to 21 from 20
Thread 22 > Hello to 22 from 21
Thread 23 > Hello to 23 from 22
Thread 24 > Hello to 24 from 23
Thread 25 > Hello to 25 from 24
Thread 26 > Hello to 26 from 25
Thread 27 > Hello to 27 from 26
Thread 28 > Hello to 28 from 27
Thread 29 > Hello to 29 from 28
```

Barriers and Condition Variables

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.
- *Barriers* are used for timing, debugging, and synchronization of the threads
- Used to make sure that they are all at the same point in a program
- Not part of the Pthreads standard, so have to build customized barrier

Using barriers to time the slowest thread

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish - my_start;

elapsed = Maximum of my_elapsed values;
```

Using barriers for debugging

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
   fflush(stdout);
}
```

Busy-waiting and a Mutex

- Implementing a barrier using busy-waiting and a mutex is straightforward.
- We use a shared counter protected by the mutex.
- When the counter indicates that every thread has entered the critical section, threads can leave the critical section.

XXX

Busy-waiting and a Mutex

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&barrier_mutex);
   counter++;
   pthread_mutex_unlock(&barrier_mutex);
   while (counter < thread_count);
   . . .
}
```

We need one counter variable for each instance of the barrier, otherwise problems are likely to occur.

 52

PE's could still end up spinning. Issue with global mutex counter: not all threads will see its value, could result in hung processes.

Implementing a barrier with semaphores

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
   . . .
```

Condition Variables

- A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.
- 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

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

Send_msg output on OS Mountain Lion

```
API:
pthread_cond_init (condition,attr)   -- dynamically initialize condition variables
pthread_cond_destroy (condition)  -- destroy  condition variables
pthread_condattr_init (attr)
pthread_condattr_destroy (attr)

pthread_mutex_lock (mutex) --  used by a thread to acquire a lock on the specified mutex variable
pthread_mutex_trylock (mutex)
pthread_mutex_unlock (mutex)

pthread_cond_wait (condition,mutex)  -- blocks the calling thread until the specified condition is signalled
pthread_cond_signal (condition)          -- signal (or wake up) another thread which is waiting on the condition variable.
pthread_cond_broadcast (condition)   -- use instead of pthread_cond_signal() if more than one thread is waiting
```

Implementing a barrier with condition variables

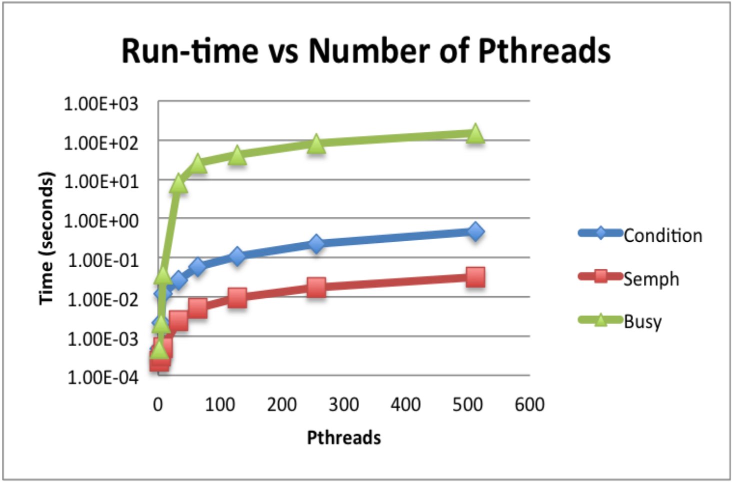
```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while (pthread_cond_wait(&cond_var, &mutex) != 0);
   }
   pthread_mutex_unlock(&mutex);
   . . .
}
```

Comparing three barrier methods

pthreads	pth_cond_bar	pth_sem_bar	pth_busy_bar
2	4.87E-04	2.36E-04	4.66E-04
4	2.24E-03	3.14E-04	2.15E-03
8	1.21E-02	4.95E-04	3.88E-02
32	2.65E-02	2.53E-03	8.22E+00
64	6.03E-02	5.12E-03	2.60E+01
128	1.10E-01	9.60E-03	4.12E+01
256	2.20E-01	1.79E-02	8.04E+01
512	4.67E-01	3.18E-02	1.49E+02

/Users/mthom
Teaching-Material/Topics/Pthreads/pach-ch4-imgs/Slide0

Comparing three barrier methods



Implementing a barrier with condition variables

```
/* 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);
   . . .
}
```

PThread Condition Barrier Code Example

```
int thread_count;
int barrier_thread_count = 0;
pthread_mutex_t barrier_mutex;
pthread_cond_t ok_to_proceed;

void Usage(char* prog_name);
void *Thread_work(void* rank);

/*-------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;
   double start, finish;

   if (argc != 2)
      Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));
   pthread_mutex_init(&barrier_mutex, NULL);
   pthread_cond_init(&ok_to_proceed, NULL);

   GET_TIME(start);
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Thread_work, (void*) thread);

   for (thread = 0; thread < thread_count; thread++) {
      pthread_join(thread_handles[thread], NULL);
   }
   GET_TIME(finish);
   printf("Elapsed time = %e seconds\n", finish - start);
   pthread_mutex_destroy(&barrier_mutex);
   pthread_cond_destroy(&ok_to_proceed);
   free(thread_handles);
   return 0;
}  /* main */
```

PThread Condition Barrier Code Example

```
void *Thread_work(void* rank) {
#  ifdef DEBUG
   long my_rank = (long) rank;
#  endif
   int i;

   for (i = 0; i < BARRIER_COUNT; i++) {
      pthread_mutex_lock(&barrier_mutex);
      barrier_thread_count++;
      if (barrier_thread_count == thread_count) {
         barrier_thread_count = 0;
#        ifdef DEBUG
         printf("Thread %ld > Signalling other threads in barrier %d\n",
               my_rank, i);
         fflush(stdout);
#        endif
         pthread_cond_broadcast(&ok_to_proceed);
      } else {
         // Wait unlocks mutex and puts thread to sleep.
         //    Put wait in while loop in case some other
         // event awakens thread.
         while (pthread_cond_wait(&ok_to_proceed,
                   &barrier_mutex) != 0);
         // Mutex is relocked at this point.
#        ifdef DEBUG
         printf("Thread %ld > Awakened in barrier %d\n", my_rank, i);
         fflush(stdout);
#        endif
      }
      pthread_mutex_unlock(&barrier_mutex);
#     ifdef DEBUG
      if (my_rank == 0) {
         printf("All threads completed barrier %d\n", i);
         fflush(stdout);
      }
#     endif
   }
```

pthd_cond_bar.c output
arrival time into/out of barrier is non-deterministic

```
ipp.ch4/crit-sect] ./pth_cond_bar 4
Thread 3 > Signalling other threads in barrier 0
Thread 0 > Awakened in barrier 0
All threads completed barrier 0
Thread 1 > Awakened in barrier 0
Thread 2 > Awakened in barrier 0
Thread 2 > Signalling other threads in barrier 1
Thread 3 > Awakened in barrier 1
Thread 1 > Awakened in barrier 1
Thread 0 > Awakened in barrier 1
All threads completed barrier 1
Thread 0 > Signalling other threads in barrier 2
All threads completed barrier 2
Thread 2 > Awakened in barrier 2
Thread 1 > Awakened in barrier 2
Thread 3 > Awakened in barrier 2
Thread 3 > Signalling other threads in barrier 3
Thread 0 > Awakened in barrier 3
All threads completed barrier 3
Thread 2 > Awakened in barrier 3
Thread 1 > Awakened in barrier 3
Elapsed time = 5.729198e-04 seconds
```


```
ipp.ch4/crit-sect] ./pth_cond_bar 4
Thread 3 > Signalling other threads in barrier 0
Thread 0 > Awakened in barrier 0
All threads completed barrier 0
Thread 1 > Awakened in barrier 0
Thread 2 > Awakened in barrier 0
Thread 2 > Signalling other threads in barrier 1
Thread 3 > Awakened in barrier 1
Thread 1 > Awakened in barrier 1
Thread 0 > Awakened in barrier 1
All threads completed barrier 1
Thread 0 > Signalling other threads in barrier 2
All threads completed barrier 2
Thread 2 > Awakened in barrier 2
Thread 1 > Awakened in barrier 2
Thread 3 > Awakened in barrier 2
Thread 3 > Signalling other threads in barrier 3
Thread 0 > Awakened in barrier 3
All threads completed barrier 3
Thread 2 > Awakened in barrier 3
Thread 1 > Awakened in barrier 3
Elapsed time = 5.729198e-04 seconds
```