Table of Contents

1 Pthreads - Controlling Access and Synchronization
   - Pthreads: Producer/Consumer, Synchronization, Semaphores
   - Pthreads: Barriers and Condition Variables
2 Next Time
Busy-waiting enforces the order 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.
Problem with a mutex solution

/* 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;
}

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 */
```

\[
[P_{source}] \rightarrow [P_{myrank}] \rightarrow [P_{destination}]
\]
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.
  
  ```c
  while (messages[my_rank] == NULL)
  {
    printf(" Thread %d > %s", 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
  
  ```c
  messages[dest] = my_msg;
  Notify thread [P_dest] to enter block
  ```

  ```c
  \ldots
  ```

  ```c
  Await notification from thread [P_source]
  printf(" Thread %d > %s", 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 in certain 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

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

- unsigned int
- binary semaphore = 0,1 == locked,unlocked

usage:

1. \textit{init} semaphore to 1 (unlocked)
2. before critical block, thread places call to \texttt{sem\_wait}
3. if \texttt{semaphore} > 1, decrement semaphore and enter critical block
4. when done, call \texttt{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 \texttt{semaphore.h}
Syntax of the various semaphore functions

```c
#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

- used for timing, debugging, synchronization
- not part Pthreads, so have to build customized barrier
- we have looked at busy-wait and semaphores
- next we’ll look at using Pthreads objects: *condition variable*
Barriers

- 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.
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

```c
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.
Busy-waiting and a Mutex

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

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

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

```c
/* 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

```c
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

<table>
<thead>
<tr>
<th>pthreads</th>
<th>pth_cond_bar</th>
<th>pth_sem_bar</th>
<th>pth_busy_bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4.87E-04</td>
<td>2.36E-04</td>
<td>4.66E-04</td>
</tr>
<tr>
<td>4</td>
<td>2.24E-03</td>
<td>3.14E-04</td>
<td>2.15E-03</td>
</tr>
<tr>
<td>8</td>
<td>1.21E-02</td>
<td>4.95E-04</td>
<td>3.88E-02</td>
</tr>
<tr>
<td>32</td>
<td>2.65E-02</td>
<td>2.53E-03</td>
<td>8.22E+00</td>
</tr>
<tr>
<td>64</td>
<td>6.03E-02</td>
<td>5.12E-03</td>
<td>2.60E+01</td>
</tr>
<tr>
<td>128</td>
<td>1.10E-01</td>
<td>9.60E-03</td>
<td>4.12E+01</td>
</tr>
<tr>
<td>256</td>
<td>2.20E-01</td>
<td>1.79E-02</td>
<td>8.04E+01</td>
</tr>
<tr>
<td>512</td>
<td>4.67E-01</td>
<td>3.18E-02</td>
<td>1.49E+02</td>
</tr>
</tbody>
</table>
Comparing three barrier methods

Run-time vs Number of Pthreads

- Condition
- Sempf
- Busy
Next class: 10/28/14
HW #3 Due: 10/28/14
Quiz 2 (MPI) and Quiz 3 (Pthreads) will be held on 10/30/14.
Quiz 2: two parts:
  Part 1: take home assignment on 10/30/14, due 11/04/14
    [was 10/23/14, due 10/28]
    (50% of the grade)
  Part 2: in class quiz on TBA 10/30/14 [was 10/28]
    (50% of the grade)