Table of Contents

1. Pthreads: Controlling Access and Synchronization
   - PThreads - 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.
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.

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

```c
/* Shared variables */
int counter;    /* Initialize to 0 */
sem_t count_sem; /* Initialize to 1 */
sem_t barrier_sem; /* Initialize to 0 */

...,

/* 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

```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);
}
```
## 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

Time (seconds)

Condition
Semaphore
Busy
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);
}
```
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 */
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
}
return NULL; /* Thread_work */