Lecture 14: Mutual Exclusion, Locks and Barrier with PThreads

Concurrent and Multicore Programming

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

Review and Overview

- Thread basics and the POSIX Thread API
  - Process vs threads
- Thread creation, termination and joining
  - pthread_create, pthread_join and pthread_exit
  - **Boxing** multiple arguments in **struct** to pass to thread function
- Thread safety
- **Synchronization primitives in Pthreads**
  - **Mutual exclusion, locks and barrier**

Data Racing in a Multithread Program

Consider:

```
/* each thread to update shared variable
   best_cost */

if (my_cost < best_cost)
   best_cost = my_cost;
```

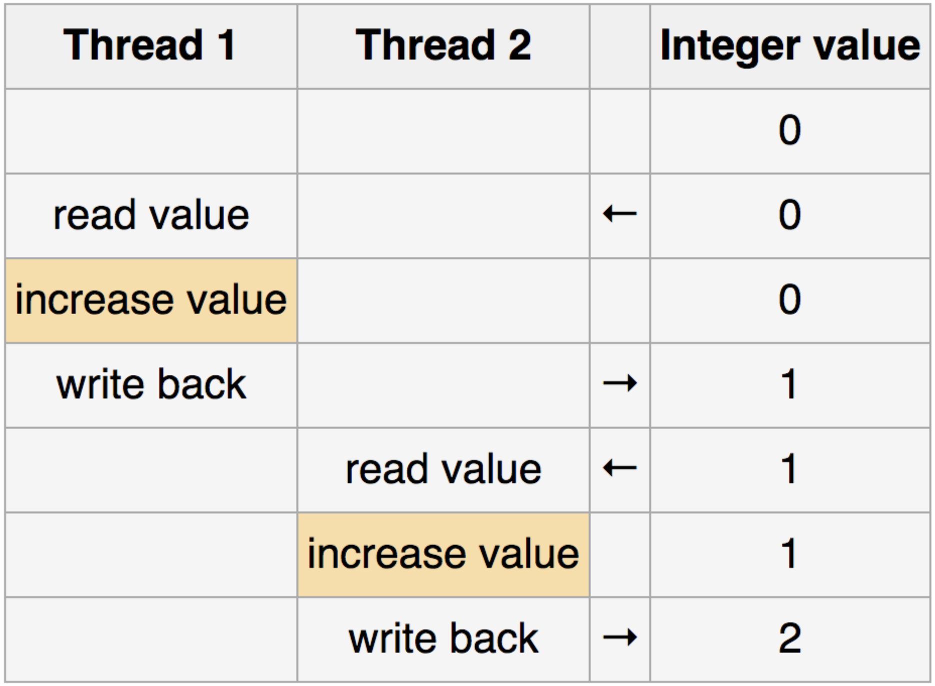
- two threads,
- the initial value of `best_cost` is 100,
- the values of `my_cost` are 50 and 75 for threads t1 and t2

T1	T2
`if (my_cost (50) < best_cost)`	`if (my_cost (75) < best_cost)`
`best_cost = my_cost;`	`best_cost = my_cost;`

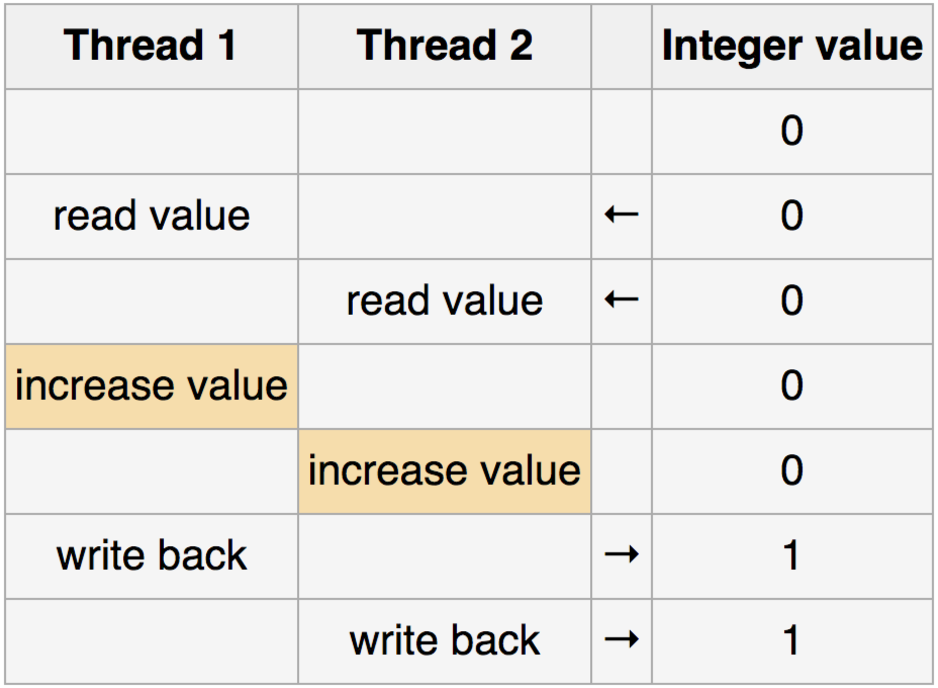
- The value of `best_cost` could be 50 or 75!
- The value 75 does not correspond to any serialization of the two threads.

Same Situation for Reading/Updating a Single Variable

```
int count = 0;
int * cp = &count;
....
*cp++; /* by two threads */
```



Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

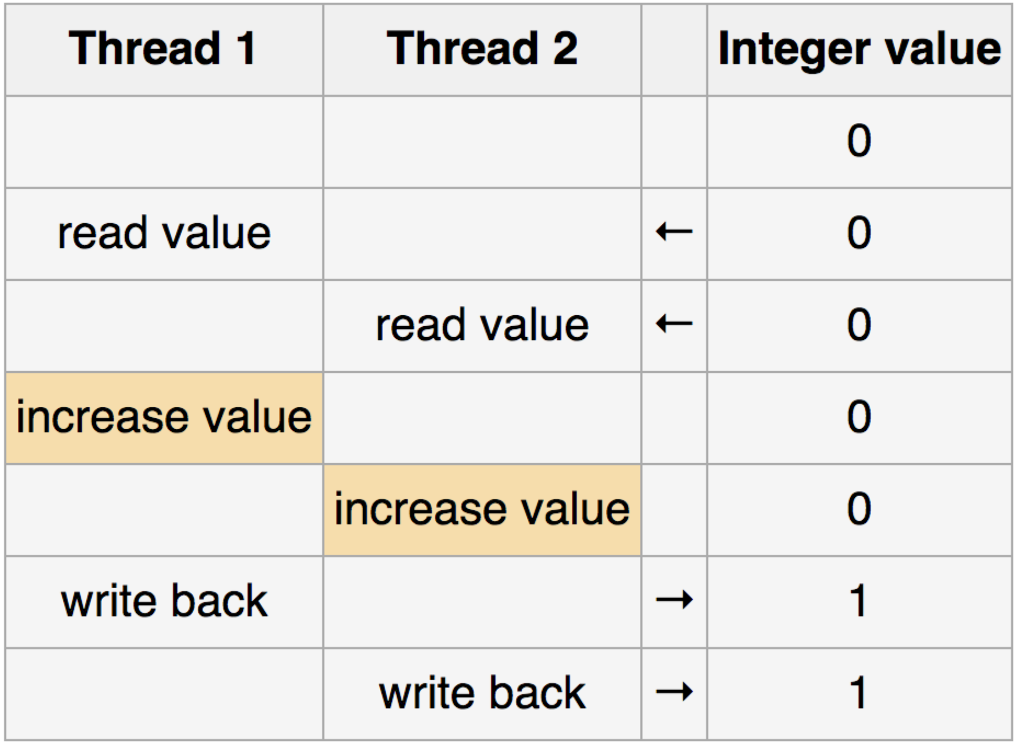


Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Pictures from wikipedia: http://en.wikipedia.org/wiki/Race_condition

Why this happens

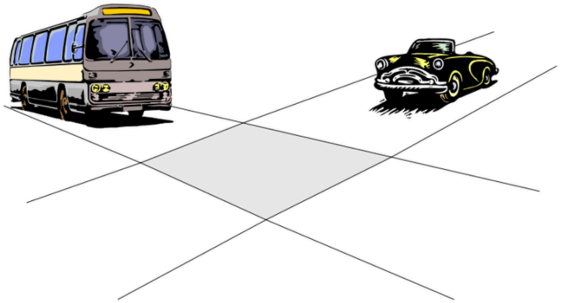
Read/write to the same location by the two threads interleaved



Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

General Solution: Critical Section and Mutual Exclusion

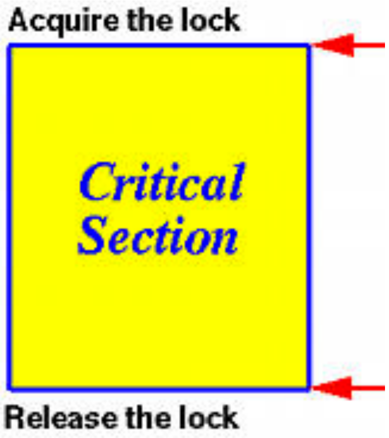
- Critical section = a segment that must be executed by only one thread at any time



```
if (my_cost < best_cost)
    best_cost = my_cost;
```

- Mutex locks protect critical sections in Pthreads
  - locked and unlocked
  - At any point of time, only one thread can acquire a mutex lock

- Using mutex locks
  - request lock before executing critical section
  - enter critical section when lock granted
  - release lock when leaving critical section



Mutual Exclusion using Pthread Mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex_lock);
int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);
int pthread_mutex_init (pthread_mutex_t  *mutex_lock,
    const pthread_mutexattr_t *lock_attr);
```

```
pthread_mutex_t cost_lock;
int main() {
      ...
   pthread_mutex_init(&cost_lock, NULL);
   pthread_create(&thhandle, NULL, find_best, …)
      ...
}
void *find_best(void *list_ptr) {
   ...
   pthread_mutex_lock(&cost_lock); // enter CS
   if (my_cost < best_cost)
      best_cost = my_cost;
   pthread_mutex_unlock(&cost_lock); // leave CS
}
```

Critical Section

pthread_mutex_lock blocks the calling thread if another thread holds the lock

When **pthread_mutex_lock** call returns

1. Mutex is locked, enter CS
2. Any other locking attempt (call to thread_mutex_lock) will cause the blocking of the calling thread

When **pthread_mutex_unlock** returns

1. Mutex is unlocked, leave CS
2. One thread who blocks on thread_mutex_lock call will acquire the lock and enter CS

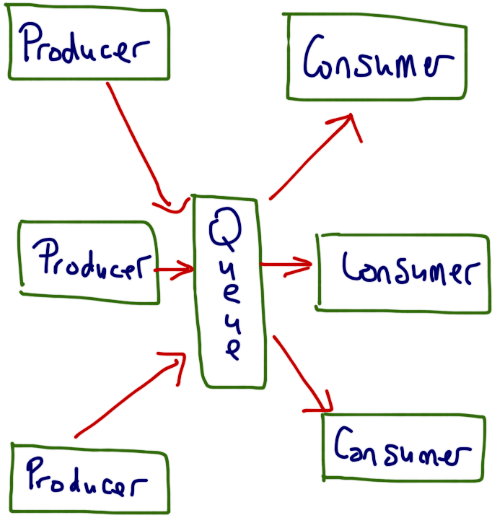
Producer-Consumer Using Locks

Constrains:

- The producer threads
  - must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
- The consumer threads
  - must not pick up tasks until there is something present in the shared data structure.
  - Individual consumer thread should pick up tasks one at a time

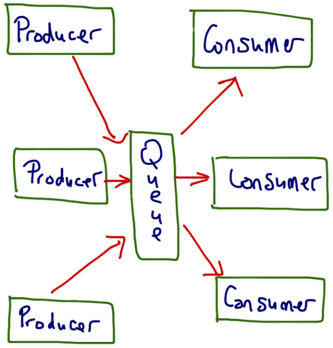
Contention:

- Between producers
- Between consumers
- Between producers and consumers



Producer-Consumer Using Locks

```
pthread_mutex_t task_queue_lock;
int task_available;
main() {
    ....
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    ....
}
```



```
void *producer(void *producer_thread_data) {
    ....
    while (!done()) {
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1; inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
```

Note the purpose of inserted and extraced variables

Critical Section

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0; extracted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
        process_task(my_task);
    }
}
```

Three Types of Mutexes

- Normal
  - Deadlocks if a thread already has a lock and tries a second lock on it.
- Recursive
  - Allows a single thread to lock a mutex as many times as it wants.
    - It simply increments a count on the number of locks.
  - A lock is relinquished by a thread when the count becomes zero.
- Error check
  - Reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case).

- The type of the mutex can be set in the attributes object before it is passed at time of initialization
  - pthread_mutex_attr_init

Overheads of Locking

- Locks enforce serialization
  - Thread must execute critical sections one after another
- Large critical sections can lead to significant performance degradation.
- Reduce the blocking overhead associated with locks using:

```
int pthread_mutex_trylock (
        pthread_mutex_t *mutex_lock);
```

- acquire lock if available
- return EBUSY if not available
- enables a thread to do something else if lock unavailable

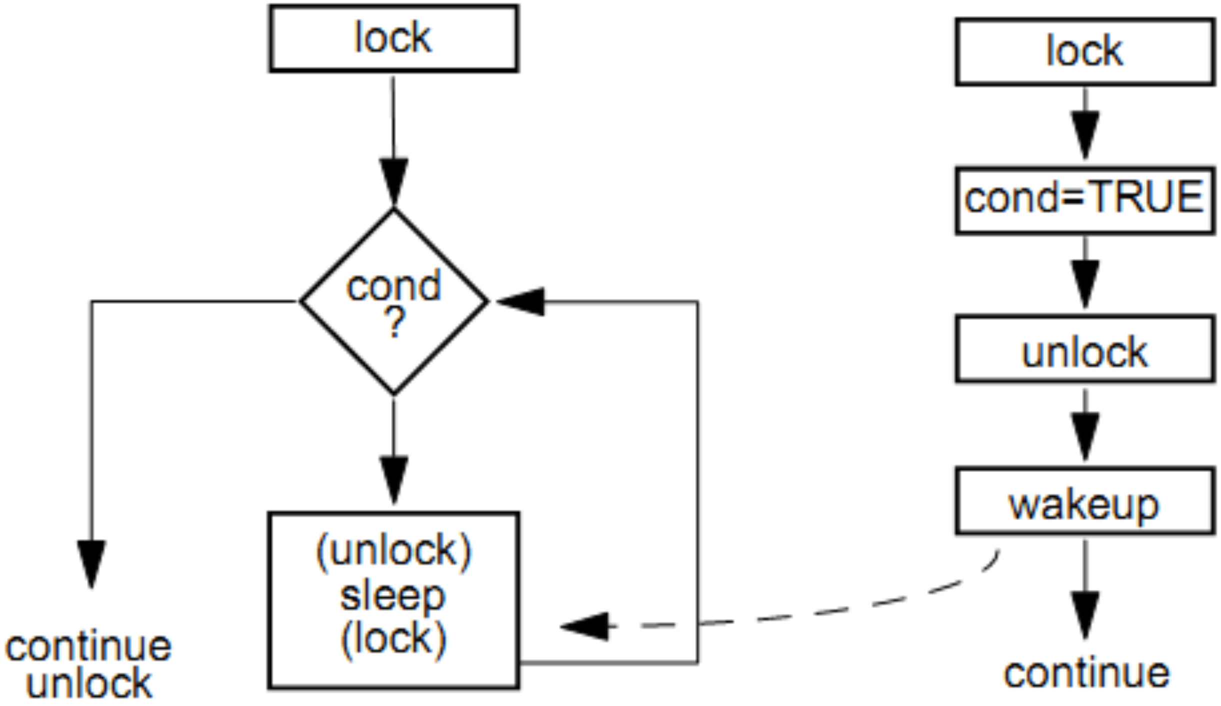
- pthread **trylock** typically much faster than **lock** on certain systems
  - It does not have to deal with queues associated with locks for multiple threads waiting on the lock.

Condition Variables for Synchronization

A condition variable: associated with a predicate and a mutex

- A sync variable for a condition, e.g. mybalance > 500

- A thread can block itself until a condition becomes true
  - Wh e it
- Whe another
  thre l other
  thre
- A co ciated with
  it.
  - A t



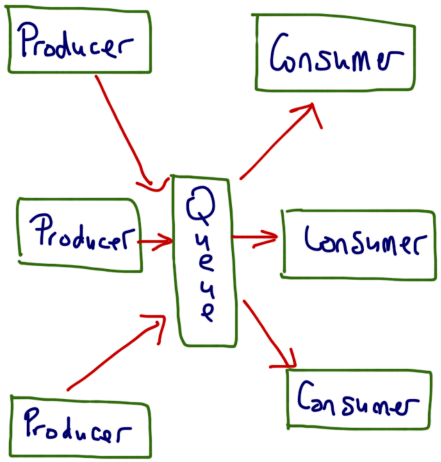
Using a Condition Variable

Condition Variables for Synchronization

```
/* the opaque data structure */
pthread_cond_t

/* initialization and destroying */
int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

/* block and release lock until a condition is true */
int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
pthread_mutex_t *mutex, const struct timespec *wtime);

/* signal one or all waiting threads that condition is true */
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Producer-Consumer Using Condition Variables

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */

main() {
  /* declarations and initializations */
  task_available = 0;
  pthread_cond_init(&cond_queue_empty, NULL);
  pthread_cond_init(&cond_queue_full, NULL);
  pthread_mutex_init(&task_queue_cond_lock, NULL);
  /* create and join producer and consumer threads */
}
```



- Two conditions:
  - Queue is full: (task_available == 1) ← `cond_queue_full`
  - Queue is empty: (task_available == 0) ← `cond_queue_empty`
- A mutex for protecting accessing the queue (CS): `task_queue_cond_lock`

Producer-Consumer Using Condition Variables

```
  void *producer(void *producer_thread_data) {
    int inserted;
    while (!done()) {
       create_task();
       pthread_mutex_lock(&task_queue_cond_lock);

       while (task_available == 1)
1        pthread_cond_wait(&cond_queue_empty,
             &task_queue_cond_lock);

       insert_into_queue();
2      task_available = 1;             CS

3      pthread_cond_signal(&cond_queue_full);

       pthread_mutex_unlock(&task_queue_cond_lock);
    }
  }
```

Release mutex (unlock) when blocked/wait

Acquire mutex (lock) when awaken

Producer:

1. **Wait for queue to become empty, notified by consumer through cond_queue_empty**
2. **insert into queue**
3. **Signal consumer through cond_queue_full**

Producer-Consumer Using Condition Variables

```
   void *consumer(void *consumer_thread_data) {
     while (!done()) {
     pthread_mutex_lock(&task_queue_cond_lock);

     while (task_available == 0)
1      pthread_cond_wait(&cond_queue_full,
         &task_queue_cond_lock);

2      my_task = extract_from_queue();
       task_available = 0;

3      pthread_cond_signal(&cond_queue_empty);

       pthread_mutex_unlock(&task_queue_cond_lock);
       process_task(my_task);
     }
   }
```

Release mutex (unlock) when blocked/wait

Acquire mutex (lock) when awaken

Consumer:

1. **Wait for queue to become full, notified by producer through cond_queue_full**
2. **Extract task from queue**
3. **Signal producer through cond_queue_empty**

Thread and Synchronization Attributes

- Three major objects
  - pthread_t
  - pthread_mutex_t
  - pthread_cond_t
- Default attributes when being created/initialized
  - NULL
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
  - Once these properties are set, the attributes object can be passed to the method initializing the entity.
  - Enhances modularity, readability, and ease of modification.

Attributes Objects for Threads

- Initialize an attribute objects using **`pthread_attr_init`**
- Individual properties associated with the attributes object can be changed using the following functions:

`pthread_attr_setdetachstate`,
`pthread_attr_setguardsize_np`,
`pthread_attr_setstacksize`,
`pthread_attr_setinheritsched`,
`pthread_attr_setschedpolicy`, and
`pthread_attr_setschedparam`

Attributes Objects for Mutexes

- Initialize an attributes object using function: `pthread_mutexattr_init`.
- **pthread_mutexattr_settype_np** for setting the mutex type

  pthread_mutexattr_settype_np (pthread_mutexattr_t *attr,int type);
- Specific types:
  - PTHREAD_MUTEX_NORMAL_NP
  - PTHREAD_MUTEX_RECURSIVE_NP
  - PTHREAD_MUTEX_ERRORCHECK_NP

Attributes Objects for Condition Variable

- Initialize an attribute object using `pthread_condattr_init`

- **int pthread_condattr_setpshared(pthread_condattr_t *cattr, int pshared)** to specifies the scope of a condition variable to either process private (intraprocess) or system wide (interprocess) via pshared
  - PTHREAD_PROCESS_SHARED
  - PTHREAD_PROCESS_PRIVATE

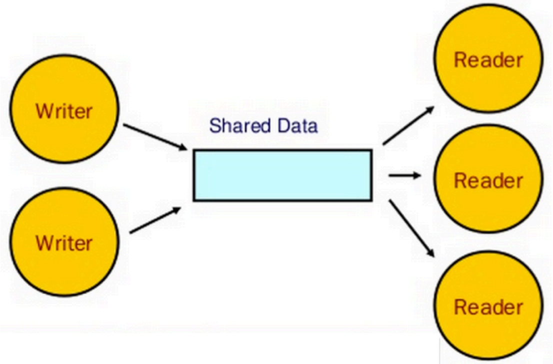
Composite Synchronization Constructs

- Pthread **Mutex** and **Condition Variables** are two basic sync operations.
- Higher level constructs can be built using basic constructs.
  - Read-write locks
  - Barriers

- Pthread has its corresponding implementation
  - pthread_rwlock_t
  - pthread_barrier_t

- We will discuss our own implementations

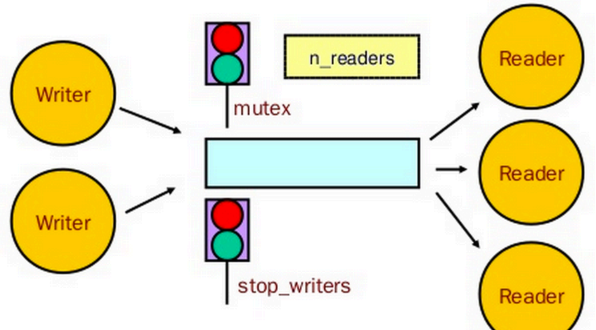
Read-Write Locks



- Concurrent access to data structure:
  - Read frequently but
  - Written infrequently
- Behavior:
  - Concurrent read: A read request is granted when there are other reads or no write (pending write request).
  - Exclusive write: A write request is granted only if there is no write or pending write request, or reads.
- Interfaces:
  - The rw lock data structure: **struct** **`mylib_rwlock_t`**
  - Read lock: **mylib_rwlock_rlock**
  - write lock: **mylib_rwlock_wlock**
  - Unlock: **mylib_rwlock_unlock**.

Read-Write Locks

- Two types of mutual exclusions
  - 0/1 mutex for protecting access to write
  - Counter mutex (semaphore) for counting read access
- Component sketch
  - a count of the number of readers,
  - 0/1 integer specifying whether a writer is present,
  - a condition variable readers_proceed that is signaled when readers can proceed,
  - a condition variable writer_proceed that is signaled when one of the writers can proceed,
  - a count pending_writers of pending writers, and
  - a pthread_mutex_t read_write_lock associated with the shared data structure



Read-Write Locks

```
typedef struct {
  int readers;
  int writer;
  pthread_cond_t readers_proceed;
  pthread_cond_t writer_proceed;
  int pending_writers;
  pthread_mutex_t read_write_lock;
} mylib_rwlock_t;


void mylib_rwlock_init (mylib_rwlock_t *l) {
  l->readers=0; l->writer=0; l->pending_writers=0;
  pthread_mutex_init(&(l->read_write_lock), NULL);
  pthread_cond_init(&(l->readers_proceed), NULL);
  pthread_cond_init(&(l->writer_proceed), NULL);
}
```

Read-Write Locks

```
  void mylib_rwlock_rlock(mylib_rwlock_t *l) {
    pthread_mutex_lock(&(l->read_write_lock));

    while ((l->pending_writers > 0) || (l->writer > 0))
1     pthread_cond_wait(&(l->readers_proceed),
      &(l->read_write_lock));

2   l->readers ++;

    pthread_mutex_unlock(&(l->read_write_lock));
  }
```

Reader lock:

1. **if there is a write or pending writers, perform condition wait,**
2. **else increment count of readers and grant read lock**

Read-Write Locks

```
  void mylib_rwlock_wlock(mylib_rwlock_t *l) {
    pthread_mutex_lock(&(l->read_write_lock));
    l->pending_writers ++;

    while ((l->writer > 0) || (l->readers > 0)) {
1      pthread_cond_wait(&(l->writer_proceed),
           &(l->read_write_lock));
    }

    l->pending_writers --;
2   l->writer ++;

    pthread_mutex_unlock(&(l->read_write_lock));
  }
```

Writer lock:

1. **If there are readers or writers, increment pending writers count and wait.**
2. **On being woken, decrement pending writers count and increment writer count**

Read-Write Locks

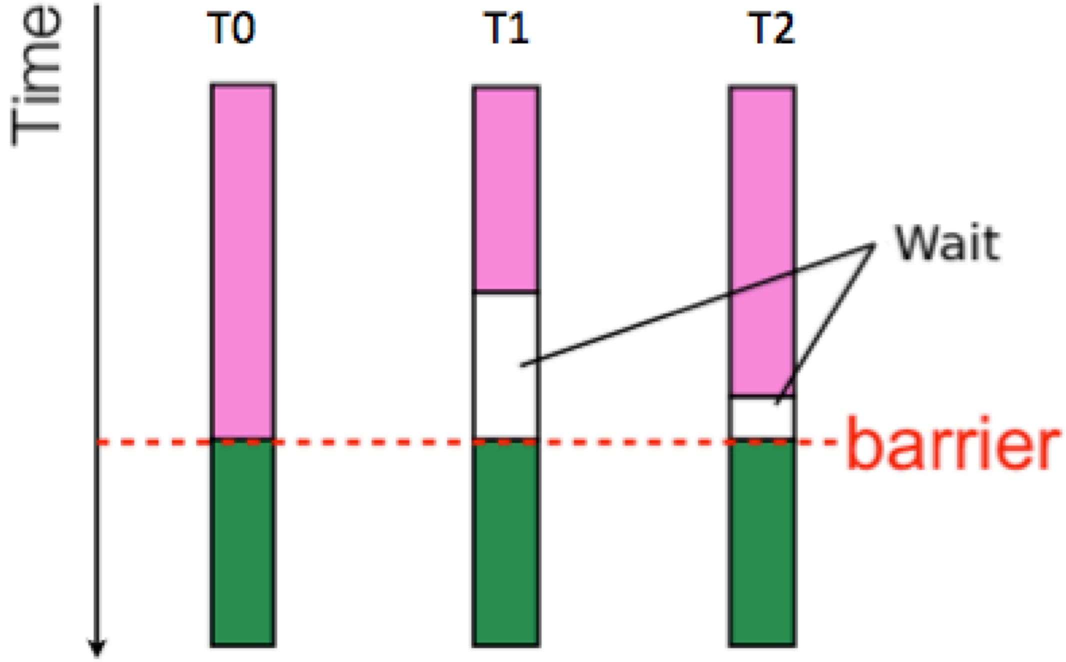
```
   void mylib_rwlock_unlock(mylib_rwlock_t *l) {
     pthread_mutex_lock(&(l->read_write_lock));
1    if (l->writer > 0) /* only writer */
       l->writer = 0;
2    else if (l->readers > 0) /* only reader */
       l->readers --;
     pthread_mutex_unlock(&(l->read_write_lock));

3    if ((l->readers == 0) && (l->pending_writers > 0))
       pthread_cond_signal(&(l->writer_proceed));
4    else if (l->readers > 0)
       pthread_cond_broadcast(&(l->readers_proceed));
   }
```

Reader/Writer unlock:

1. If there is a write lock then unlock
2. If there are read locks, decrement count of read locks.
3. If the read count becomes 0 and there is a pending writer, notify writer
4. Otherwise if there are pending readers, let them all go through

Barrier

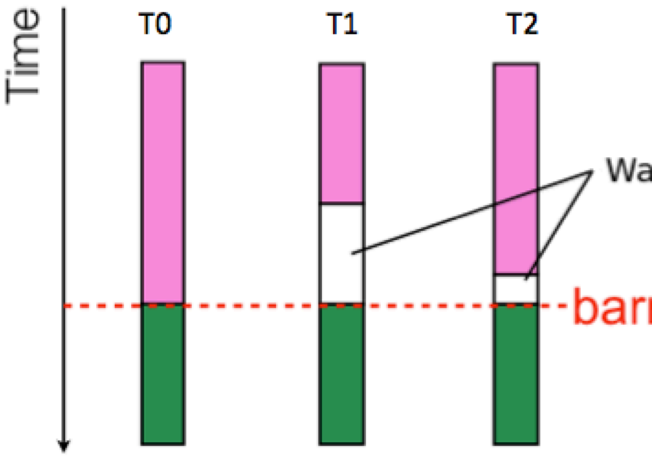
- A barrier holds one or multiple threads until all threads participating in the barrier have reached the barrier point



Barrier

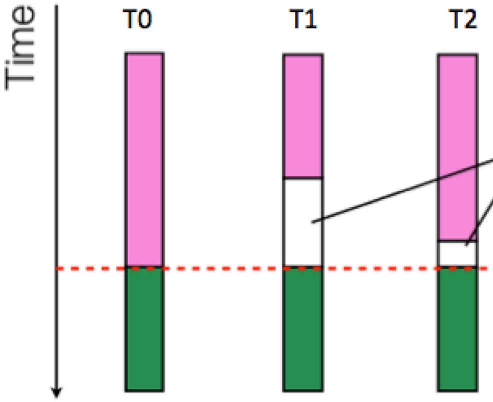
- Needs **a counter**, **a mutex** and **a condition variable**
  - The counter keeps track of the number of threads that have reached the barrier.
    - If the count is less than the total number of threads, the threads execute a condition wait.
  - The last thread entering (master) wakes up all the threads using a condition broadcast.

```
typedef struct {
  int count;
  pthread_mutex_t count_lock;
  pthread_cond_t ok_to_proceed;
} mylib_barrier_t;

void mylib_barrier_init(mylib_barrier_t *b) {
  b->count = 0;
  pthread_mutex_init(&(b->count_lock), NULL);
  pthread_cond_init(&(b->ok_to_proceed), NULL);
}
```



Barriers

```
 void mylib_barrier (mylib_barrier_t *b, int num_threads) {
   pthread_mutex_lock(&(b->count_lock));

1  b->count ++;
   if (b->count == num_threads) {
2     b->count = 0;
      pthread_cond_broadcast(&(b->ok_to_proceed));
   } else
3     while (pthread_cond_wait(&(b->ok_to_proceed),
        &(b->count_lock)) != 0);

   pthread_mutex_unlock(&(b->count_lock));
 }
```

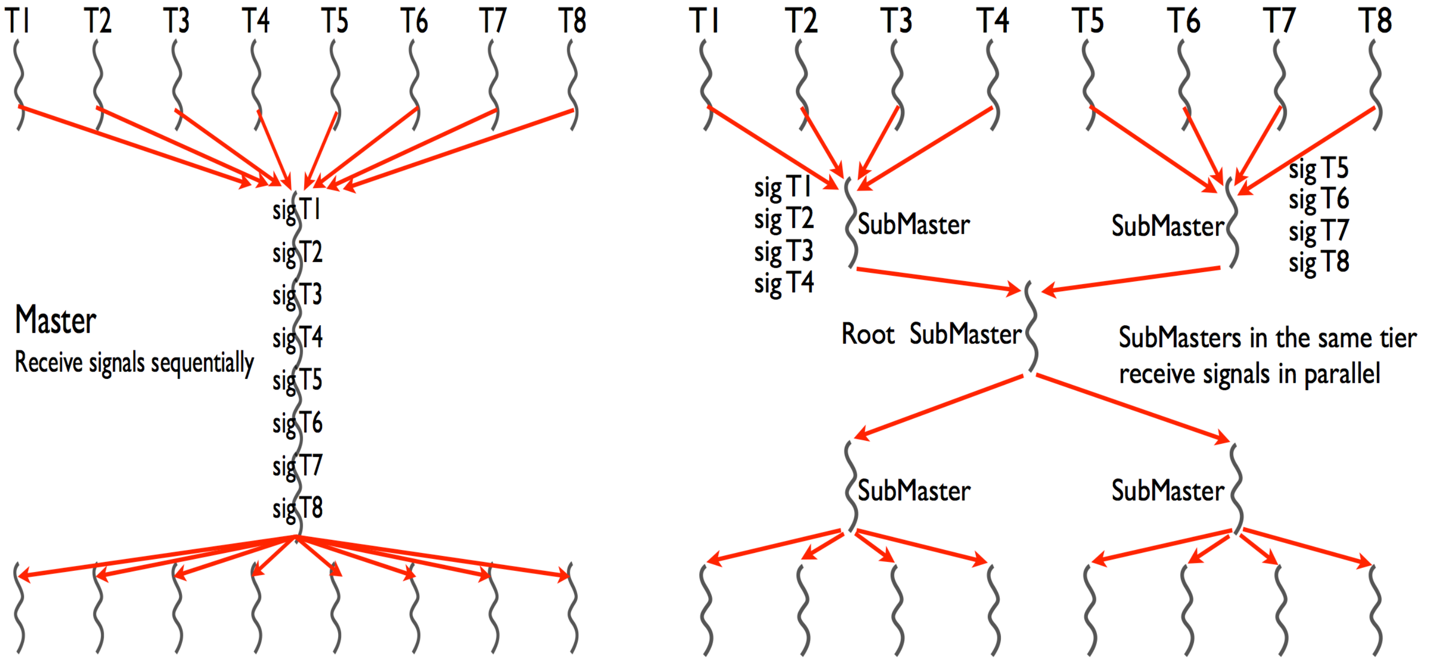


Barrier

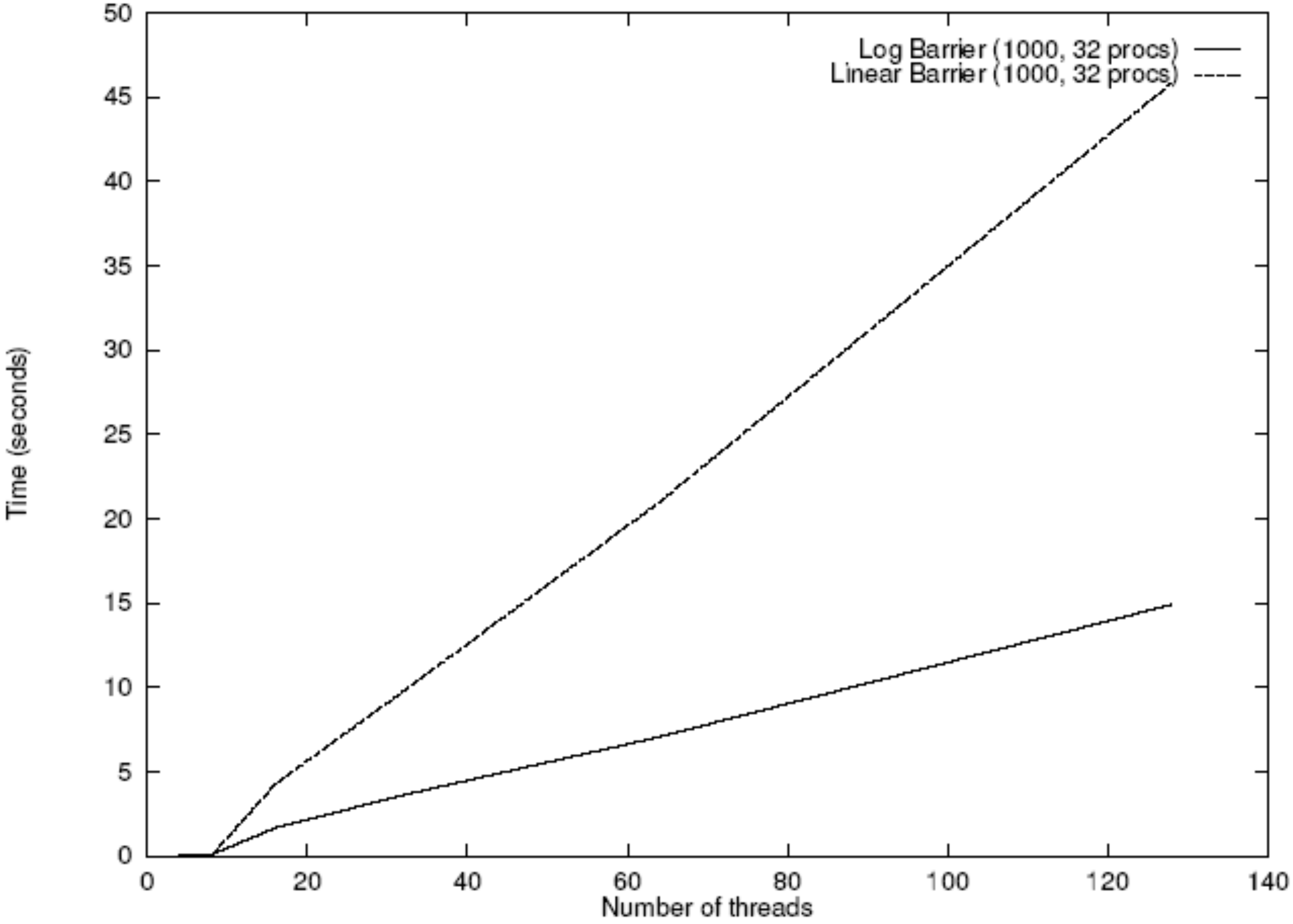
1. Each thread increments the counter and check whether all reach
2. The thread (master) who detect that all reaches signal others to proceed
3. If not all reach, the thread waits

Flat/Linear vs Tree/Log Barrier

- Linear/Flat barrier.
  - O(n) for n thread
  - A single master to collect information of all threads and notify them to continue
- Tree/Log barrier
  - Organize threads in a tree logically
  - Multiple submaster to collect and notify
  - Runtime grows as O(log p).



Barrier



- Execution time of 1000 sequential and logarithmic barriers as a function of number of threads on a 32 processor SGI Origin 2000.

References

- Adapted from slides “Programming Shared Address Space Platforms” by Ananth Grama. Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell.
- “Pthreads Programming: A POSIX Standard for Better Multiprocessing.” O'Reilly Media, 1996.
- Chapter 7. “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003

- Other pthread topics
  - int pthread_key_create(pthread_key_t *key, void (*destroy)(void *))
  - int pthread_setspecific(pthread_key_t key, const void *value)
  - void *pthread_getspecific(pthread_key_t key)

Alleviating Locking Overhead (Example)

```
/* Finding k matches in a list */
void *find_entries(void *start_pointer) {
  /* This is the thread function */
  struct database_record *next_record;
  int count;
  current_pointer = start_pointer;
  do {
     next_record = find_next_entry(current_pointer);
     count = output_record(next_record);
  } while (count < requested_number_of_records);
}
int output_record(struct database_record *record_ptr) {
  int count;
  pthread_mutex_lock(&output_count_lock);
  output_count ++;
  count = output_count;
  pthread_mutex_unlock(&output_count_lock);
  if (count <= requested_number_of_records)
     print_record(record_ptr);
  return (count);
}
```

Alleviating Locking Overhead (Example)

```
/* rewritten output_record function */
int output_record(struct database_record
   *record_ptr) {
  int count;
  int lock_status;
  lock_status=pthread_mutex_trylock(&output_count_lock);
  if (lock_status == EBUSY) {
     insert_into_local_list(record_ptr);
     return(0);
  } else {
     count = output_count;
     output_count += number_on_local_list + 1;
     pthread_mutex_unlock(&output_count_lock);
     print_records(record_ptr, local_list,
       requested_number_of_records - count);
     return(count + number_on_local_list + 1);
  }
}
```