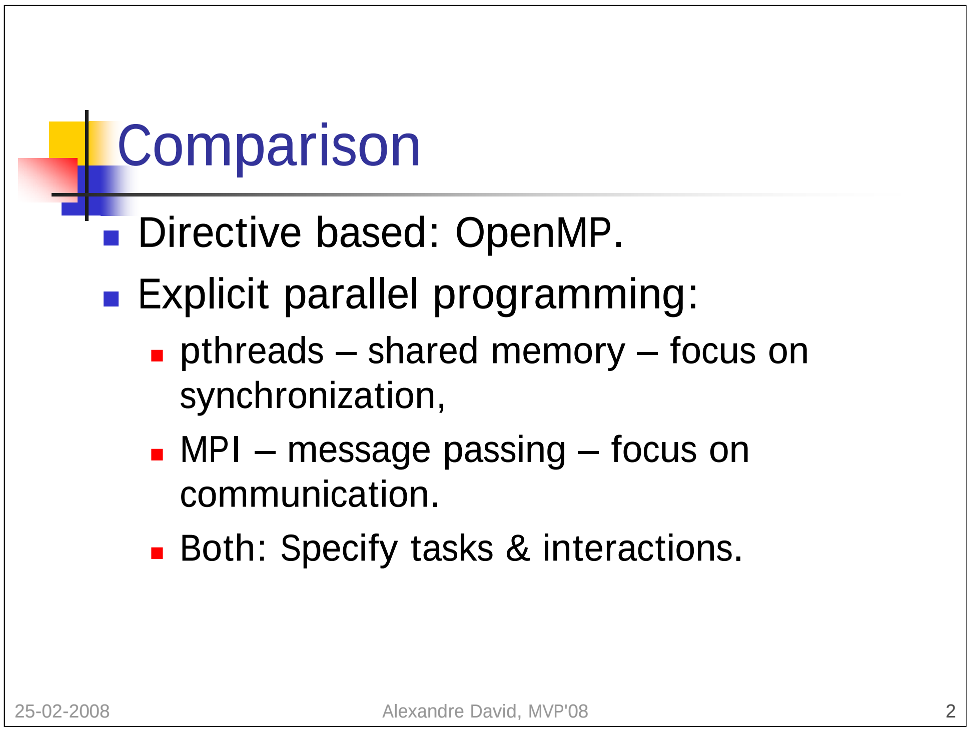
Programming Shared Address Space Platforms (Chapter 7)

Alexandre David

1.2.05

This is about pthreads.



Comparison

- Directive based: OpenMP.
- Explicit parallel programming:
  - pthreads – shared memory – focus on synchronization,
  - MPI – message passing – focus on communication.
  - Both: Specify tasks & interactions.

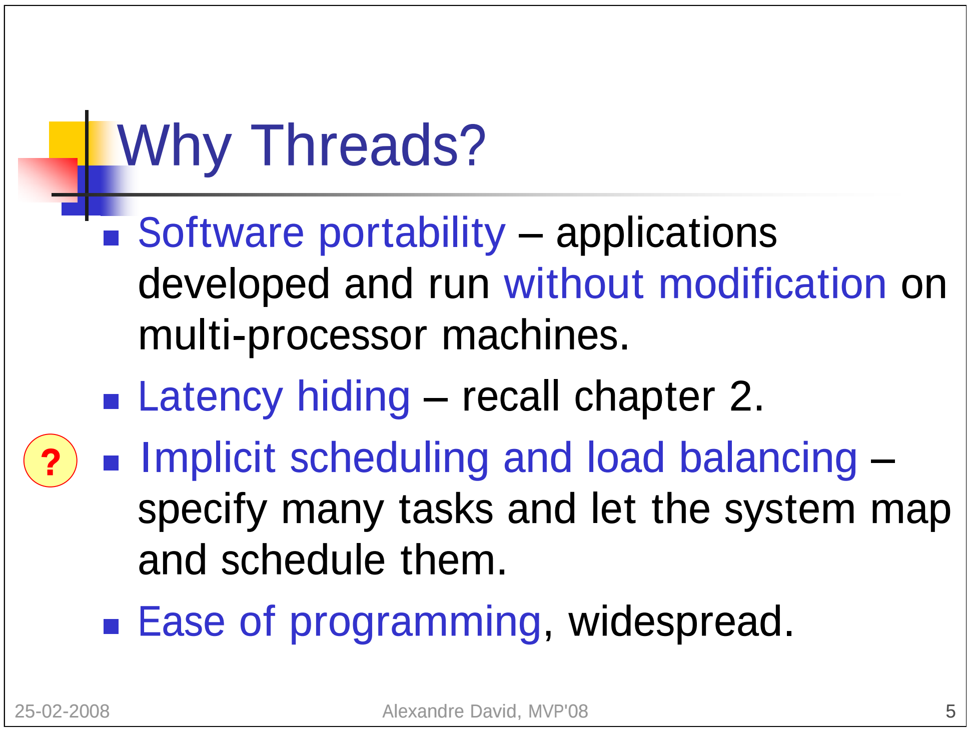
Programming paradigms for shared address space machines focus on constructs for expressing concurrency and synchronization. Communication in shared memory programming is implicitly specified. We focus on minimizing data-sharing overheads (for MPI it’s communication overheads).

Programming Models

- Concurrency supported by:
  - Processes – private data unless otherwise specified.
  - Threads – shared memory, lightweight.
  - Directive based programming – concurrency specified as high level compiler directive, OpenMP.
- See OS course.

Threads Basics

- All memory is globally accessible.
- But the stack is considered local.
  - In practice both local (private) and global (shared) memory.
  - Recall that memory is physically distributed and local accesses are faster.
  - Applicable to SMP/multi-core machines.

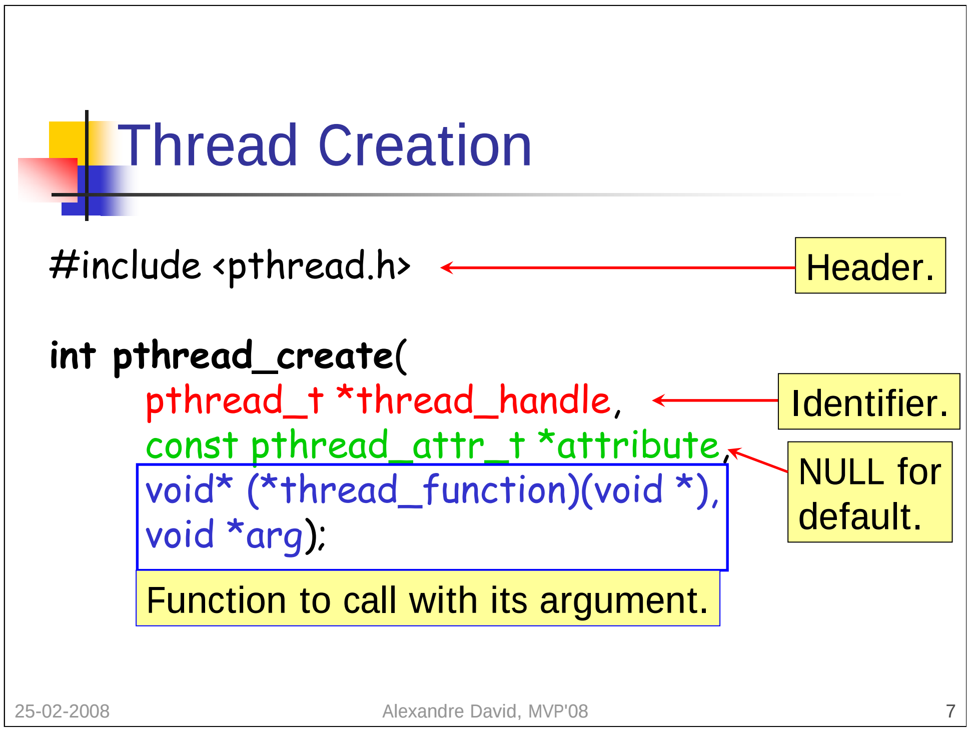


Why Threads?

- Software portability – applications developed and run without modification on multi-processor machines.
- Latency hiding – recall chapter 2.
- ? Implicit scheduling and load balancing – specify many tasks and let the system map and schedule them.
- Ease of programming, widespread.

The POSIX Thread API

- It is a standard API (like MPI).
  - Supported by most vendors.
- General concepts applicable to other thread APIs (java threads, NT threads, etc).
- Low level functions, API is missing high level constructs, e.g., no collective communication like in MPI.



Thread Creation

```
#include <pthread.h>

int pthread_create(
        pthread_t *thread_handle,
        const pthread_attr_t *attribute,
        void* (*thread_function)(void *),
        void *arg);
```

Header.
Identifier.
NULL for default.
Function to call with its argument.

Invokes *thread_function* as a thread.

Notes:

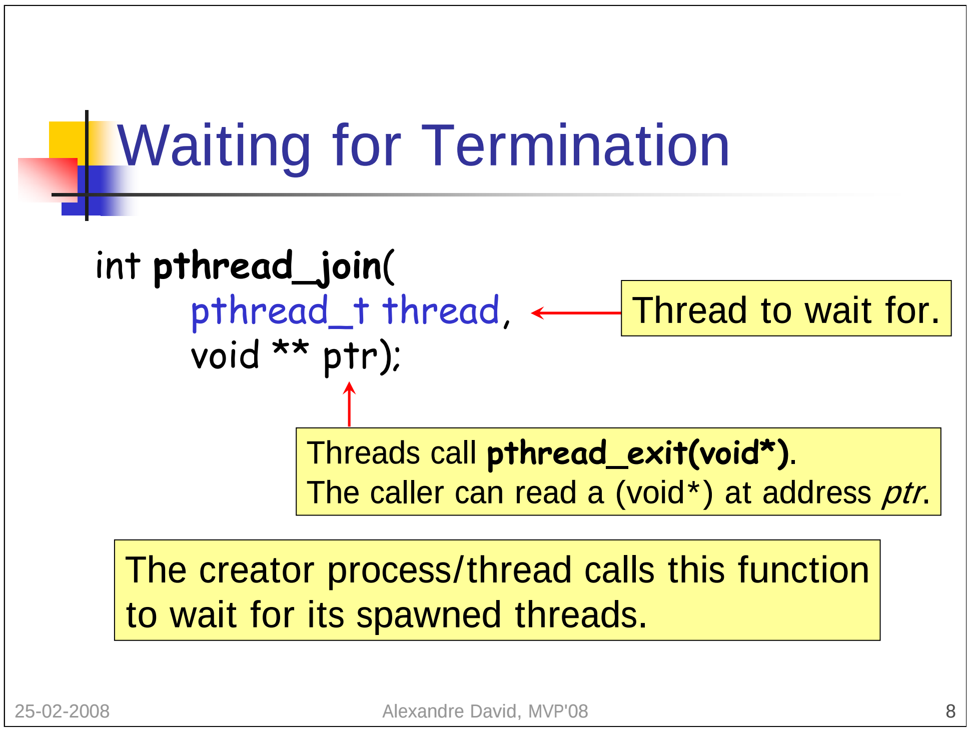
•The identifier *thread_handle* is written before the function returns.

•The function returns in the main thread, the function thread_function runs in parallel in another thread.

•On uni-processor machines the thread may preempt its creator thread.

•There is a returned result (success or not).

•Beware of race conditions: Make sure to initialize everything before creating the thread (and not after).



Waiting for Termination

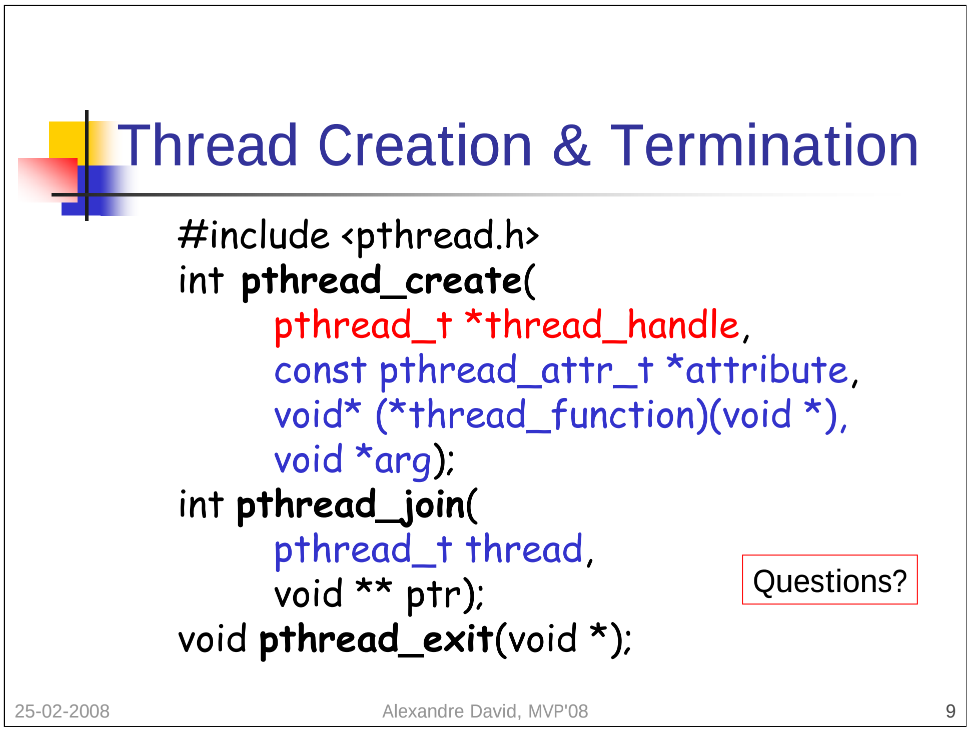
```
int pthread_join(
        pthread_t thread,
        void ** ptr);
```

Thread to wait for.

Threads call **pthread_exit(void*)**.
The caller can read a (void*) at address *ptr*.

The creator process/thread calls this function to wait for its spawned threads.

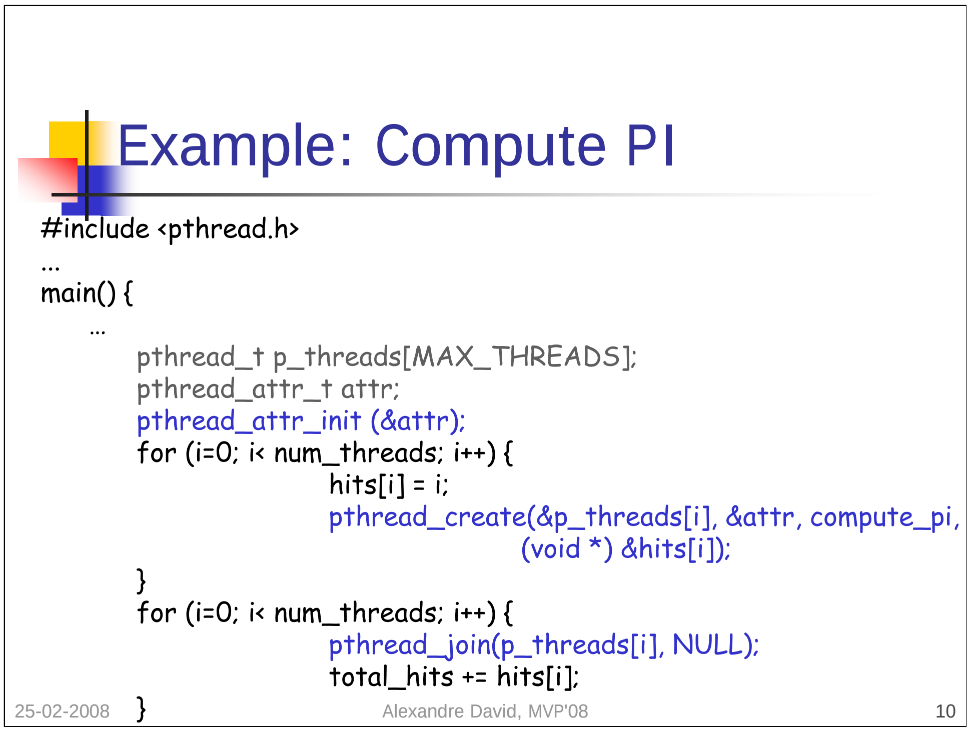
And returns success (0) or an error code.



Thread Creation & Termination

```
#include <pthread.h>
int pthread_create(
        pthread_t *thread_handle,
        const pthread_attr_t *attribute,
        void* (*thread_function)(void *),
        void *arg);
int pthread_join(
        pthread_t thread,
        void ** ptr);
void pthread_exit(void *);
```

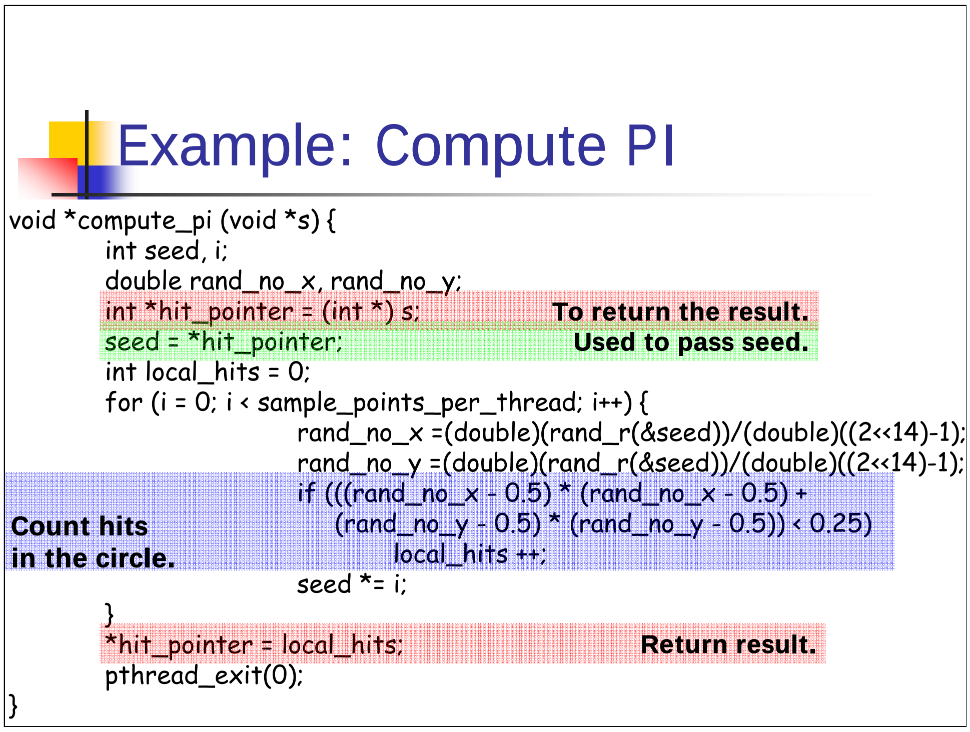
Questions?



Example: Compute PI

```
#include <pthread.h>
…
main() {
     …
          pthread_t p_threads[MAX_THREADS];
          pthread_attr_t attr;
          pthread_attr_init (&attr);
          for (i=0; i< num_threads; i++) {
                              hits[i] = i;
                              pthread_create(&p_threads[i], &attr, compute_pi,
                                        (void *) &hits[i]);
          }
          for (i=0; i< num_threads; i++) {
                              pthread_join(p_threads[i], NULL);
                              total_hits += hits[i];
          }
```

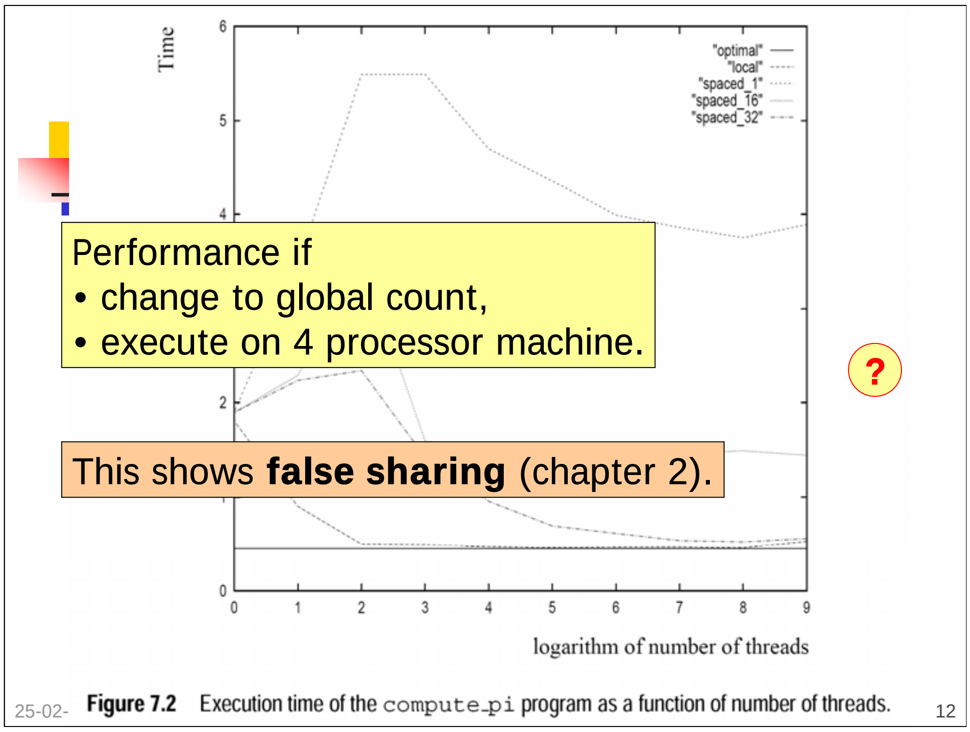
This is a lousy computation of pi. Based on area ratios. Take a square 1x1 and put a circle inside. Square area = 1, circle area = $\pi r^2 = \pi/4$ ($r = ½$). Choose many points randomly and the ratio hits/total will converge towards $\pi/4$.



Example: Compute PI

```
void *compute_pi (void *s) {
        int seed, i;
        double rand_no_x, rand_no_y;
        int *hit_pointer = (int *) s;              To return the result.
        seed = *hit_pointer;                       Used to pass seed.
        int local_hits = 0;
        for (i = 0; i < sample_points_per_thread; i++) {
                rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
                rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
                if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
Count hits         (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
in the circle.         local_hits ++;
                seed *= i;
        }
        *hit_pointer = local_hits;                 Return result.
        pthread_exit(0);
}
```

Call to **rand_r**, worse than drand48 or rand, because we need a **reentrant** function.

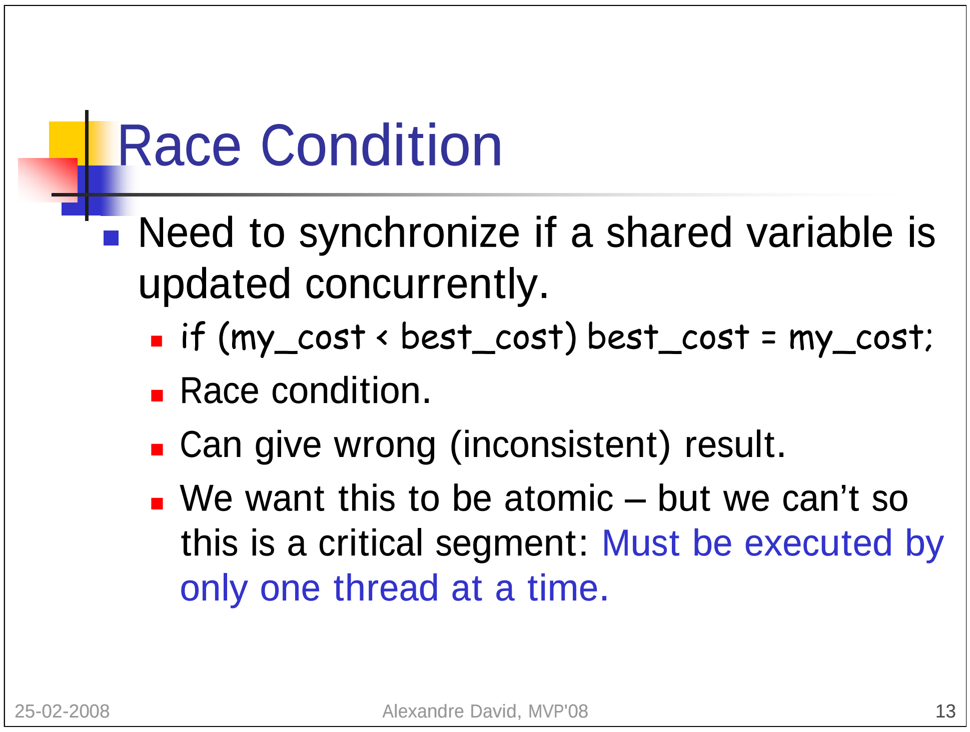


Performance if
- change to global count,
- execute on 4 processor machine.

This shows **false sharing** (chapter 2).

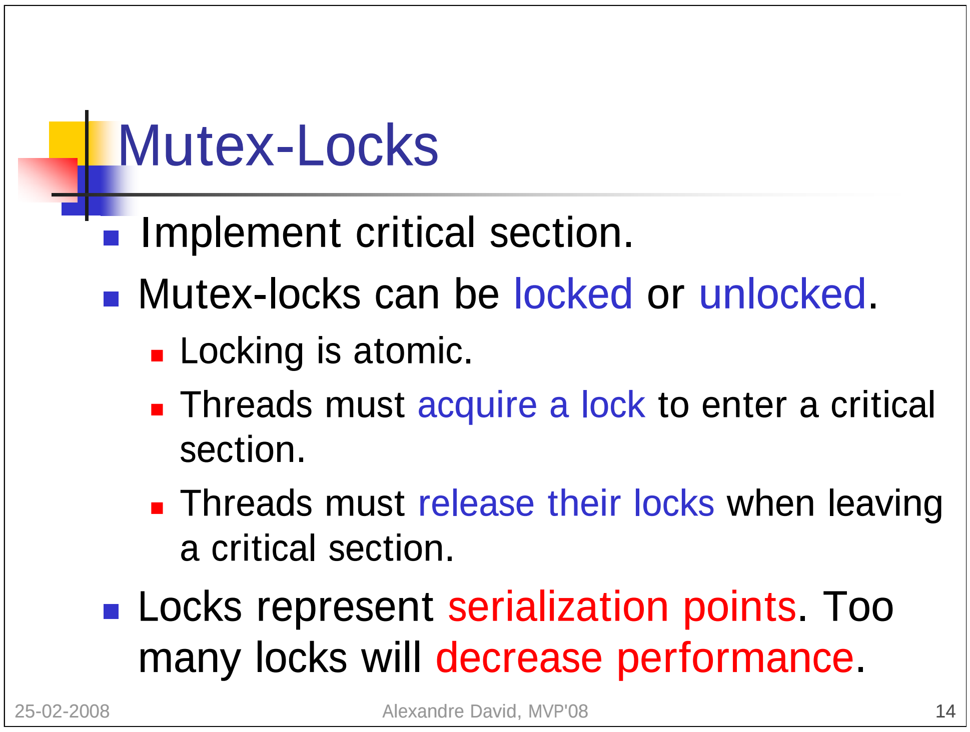
Figure 7.2 Execution time of the `compute_pi` program as a function of number of threads.

Speedup of 3.91, efficiency = 0.98. Note: The threads do not synchronize with each other.


Race Condition

- Need to synchronize if a shared variable is updated concurrently.
  - `if (my_cost < best_cost) best_cost = my_cost;`
  - Race condition.
  - Can give wrong (inconsistent) result.
  - We want this to be atomic – but we can't so this is a critical segment: Must be executed by only one thread at a time.

Race condition: The result depends on the order of the different statements in parallel, i.e., the interleaving. Inconsistent result: It does not correspond to any serialization of the threads (considering the test-and-update atomic).



Mutex-Locks

- Implement critical section.
- Mutex-locks can be locked or unlocked.
  - Locking is atomic.
  - Threads must acquire a lock to enter a critical section.
  - Threads must release their locks when leaving a critical section.
- Locks represent serialization points. Too many locks will decrease performance.

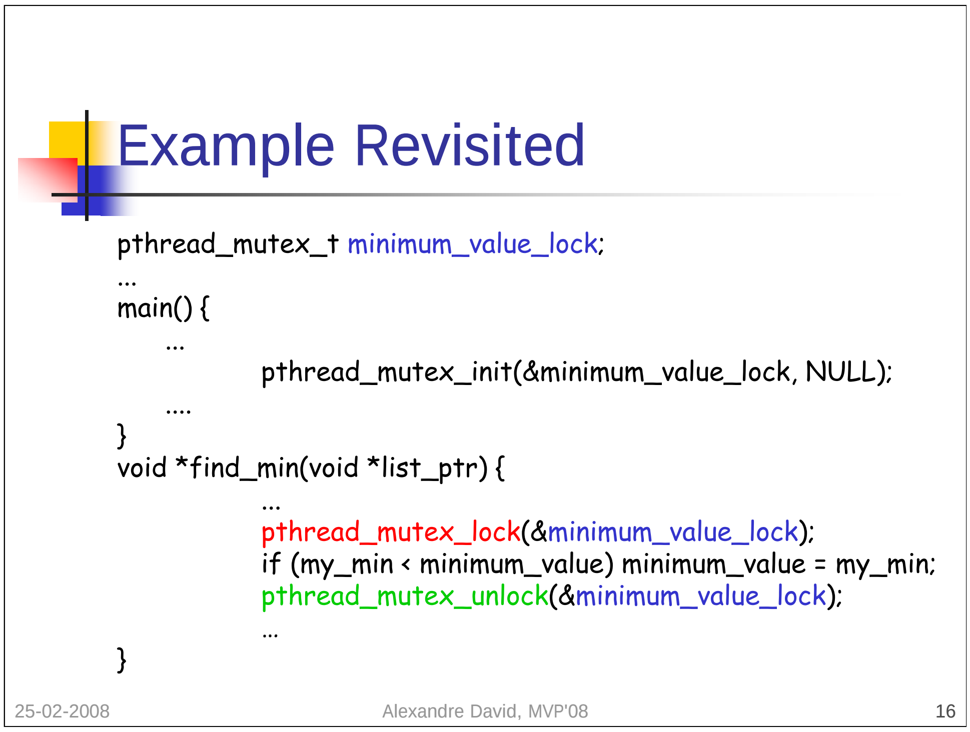
25-02-2008 Alexandre David, MVP'08 14

In the book “critical segment” but usually called “critical section”. The call to “lock-a-thread” is blocking and returns only when the lock is acquired. Of course all locks must initialized to unlocked when starting programs.

Be also careful on the granularity of what you lock. Locking big portions of code is bad since you are killing parallelism for the code you are locking.

Mutex-Lock

```
int pthread_mutex_init(
        pthread_mutex_t *mutex_lock,
        const pthread_mutextattr_t *lock_attr);

int pthread_mutex_lock(
        pthread_mutex_t *mutex_lock);

int pthread_mutex_unlock(
        pthread_mutex_t *mutex_lock);
```



Example Revisited

```
pthread_mutex_t minimum_value_lock;
...
main() {
      ...
                  pthread_mutex_init(&minimum_value_lock, NULL);
      ....
}
void *find_min(void *list_ptr) {
                  ...
                  pthread_mutex_lock(&minimum_value_lock);
                  if (my_min < minimum_value) minimum_value = my_min;
                  pthread_mutex_unlock(&minimum_value_lock);
                  ...
}
```

25-02-2008 Alexandre David, MVP'08 16

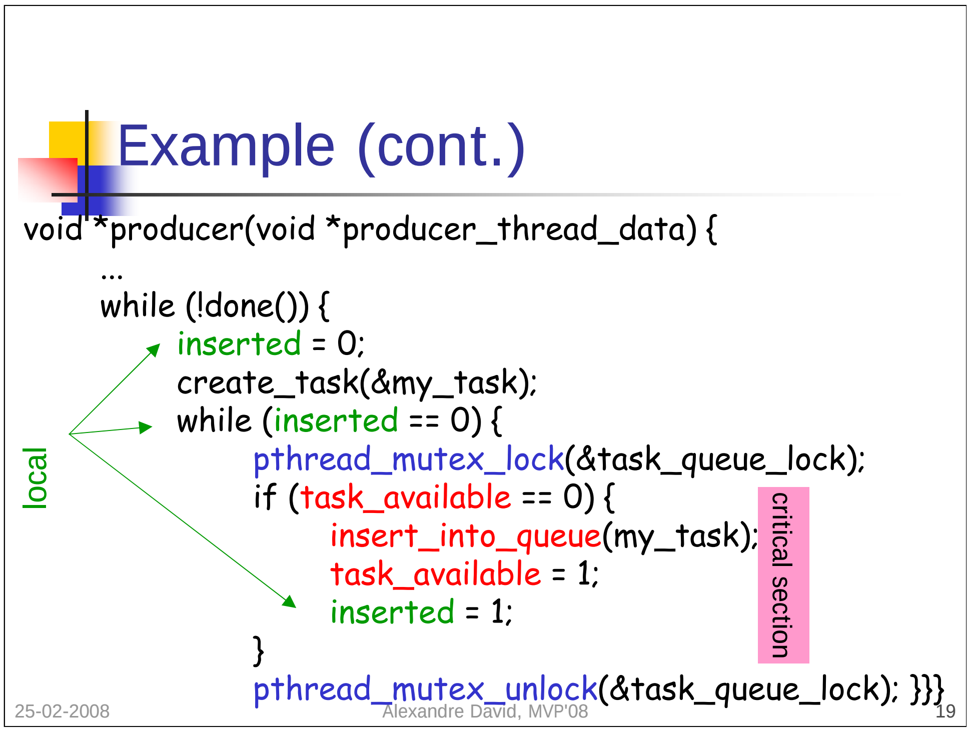
Careful with the use of mutex locks. Don't use one mutex lock for all your locks if they are independent. Use one different lock for different kinds of code segments that are not mutually exclusive – it may still be the case that you have 2 portions of code accessing the same data, in which case you need to use the same lock.

Producer-Consumer Example

- Shared buffer containing one task.
  - No overwrite until cleared.
  - No read until written.
  - Pick one task at a time.
- Note: Better with semaphores in this case.

Example

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

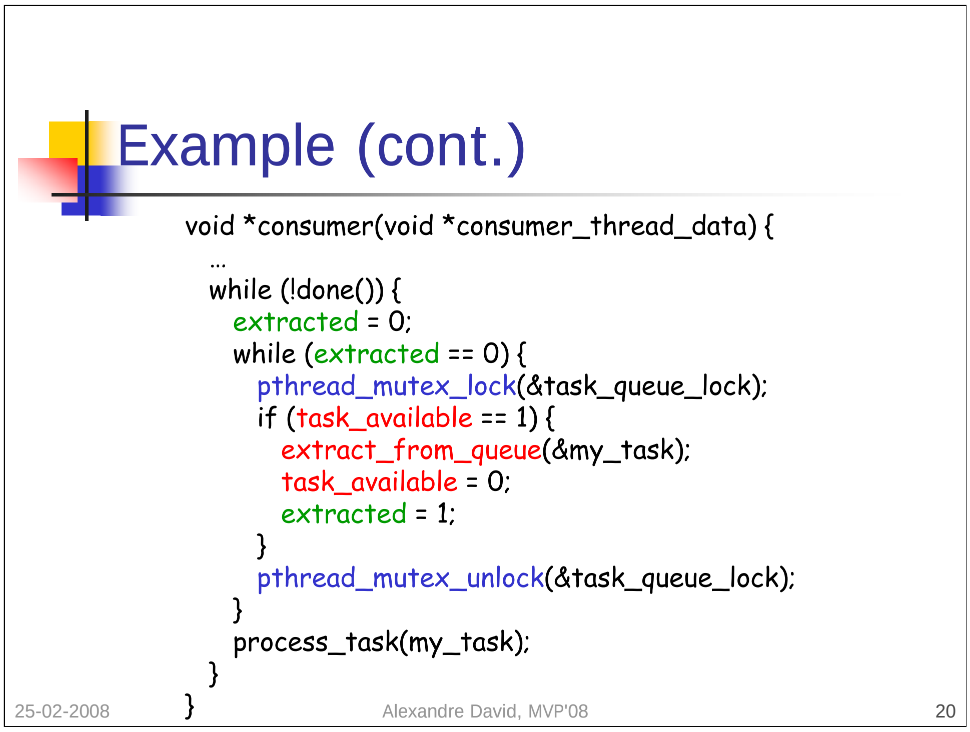


Example (cont.)

```
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); }}}
```

local
critical section

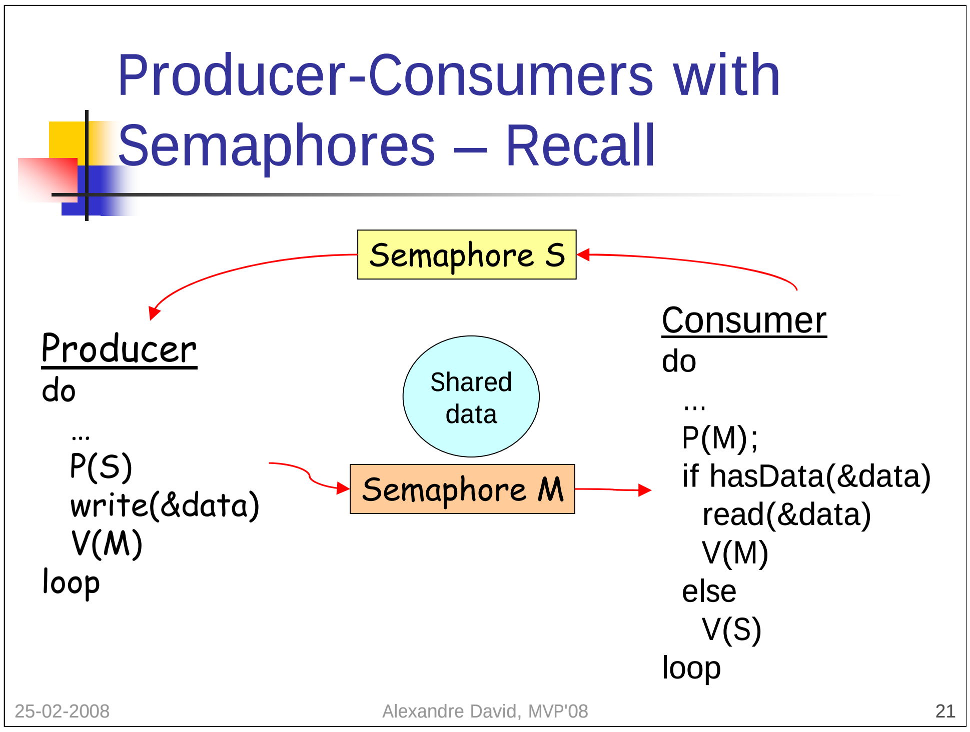
Why is this a bad example?



Example (cont.)

```
void *consumer(void *consumer_thread_data) {
  …
  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);
  }
}
```

Do it better with semaphores.



Producer-Consumers with Semaphores – Recall

Semaphore S

Shared data

Semaphore M

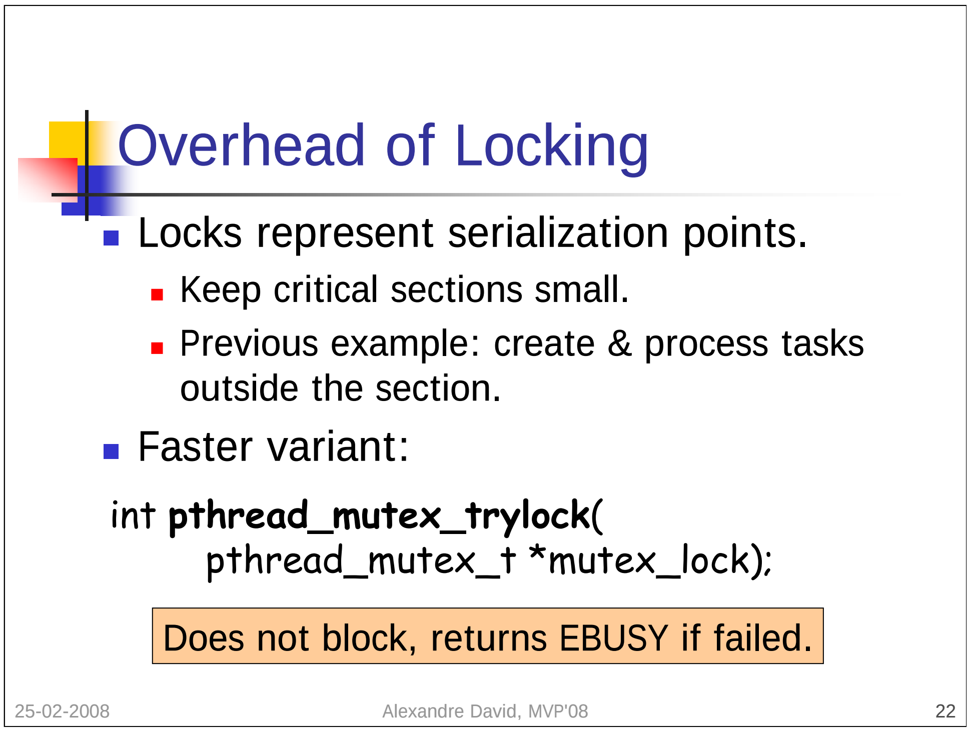
Producer

```
do
  …
  P(S)
  write(&data)
  V(M)
loop
```

Consumer

```
do
  …
  P(M);
  if hasData(&data)
    read(&data)
    V(M)
  else
    V(S)
loop
```

25-02-2008 Alexandre David, MVP'08 21



Overhead of Locking

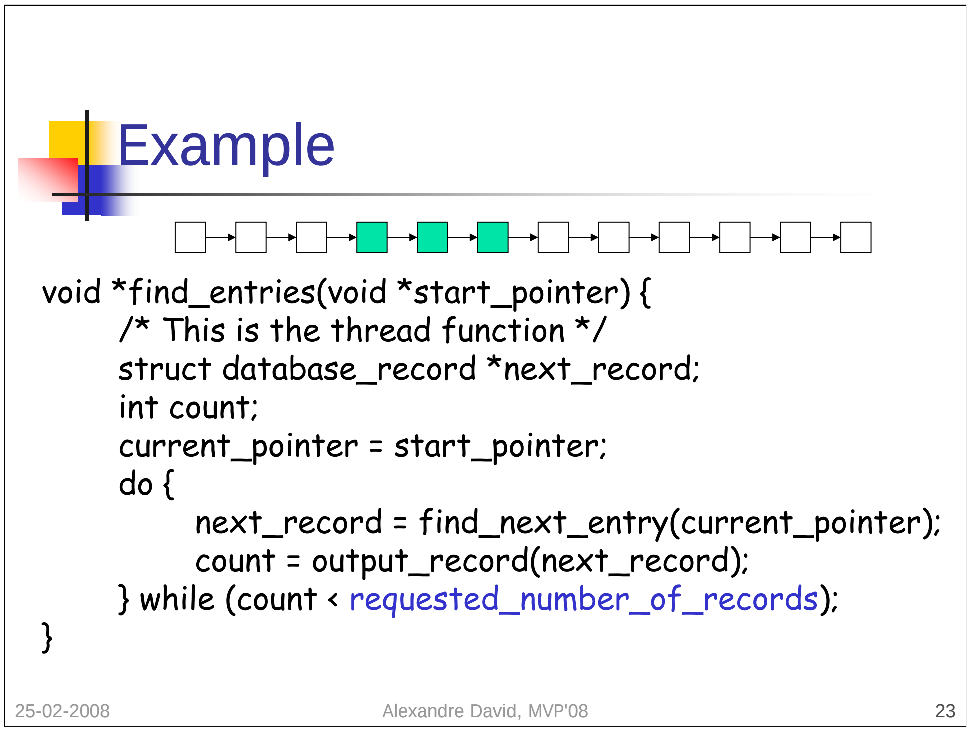
- Locks represent serialization points.
  - Keep critical sections small.
  - Previous example: create & process tasks outside the section.
- Faster variant:

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

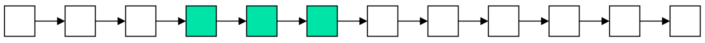
Does not block, returns EBUSY if failed.

25-02-2008 Alexandre David, MVP'08 22

This variant is faster because there is no management of waiting queues and waking up threads that are blocked.



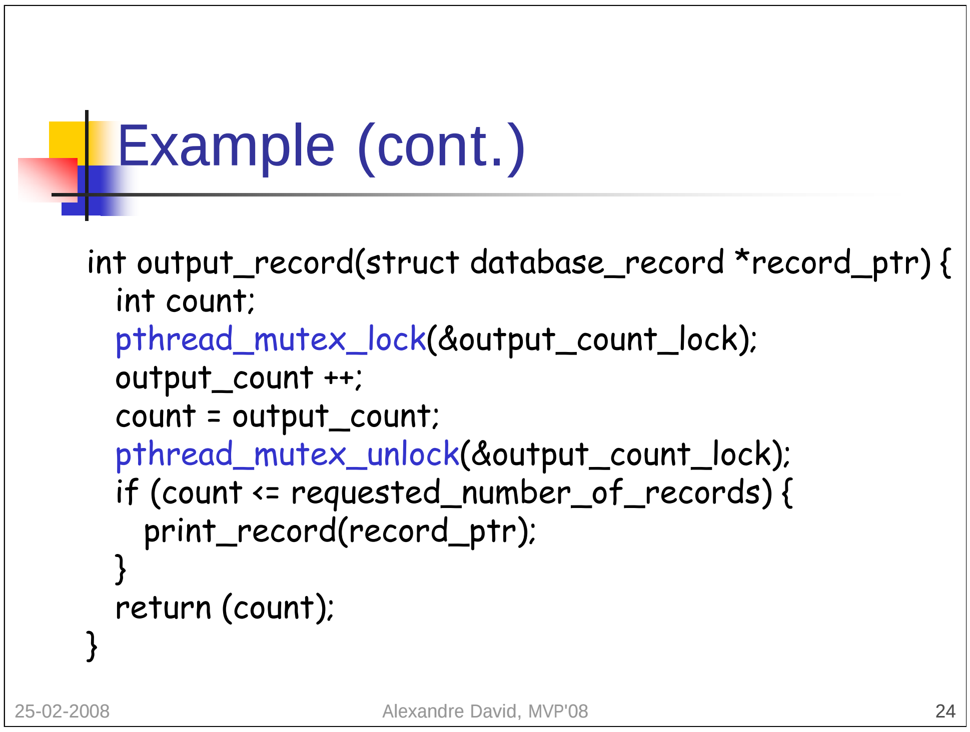
Example



```
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);
}
```

25-02-2008 Alexandre David, MVP'08 23

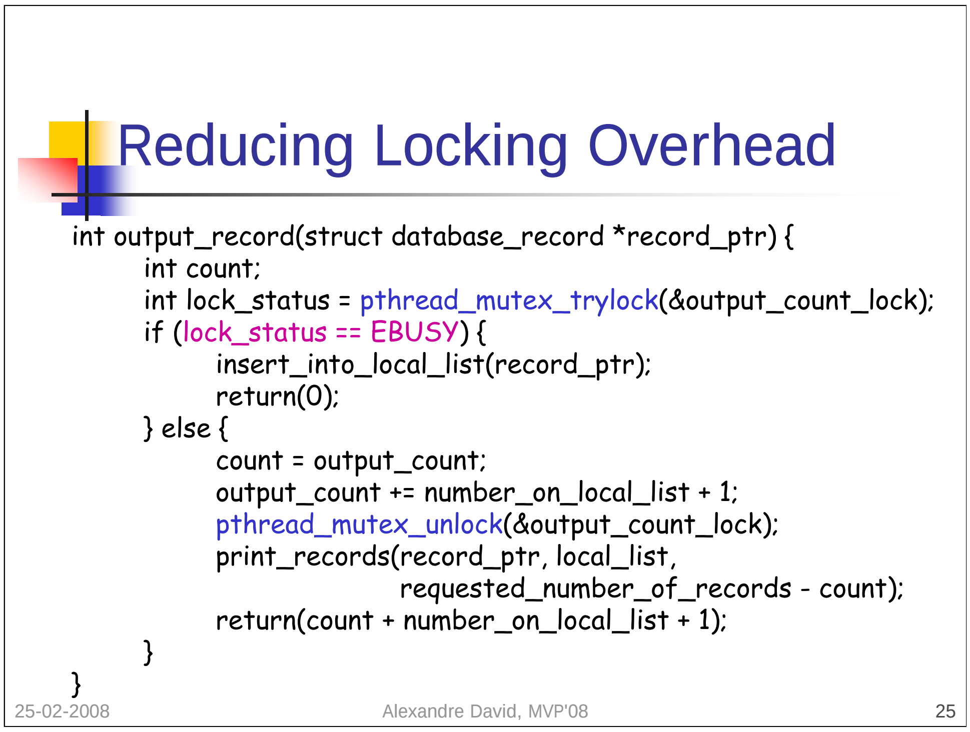
Find k matches in a list. The example is not fully correct.



Example (cont.)

```
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);
}
```

Looks ok but if the times of the previous loop and this section are comparable then we have a terrible overhead.



Reducing Locking Overhead

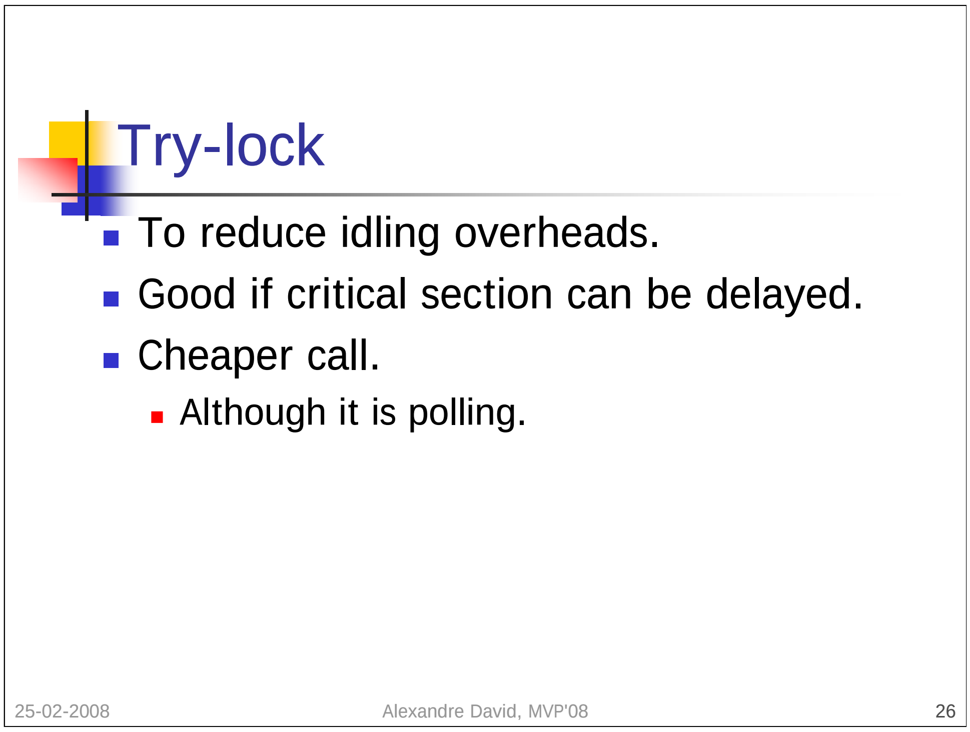
```
int output_record(struct database_record *record_ptr) {
        int count;
        int lock_status = pthread_mutex_trylock(&output_count_lock);
        if (lock_status == EBUSY) {
                insert_into_local_list(record_ptr);
                return(0);
        } else {
                count = 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);
        }
}
```

25-02-2008 Alexandre David, MVP'08 25

Example is not completely correct in fact (more entries searched than asked).

Better performance because the locking call is much faster and the number of locked operations is reduced.

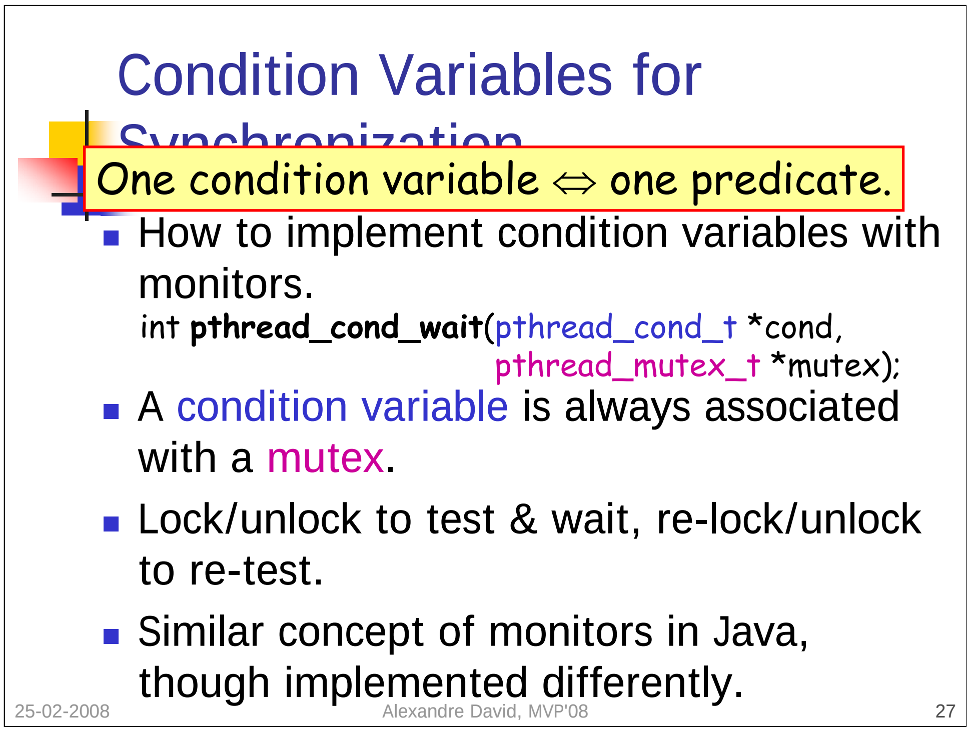
Very important: The lock must be released, and only when it was acquired.



Try-lock

- To reduce idling overheads.
- Good if critical section can be delayed.
- Cheaper call.
  - Although it is polling.

25-02-2008 Alexandre David, MVP'08 26



Condition Variables for Synchronization

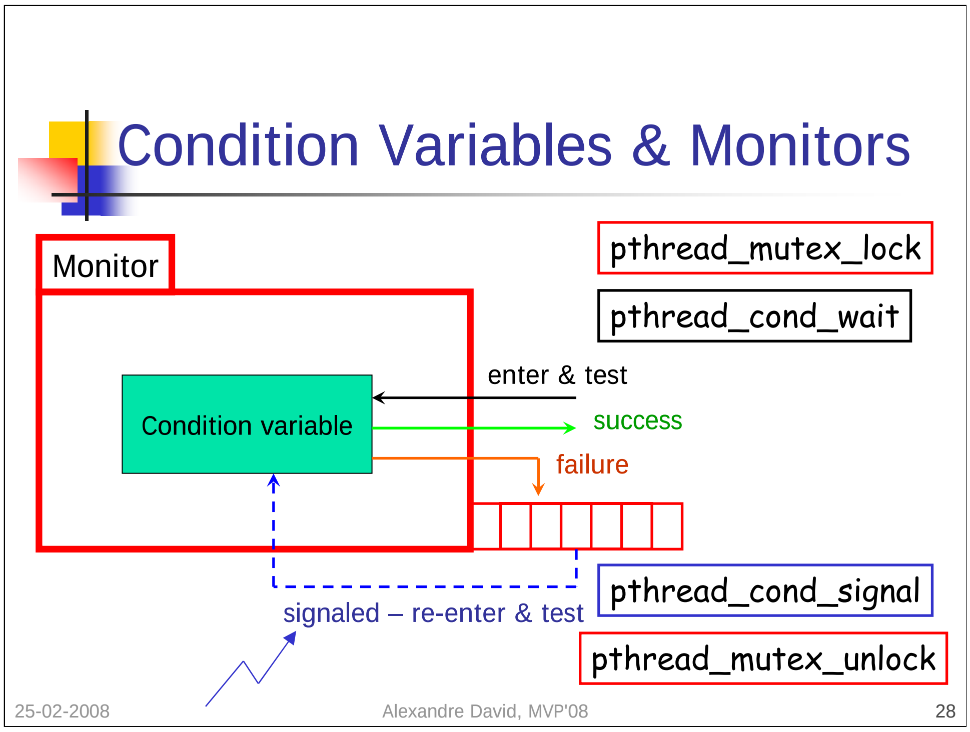
One condition variable ⇔ one predicate.

- How to implement condition variables with monitors.

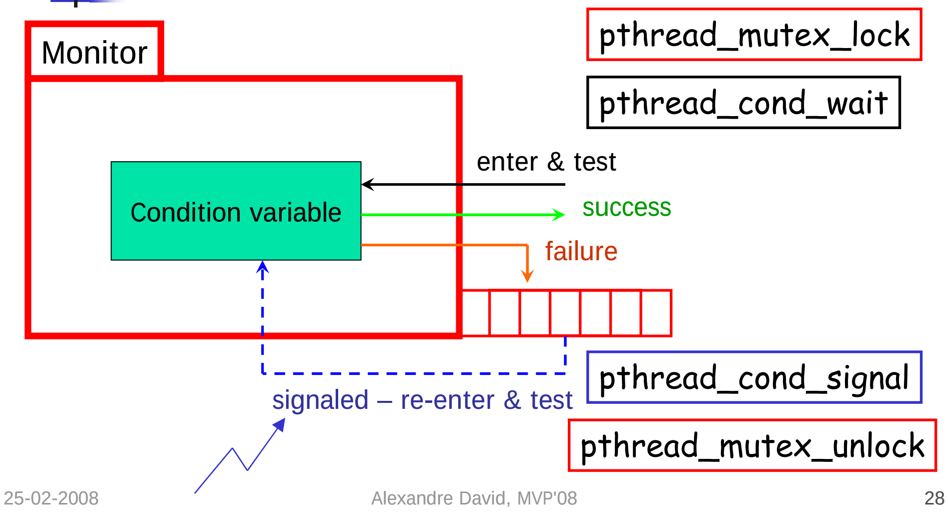
```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

- A condition variable is always associated with a mutex.
- Lock/unlock to test & wait, re-lock/unlock to re-test.
- Similar concept of monitors in Java, though implemented differently.

Associate one condition to one predicate only.



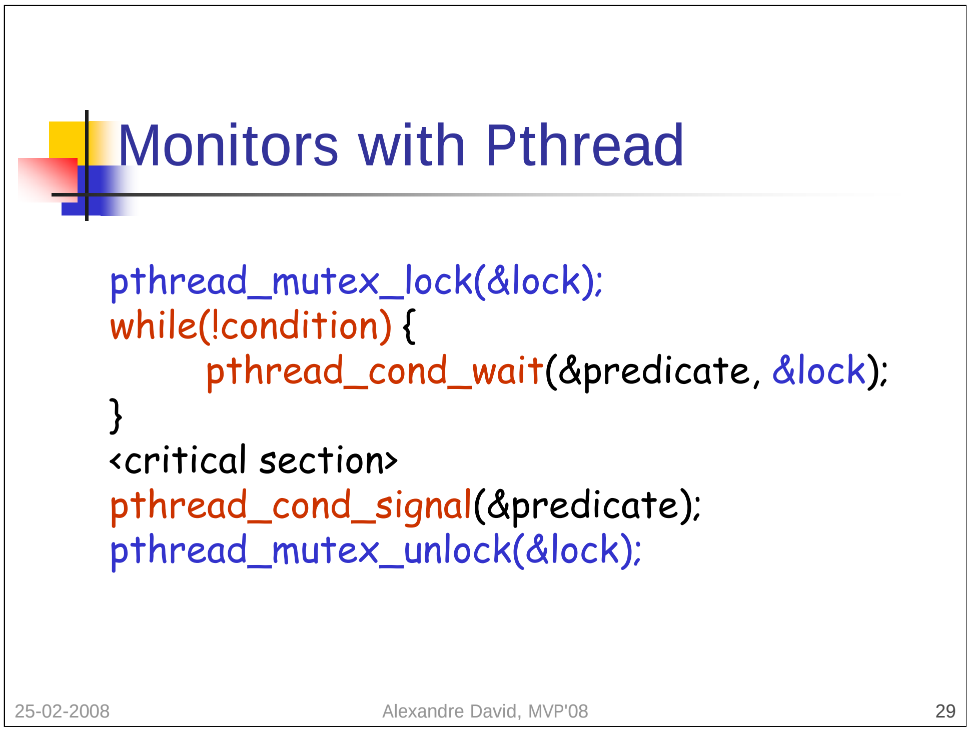
Condition Variables & Monitors



25-02-2008 Alexandre David, MVP'08 28

In java:

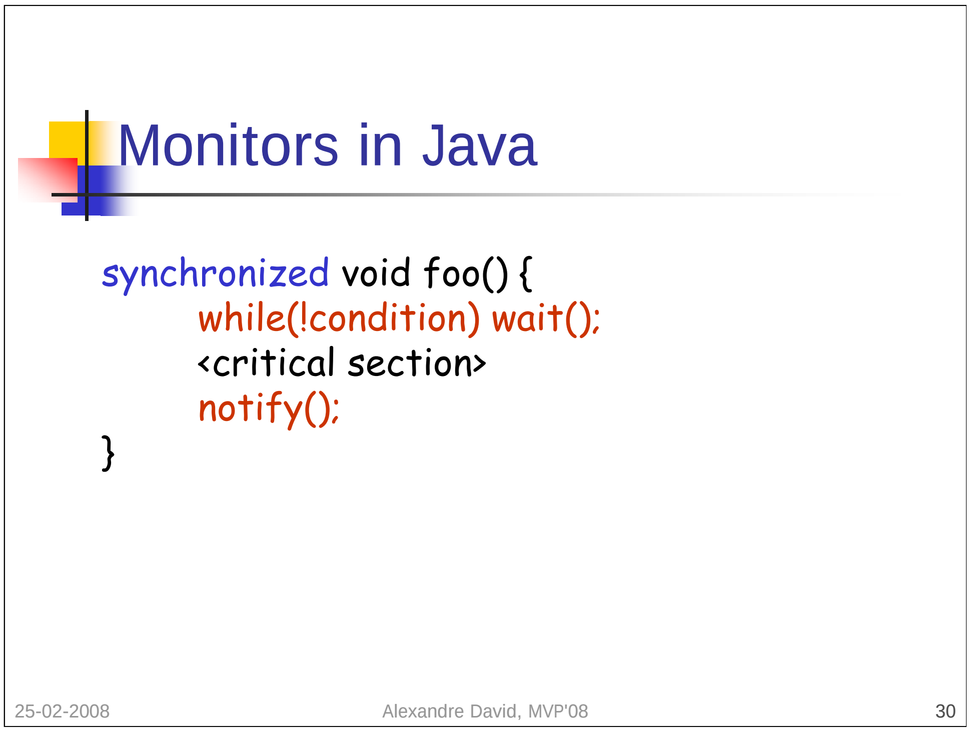
```
synchronized void foo() {
  if (!condition) wait();
  …
  notify();
}
```



Monitors with Pthread

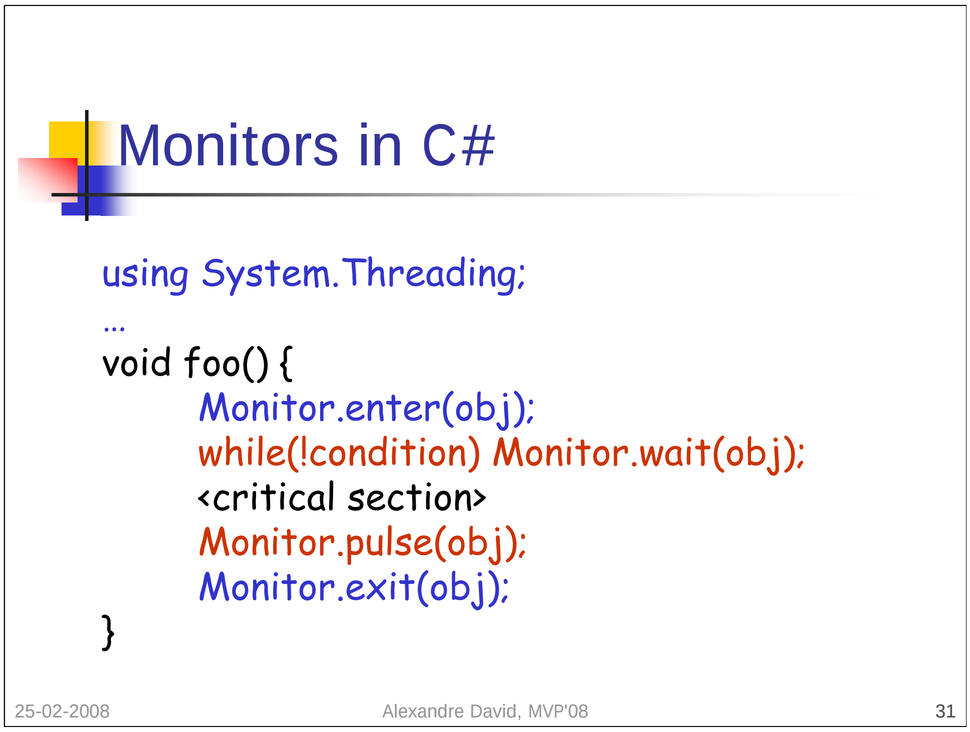
```
pthread_mutex_lock(&lock);
while(!condition) {
        pthread_cond_wait(&predicate, &lock);
}
<critical section>
pthread_cond_signal(&predicate);
pthread_mutex_unlock(&lock);
```

Why do we have a loop on the condition variable?



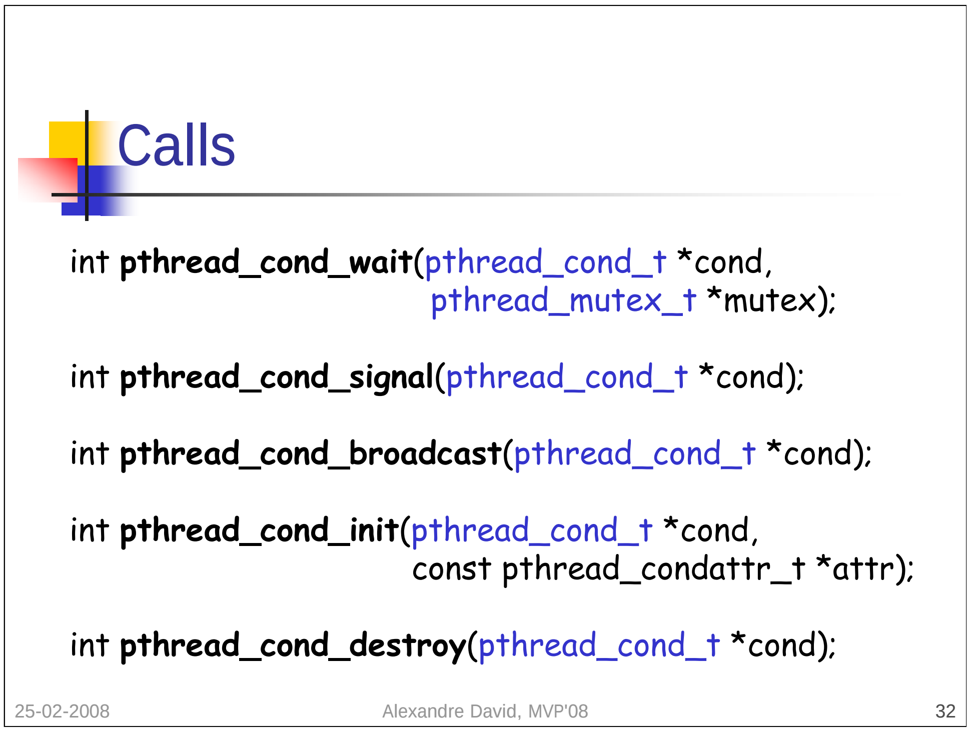
Monitors in Java

```
synchronized void foo() {
        while(!condition) wait();
        <critical section>
        notify();
}
```



Monitors in C#

```
using System.Threading;
…
void foo() {
        Monitor.enter(obj);
        while(!condition) Monitor.wait(obj);
        <critical section>
        Monitor.pulse(obj);
        Monitor.exit(obj);
}
```

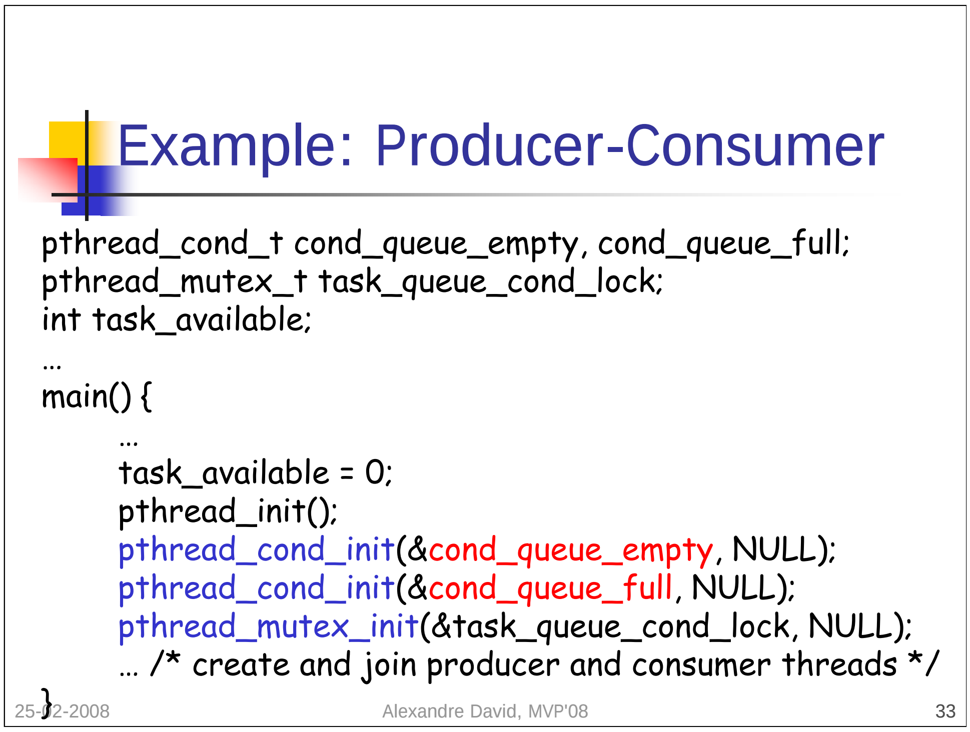


Calls

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

25-02-2008 Alexandre David, MVP'08 32

There is variant pthread_cond_timedwait for a wait with time-out.

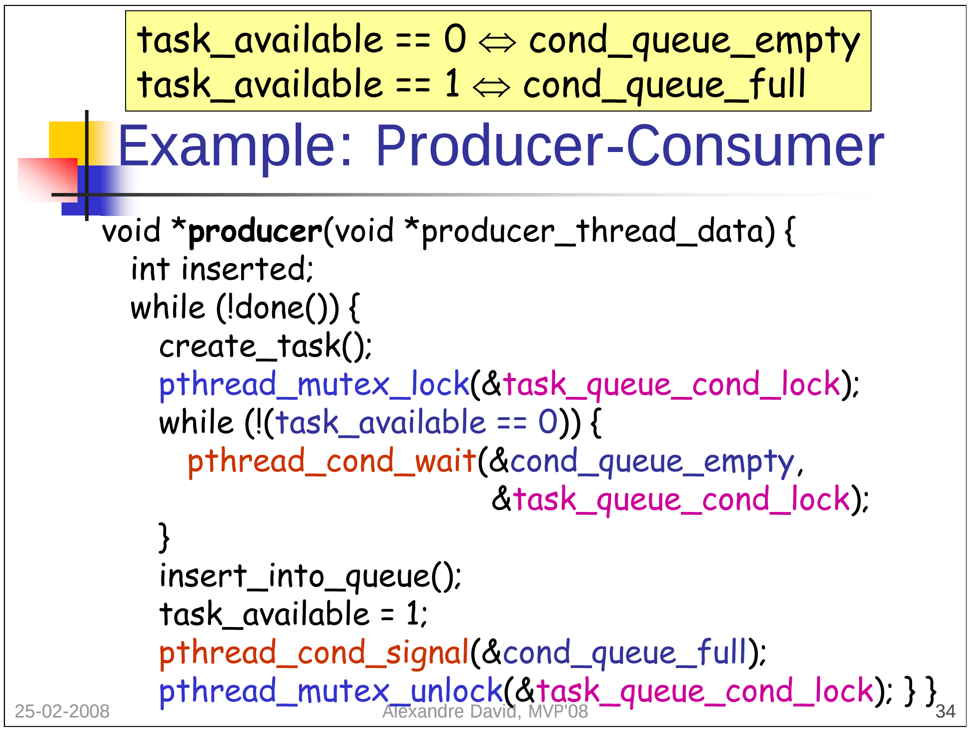


Example: Producer-Consumer

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
…
main() {
        …
        task_available = 0;
        pthread_init();
        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 */
}
```

25-02-2008 Alexandre David, MVP'08 33

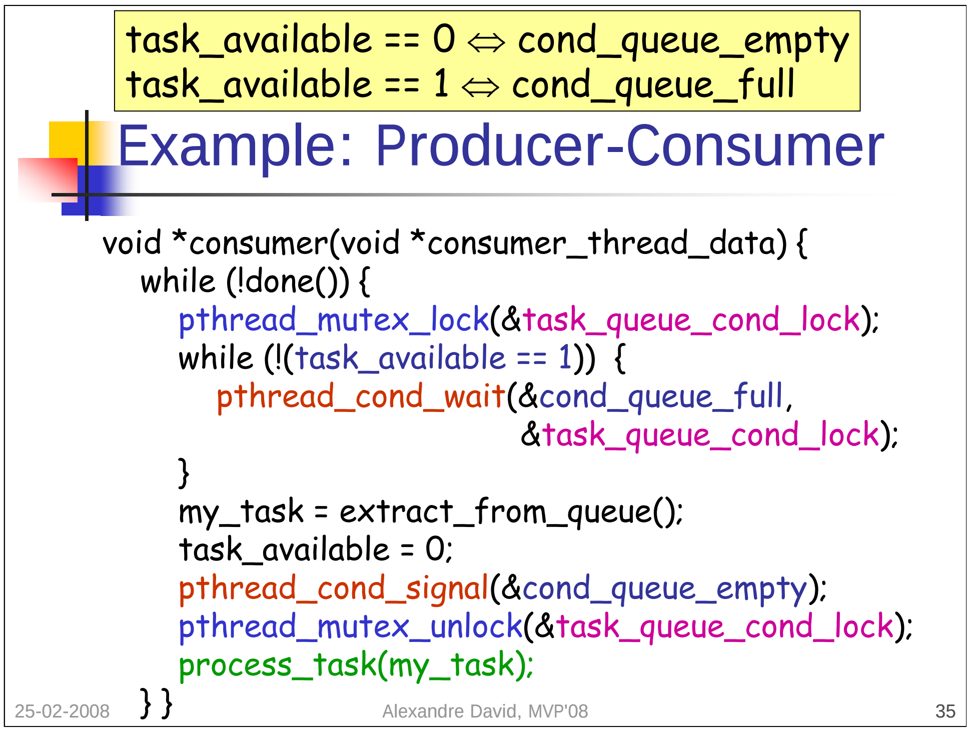
The example is overkill and is here only for pedagogical purposes.



task_available == 0 ⇔ cond_queue_empty
task_available == 1 ⇔ cond_queue_full

Example: Producer-Consumer

```
void *producer(void *producer_thread_data) {
  int inserted;
  while (!done()) {
    create_task();
    pthread_mutex_lock(&task_queue_cond_lock);
    while (!(task_available == 0)) {
      pthread_cond_wait(&cond_queue_empty,
                        &task_queue_cond_lock);
    }
    insert_into_queue();
    task_available = 1;
    pthread_cond_signal(&cond_queue_full);
    pthread_mutex_unlock(&task_queue_cond_lock); } }
```



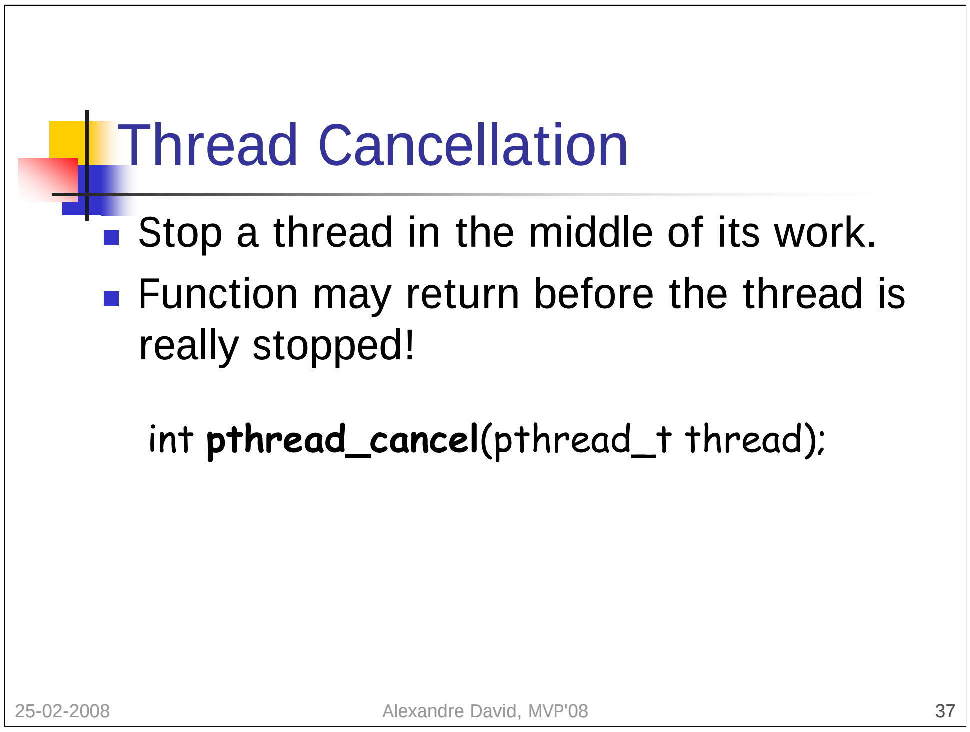
task_available == 0 ⇔ cond_queue_empty
task_available == 1 ⇔ cond_queue_full

Example: Producer-Consumer

```
void *consumer(void *consumer_thread_data) {
  while (!done()) {
    pthread_mutex_lock(&task_queue_cond_lock);
    while (!(task_available == 1))  {
      pthread_cond_wait(&cond_queue_full,
                        &task_queue_cond_lock);
    }
    my_task = extract_from_queue();
    task_available = 0;
    pthread_cond_signal(&cond_queue_empty);
    pthread_mutex_unlock(&task_queue_cond_lock);
    process_task(my_task);
  } }
```

Attribute Objects

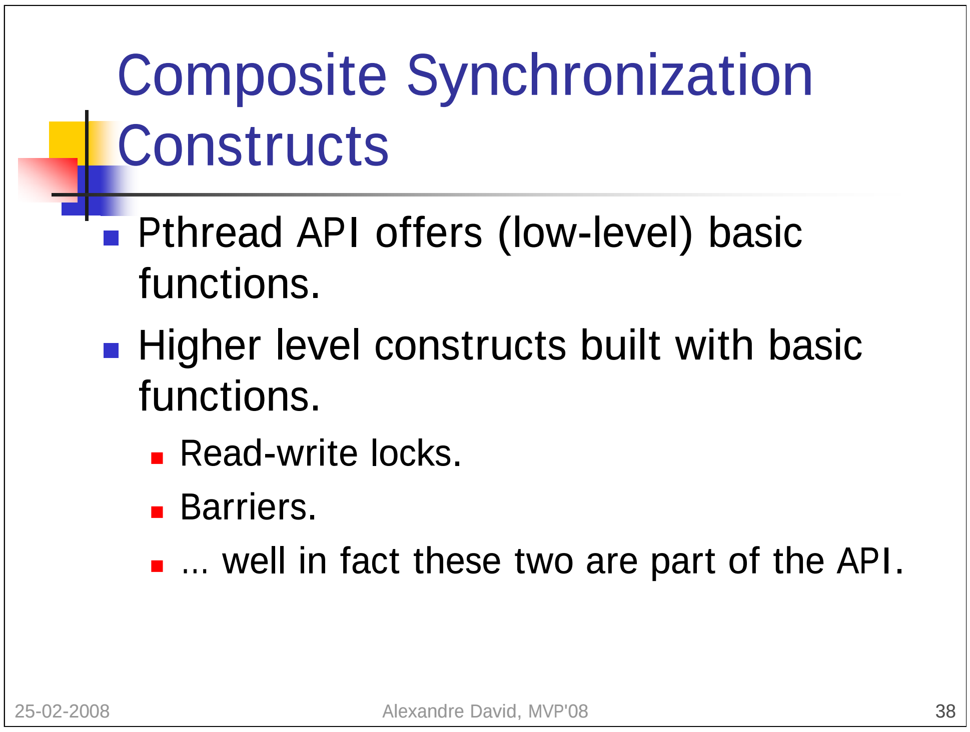
- To control threads and synchronization.
  - Change scheduling policy…
  - Specify mutex types.
- Types of mutexes:
  - Normal – 1 lock per thread or deadlock.
  - Recursive – several locks per thread OK.
  - Error check – 1 lock per thread or error.



Thread Cancellation

- Stop a thread in the middle of its work.
- Function may return before the thread is really stopped!

```
int pthread_cancel(pthread_t thread);
```

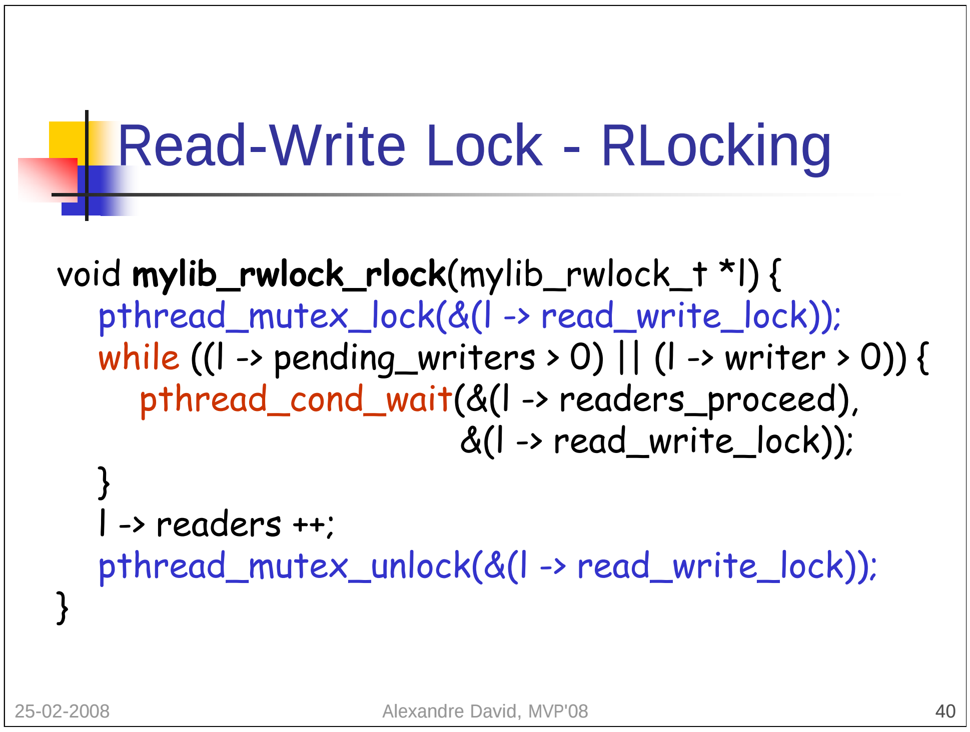


Composite Synchronization Constructs

- Pthread API offers (low-level) basic functions.
- Higher level constructs built with basic functions.
  - Read-write locks.
  - Barriers.
  - ... well in fact these two are part of the API.

Read-Write Locks

- Read often/write sometimes.
  - Multiple reads/unique write.
  - Priority of writers over readers.
- Use condition variables.
  - Count readers and writers.
  - readers_proceed
    ⇔ pending_writers == 0 && writer == 0.
  - writer_proceed
    ⇔ writer == 0 && readers == 0.

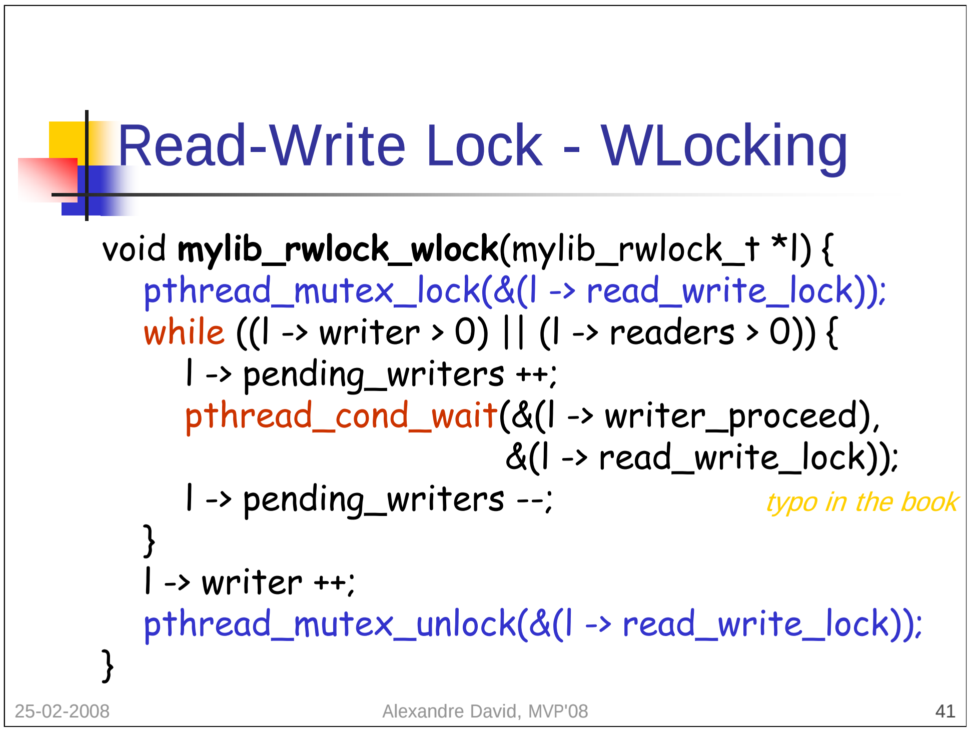


Read-Write Lock - RLocking

```
void mylib_rwlock_rlock(mylib_rwlock_t *l) {
    pthread_mutex_lock(&(l -> read_write_lock));
    while ((l -> pending_writers > 0) || (l -> writer > 0)) {
        pthread_cond_wait(&(l -> readers_proceed),
                                &(l -> read_write_lock));
    }
    l -> readers ++;
    pthread_mutex_unlock(&(l -> read_write_lock));
}
```

25-02-2008 Alexandre David, MVP'08 40

Notice that there is no signal here.



Read-Write Lock - WLocking

```
void mylib_rwlock_wlock(mylib_rwlock_t *l) {
    pthread_mutex_lock(&(l -> read_write_lock));
    while ((l -> writer > 0) || (l -> readers > 0)) {
        l -> pending_writers ++;
        pthread_cond_wait(&(l -> writer_proceed),
                          &(l -> read_write_lock));
        l -> pending_writers --;
    }
    l -> writer ++;
    pthread_mutex_unlock(&(l -> read_write_lock));
}
```

typo in the book

25-02-2008 Alexandre David, MVP'08 41

There is a mistake in the book for the while loop. Either you move the l->pending_writers-- inside the while loop, which is logical w.r.t. “pending” writers, or you move the l->pending_writers++ outside the loop. Keeping it inside is utterly incorrect.

Read-Write Lock - Unlocking

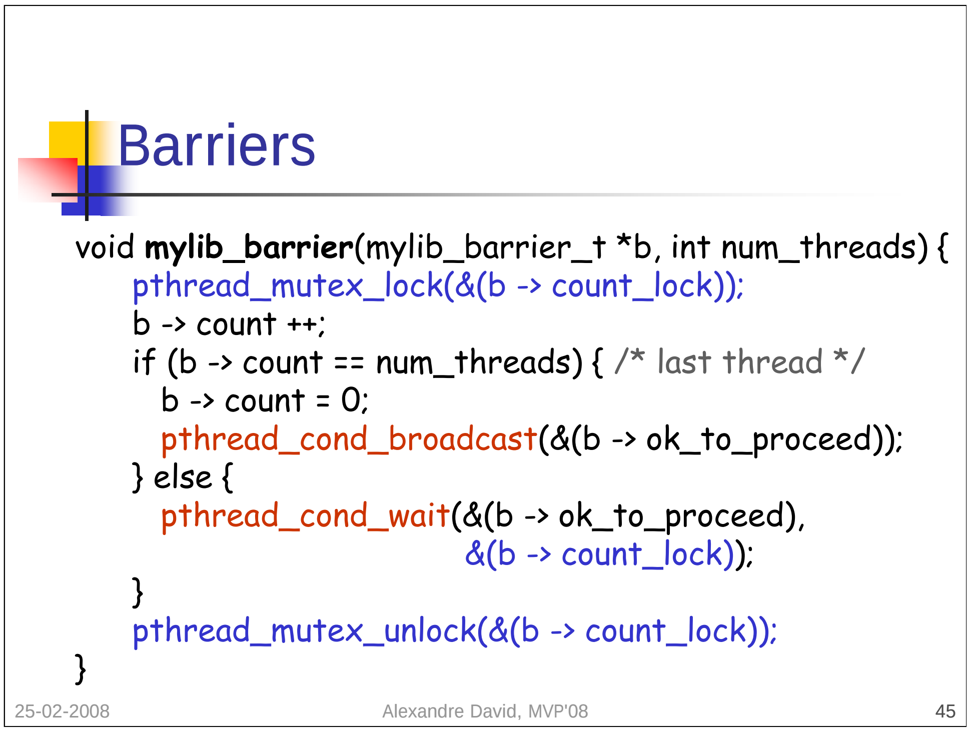
```
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
  pthread_mutex_lock(&(l -> read_write_lock));
  if (l -> writer > 0) {
    l -> writer = 0;
  } else if (l -> readers > 0) {
    l -> readers --;
  }
  if ((l -> readers == 0) && (l -> pending_writers > 0)) {
    pthread_cond_signal(&(l -> writer_proceed));
?  } else if (l -> readers > 0) {                        bug
    pthread_cond_broadcast(&(l -> readers_proceed));
  }
  pthread_mutex_unlock(&(l -> read_write_lock)); }
```

Another Bug

- Example 7.7 has a bug.
- Test & update is not atomic as you can see.
- Fix: Re-test after the write lock has been obtained.
- BTW: The read-lock is useless here.
  What you should do: Acquire a write lock, test and update.

Barriers

- Encoded with
  - a counter,
  - a mutex, and
  - a condition variable.
- Idea:
  - Count & block threads.
  - Signal them all.



Barriers

```
void mylib_barrier(mylib_barrier_t *b, int num_threads) {
    pthread_mutex_lock(&(b -> count_lock));
    b -> count ++;
    if (b -> count == num_threads) { /* last thread */
      b -> count = 0;
      pthread_cond_broadcast(&(b -> ok_to_proceed));
    } else {
      pthread_cond_wait(&(b -> ok_to_proceed),
                        &(b -> count_lock));
    }
    pthread_mutex_unlock(&(b -> count_lock));
}
```

Performance bottleneck: The mutex serializes all the threads, execution time is O(n). Possible to improve by grouping threads by pairs.

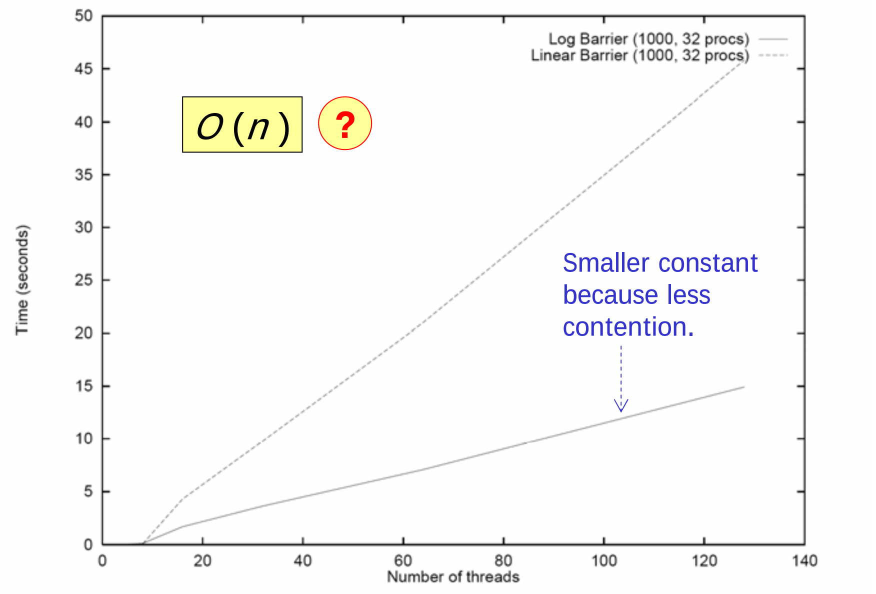
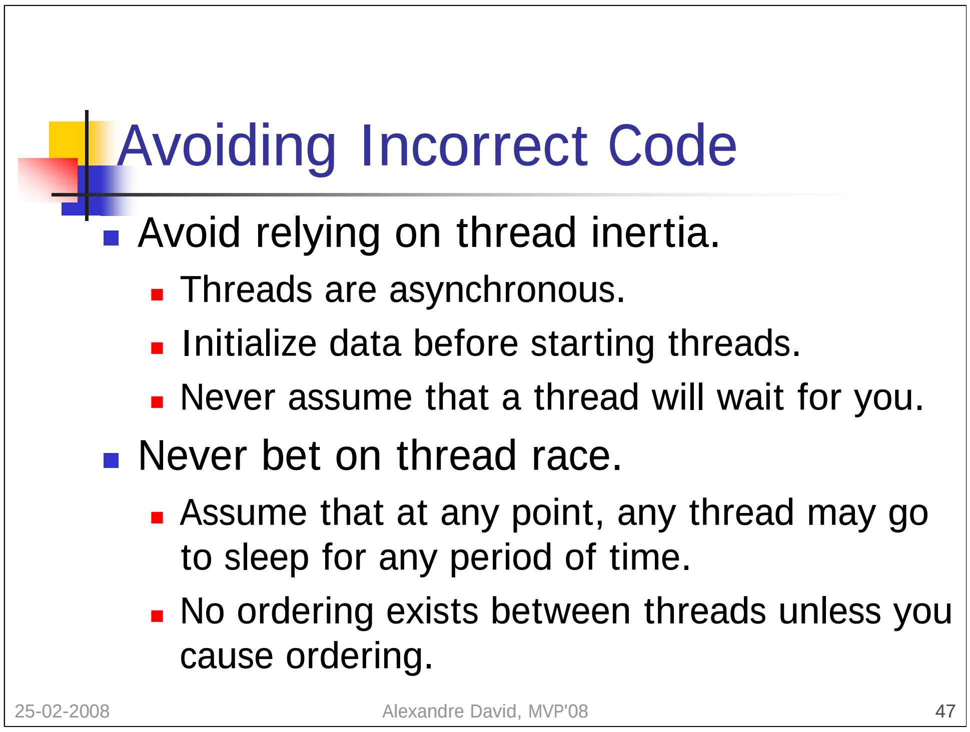
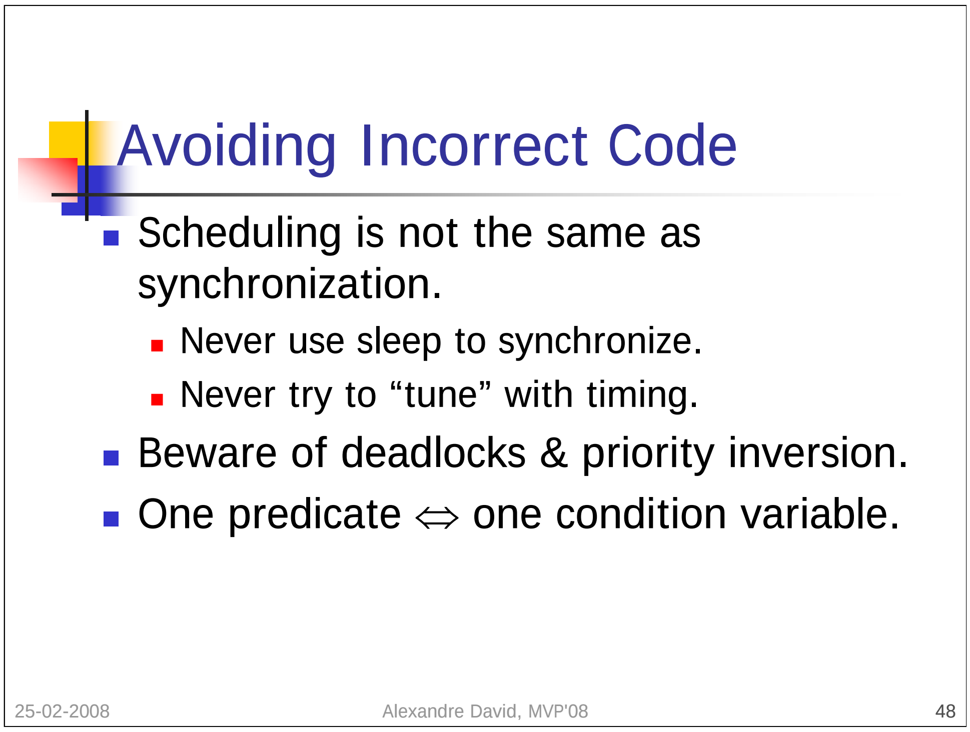


Figure 7.3 Execution time of 1000 sequential and logarithmic barriers as a function of number of threads on a 32 processor SGI Origin 2000.



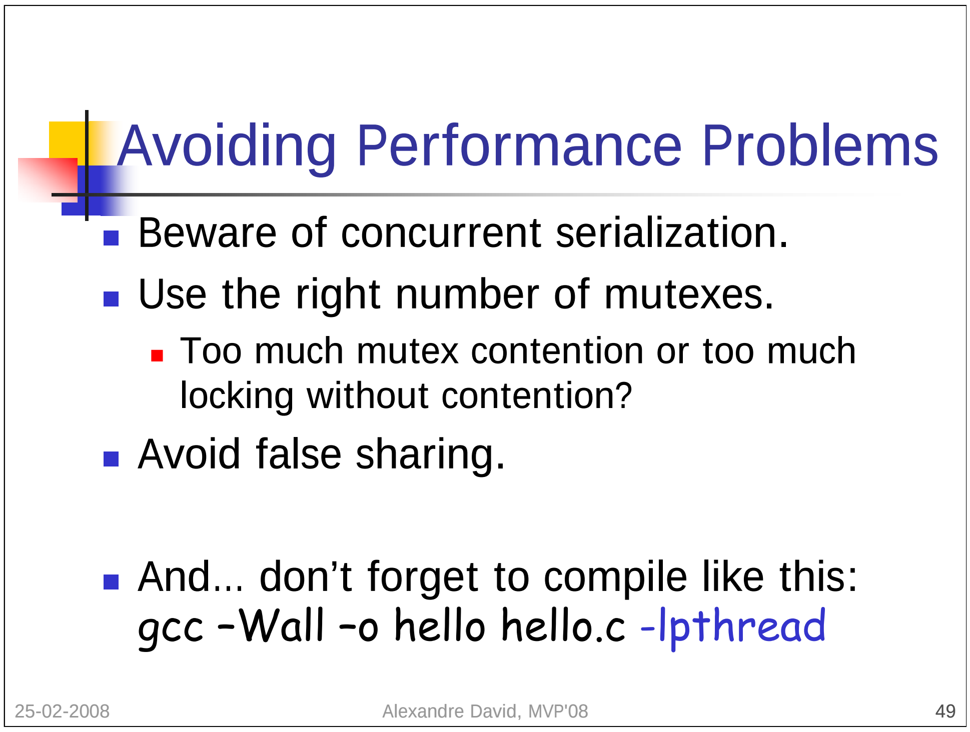
Avoiding Incorrect Code

- Avoid relying on thread inertia.
  - Threads are asynchronous.
  - Initialize data before starting threads.
  - Never assume that a thread will wait for you.
- Never bet on thread race.
  - Assume that at any point, any thread may go to sleep for any period of time.
  - No ordering exists between threads unless you cause ordering.



Avoiding Incorrect Code

- Scheduling is not the same as synchronization.
  - Never use sleep to synchronize.
  - Never try to “tune” with timing.
- Beware of deadlocks & priority inversion.
- One predicate $\Leftrightarrow$ one condition variable.



Avoiding Performance Problems

- Beware of concurrent serialization.
- Use the right number of mutexes.
  - Too much mutex contention or too much locking without contention?
- Avoid false sharing.

- And… don't forget to compile like this:
  gcc -Wall -o hello hello.c -lpthread