Programming Shared Address Space Platforms (cont.)

Alexandre David

B2-206

Today

- Finish thread synchronization.
  - Effort for pthreads = synchronization.
  - Effort for MPI = communication.
- OpenMP.
- Extra material on threads. ☺

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);
```

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);
```

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); }}}
```

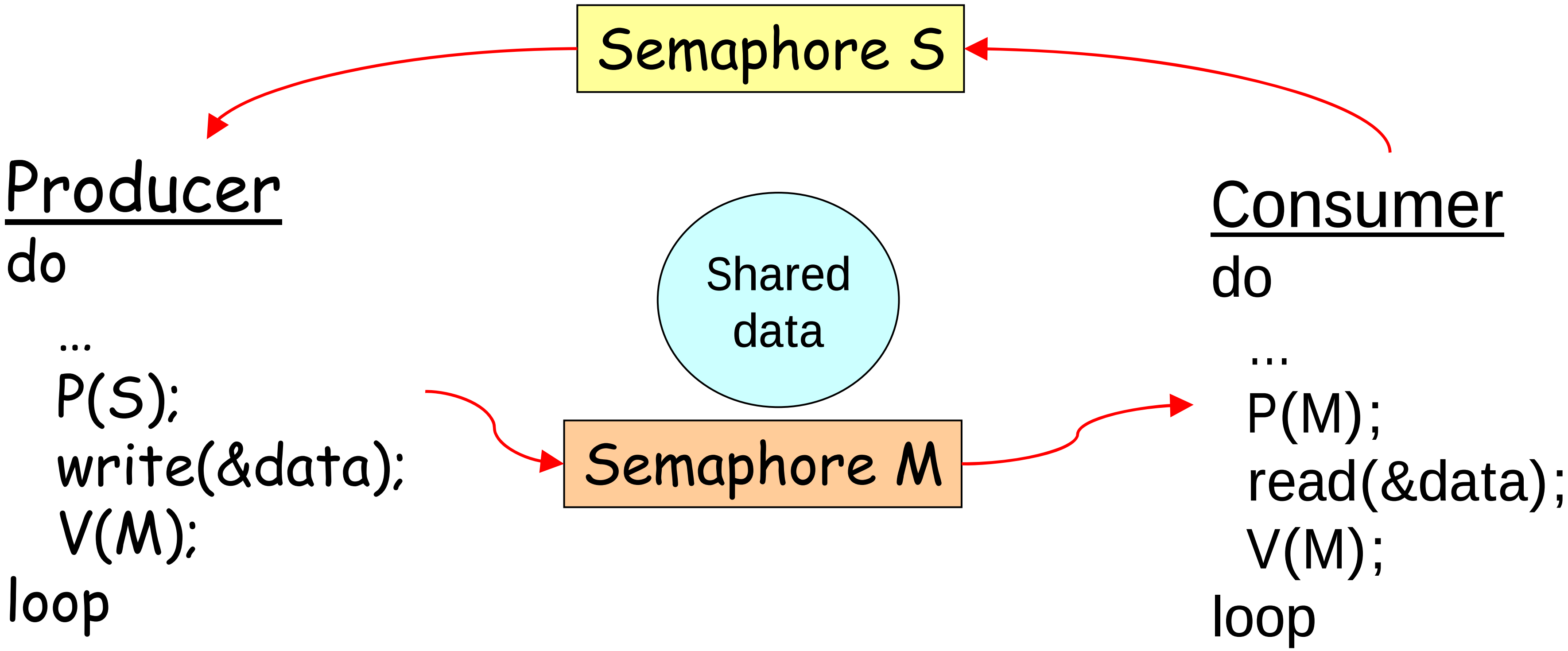
local

critical section

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);
  }
}
```

Producer-Consumer with Semaphores – Recall



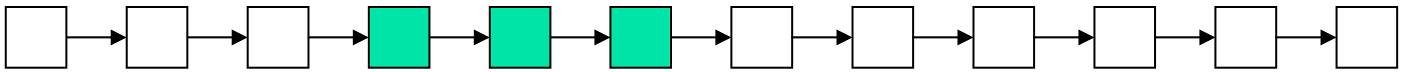
Overhead of Locking

- 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.

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);
}
```

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);
}
```

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);
    }
}
```

Try-lock

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

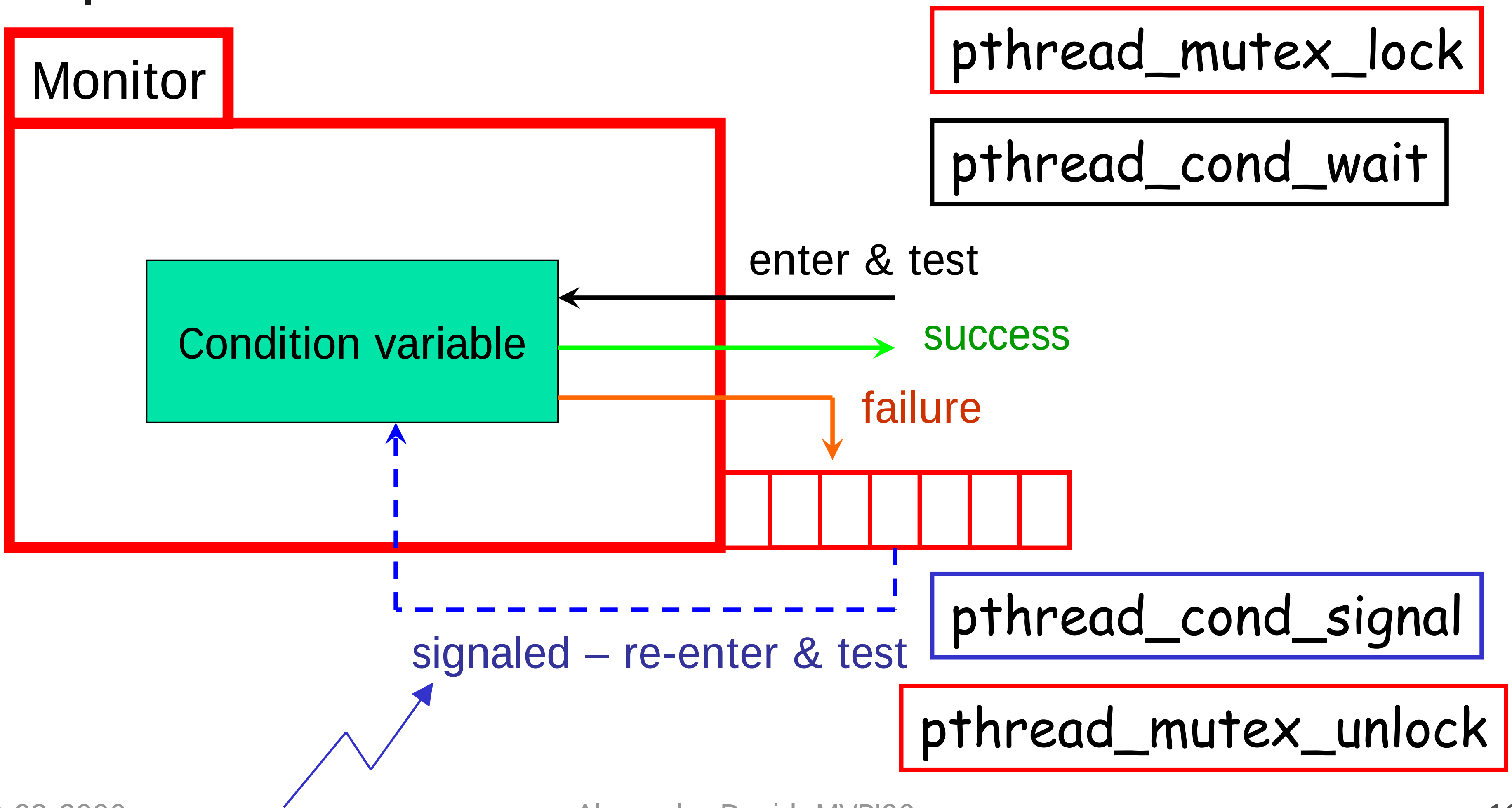
Condition Variables for Synchronization

- How **monitors** are implemented here.

One condition variable ⇔ one predicate.

- 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.

Monitors


Monitor
Condition variable
enter & test
success
failure
signaled – re-enter & test
pthread_mutex_lock
pthread_cond_wait
pthread_cond_signal
pthread_mutex_unlock

Monitors in with Pthread

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

Monitors in Java

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

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);
```

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 */
}
```

Example: Producer-Consumer

```
void *producer(void *producer_thread_data) {
```

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

```
    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); } }
```

Example: Producer-Consumer

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

```
        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.

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));
}
```

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));
}
```

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) {
    pthread_cond_broadcast(&(l -> readers_proceed));
  }
  pthread_mutex_unlock(&(l -> read_write_lock)); }
```

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.

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));
}
```

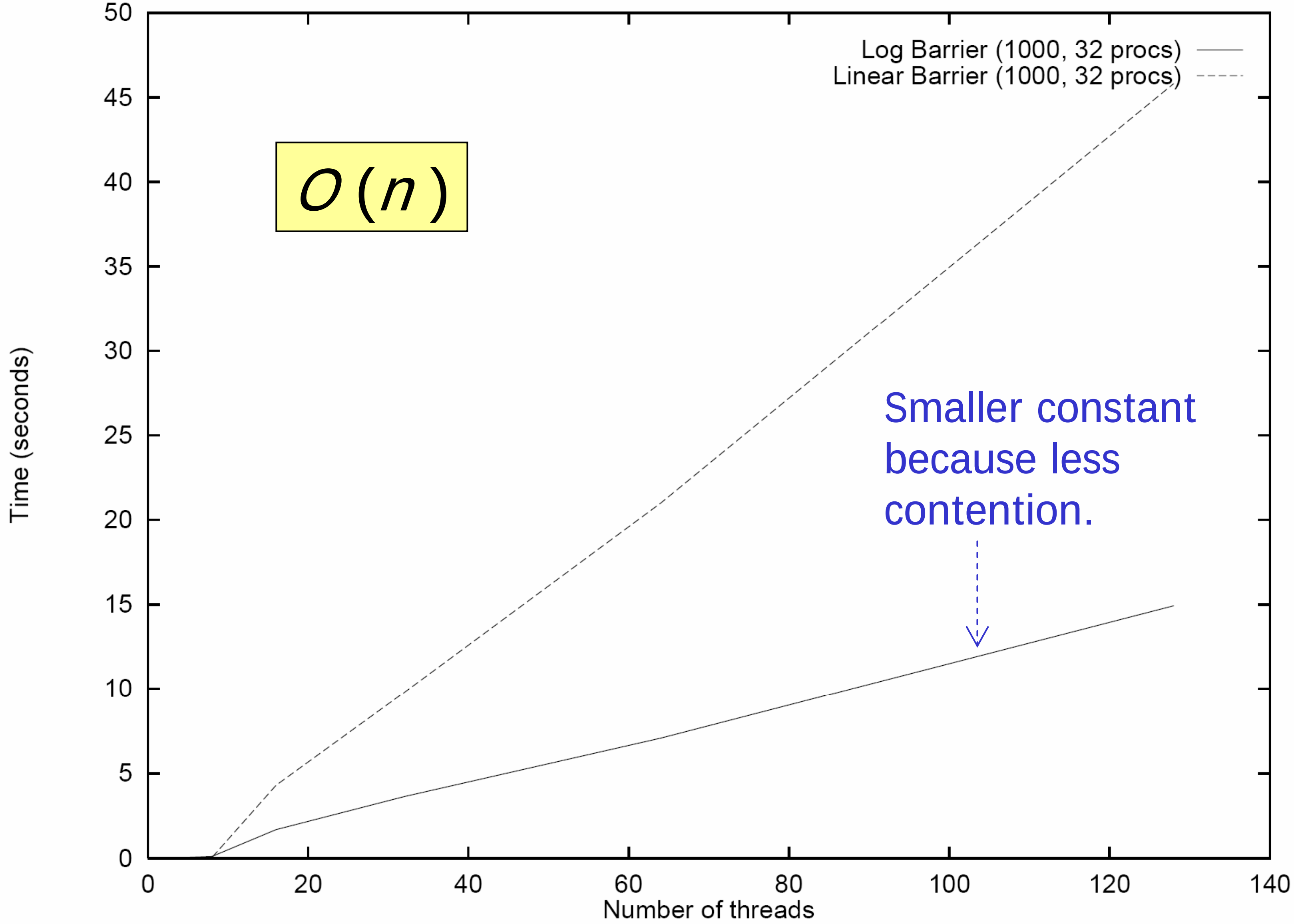


Figure 7.3 Execution time of 1000 sequential and logarithmic barriers as a function of number of threads on a 32 processor SGI Origin 2000.

OpenMP

- Directive based parallel programming.
- Support for concurrency, synchronization ... without explicit mutex, condition variable ...

OpenMP Programming Model

- Uses the #pragma compiler directive.
- OpenMP compiler as pre-processor.
- Execute serially until
  #pragma omp parallel.
- Different clauses to specify
  - conditional parallelization
  - degree of concurrency
  - data handling (local or shared).

```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
    // serial segment

    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);

    // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;

    // parallel segment

}
```

Code inserted by the OpenMP compiler

Corresponding Pthreads translation

Figure 7.4 A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

Example

```
#pragma omp parallel default(private) shared (npoints) \
                    reduction(+: sum) num_threads(8)
{
   sum = 0;
   #pragma omp for
   for (i = 0; i < npoints; 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) +
          (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum ++;
   }
}
```

OpenMP

- More directives:
  - Assign iterations to thread (map data to threads) – scheduling of loops.
  - Synchronize (or not) accross multiple for loops.
  - Specify non-iterative parallel tasks – sections.
  - Specify barriers, single thread execution – barrier, single, master.
  - Specify critical sections – critical.

OpenMP Library

- #include <omp.h>
- Access to OpenMP functions.
  - Number of threads.
  - Thread creation.
  - Mutual exclusion.

Hints to Avoid Debugging

- Chapter 8 of *Programming with POSIX Threads*.
- Hints to avoid mistakes.
- You will get a copy of the chapter.

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 ⇔ 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`