



Programming Shared Address Space Platforms (Chapter 7)

Alexandre David
B2-206

This is about pthreads.



Today

- Thread Basics (7.1 – 7.4).
- Synchronization Primitives in Pthreads (7.5).



Comparison

- Explicit parallel programming: specify tasks and interactions.
 - Communication of intermediate results.
 - Synchronization.
- MPI: focus on communication.
- Shared memory: focus on synchronization.

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.



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>
```

Header.

```
int pthread_create(
```

```
pthread_t *thread_handle,
```

Identifier.

```
const pthread_attr_t *attribute,
```

NULL for default.

```
void* (*thread_function)(void *),  
void *arg);
```

Function to call with its argument.

21-03-2006

Alexandre David, MVP'06

8

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, ← Thread to wait for.  
    void ** ptr);
```

Threads call pthread_exit(value).
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.



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];
    }
}
21-03-2006 Alexandre David, MVP'06 10
```

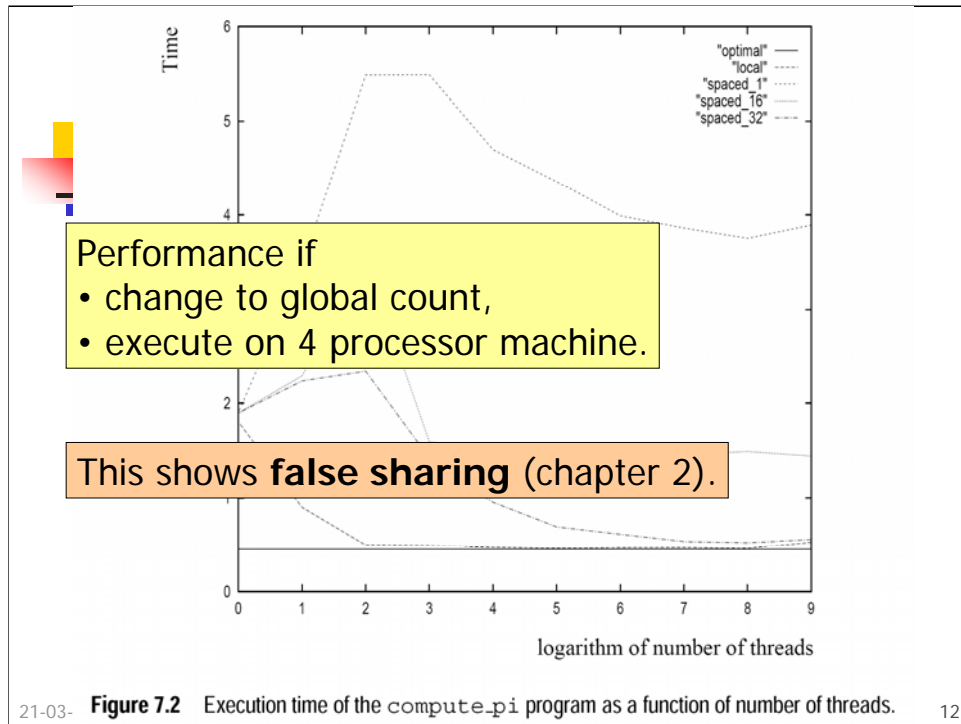
This is a lousy computation of pi. Based on area ratios. Take a square 1x1 and put a circle inside. Square area = 1, circle area = $\pi r^2 = \pi/4$ ($r = 1/2$). Choose many points randomly and the ratio hits/total will converge towards $\pi/4$.



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) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
Count hits
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.



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**.

21-03-2006

Alexandre David, MVP'06

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 be 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_mutexattr_t *lock_attr);  
  
int pthread_mutex_lock(  
    pthread_mutex_t *mutex_lock);  
  
int pthread_mutex_unlock(  
    pthread_mutex_t *mutex_lock);
```

21-03-2006

Alexandre David, MVP'06

15

Return value, as usual.

Do not unlock an already unlocked mutex.

When unlocking a mutex, there may be another thread that will be unblocked. The choice depends on the scheduler, i.e., you don't know and you shouldn't assume anything.

NULL argument means default value – but no NULL for 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);
    ...
}
```

21-03-2006

Alexandre David, MVP'06

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.