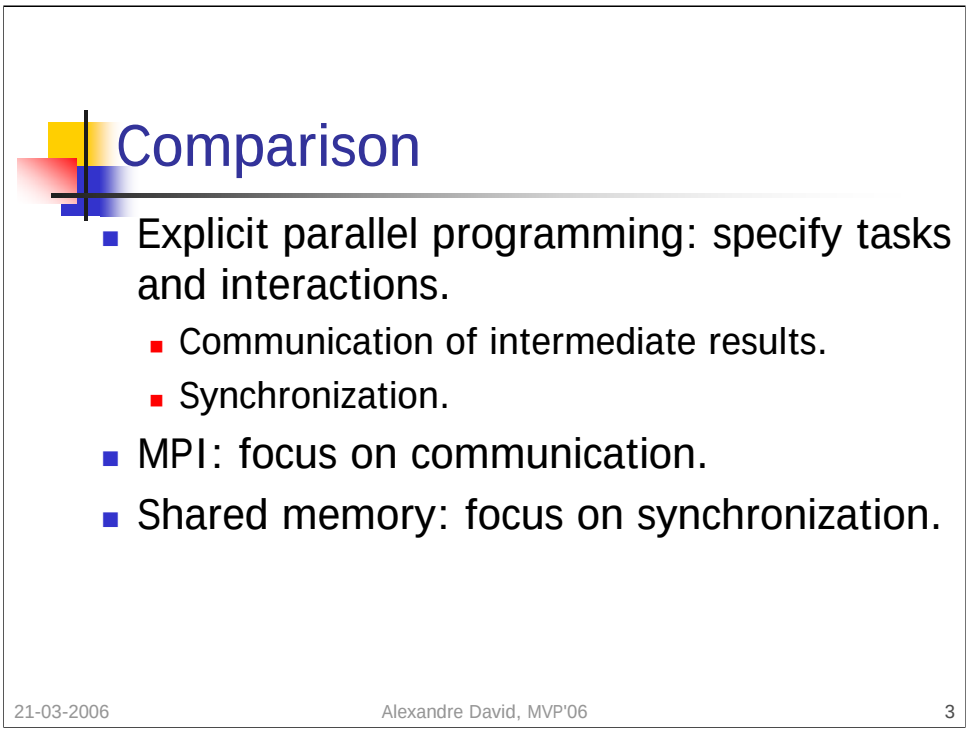
This is about pthreads.

Today

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



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

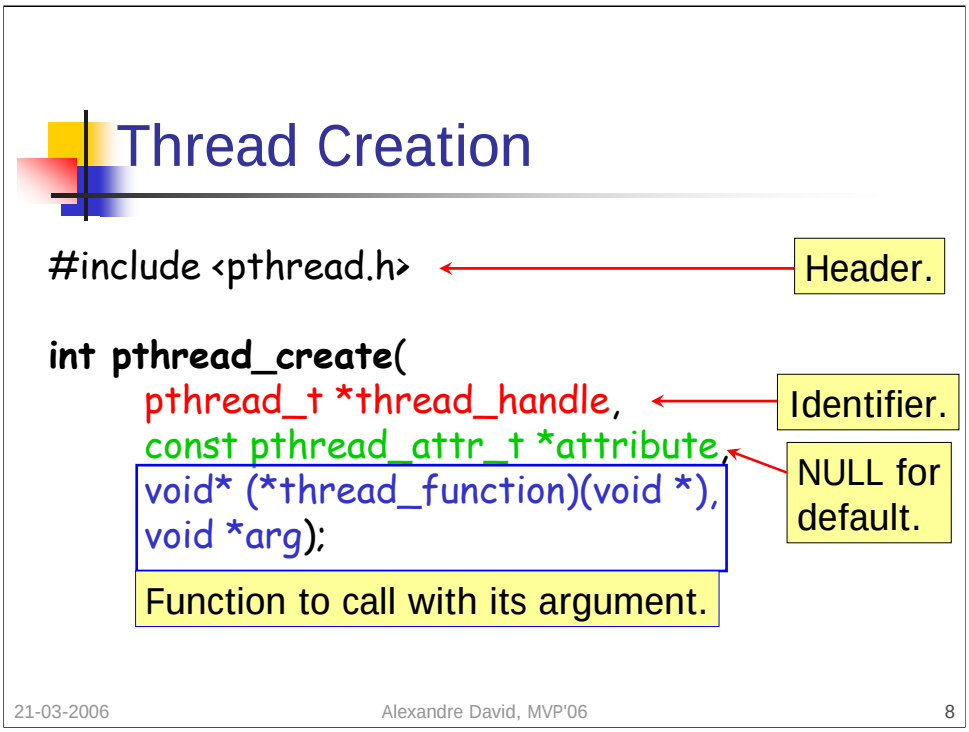
- All memory is globally accessible.
- But the stack is considered local.
  - In practice both local (private) and global (shared) memory.
  - Recall that memory is physically distributed and local accesses are faster.

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.



Invokes *thread_function* as a thread.

Notes:

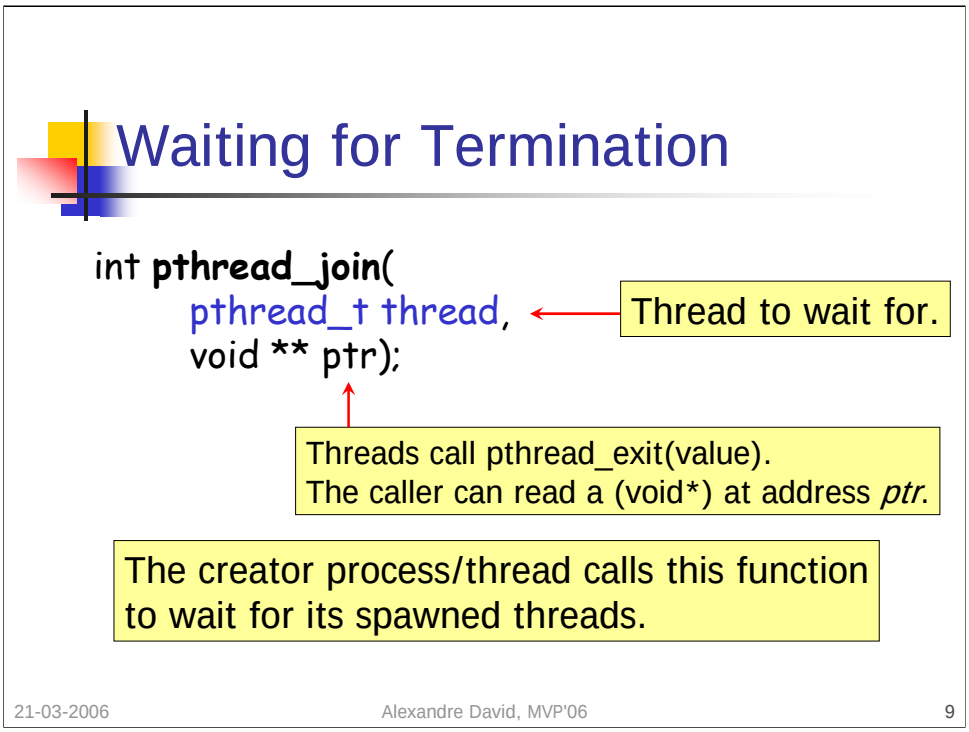
•The identifier *thread_handle* is written before the function returns.

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

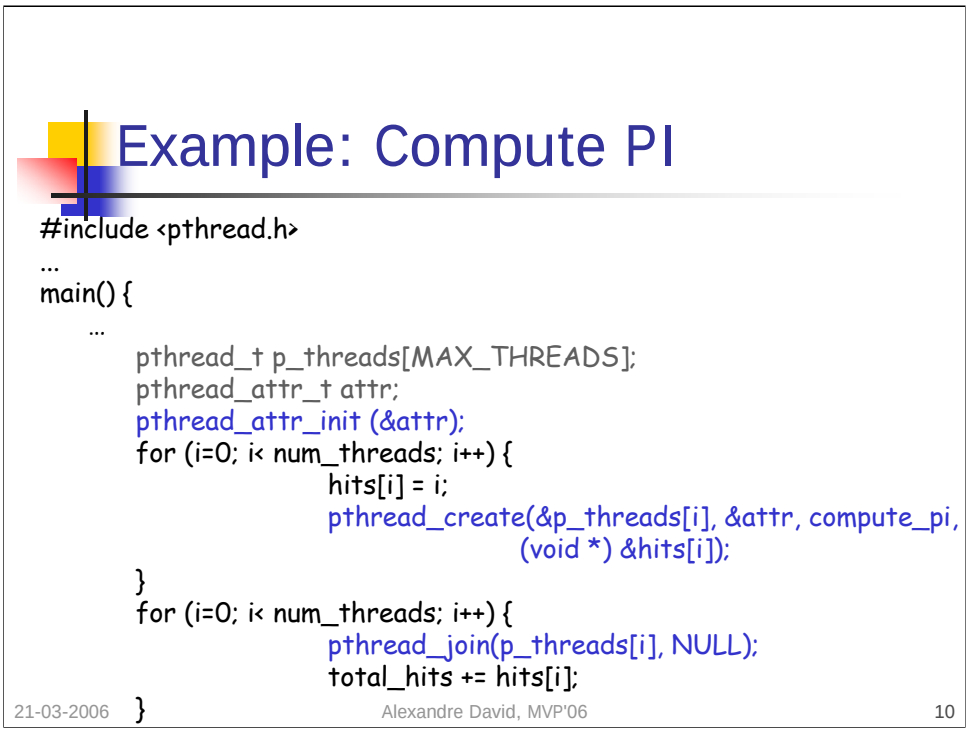
•On uni-processor machines the thread may preempt its creator thread.

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

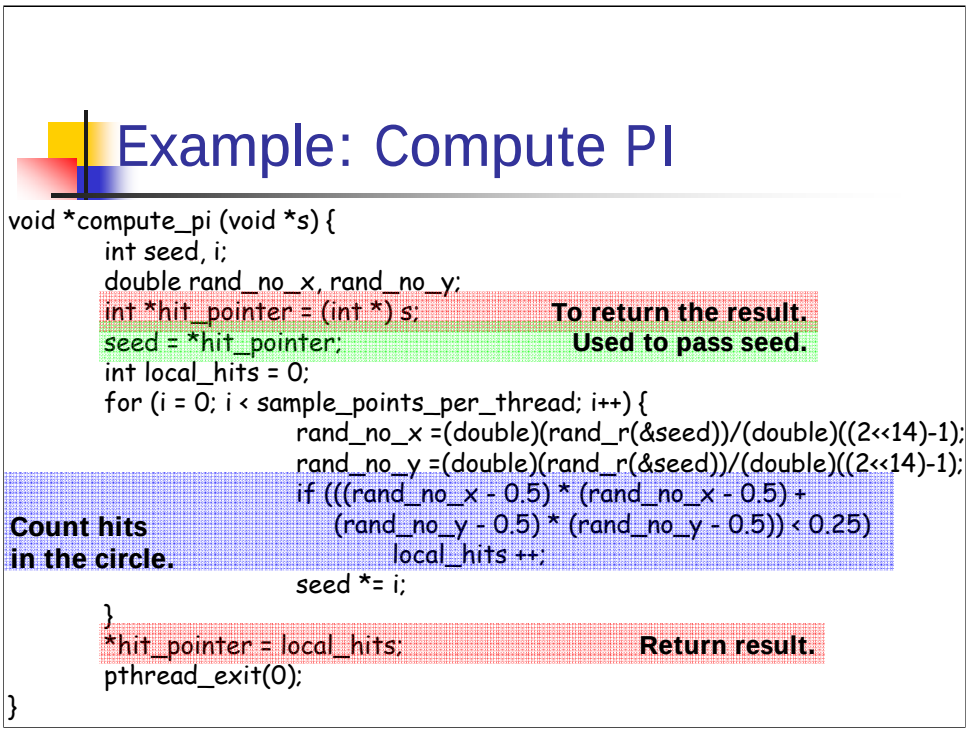
•Beware of race conditions: Make sure to initialize everything before creating the thread (and not after).



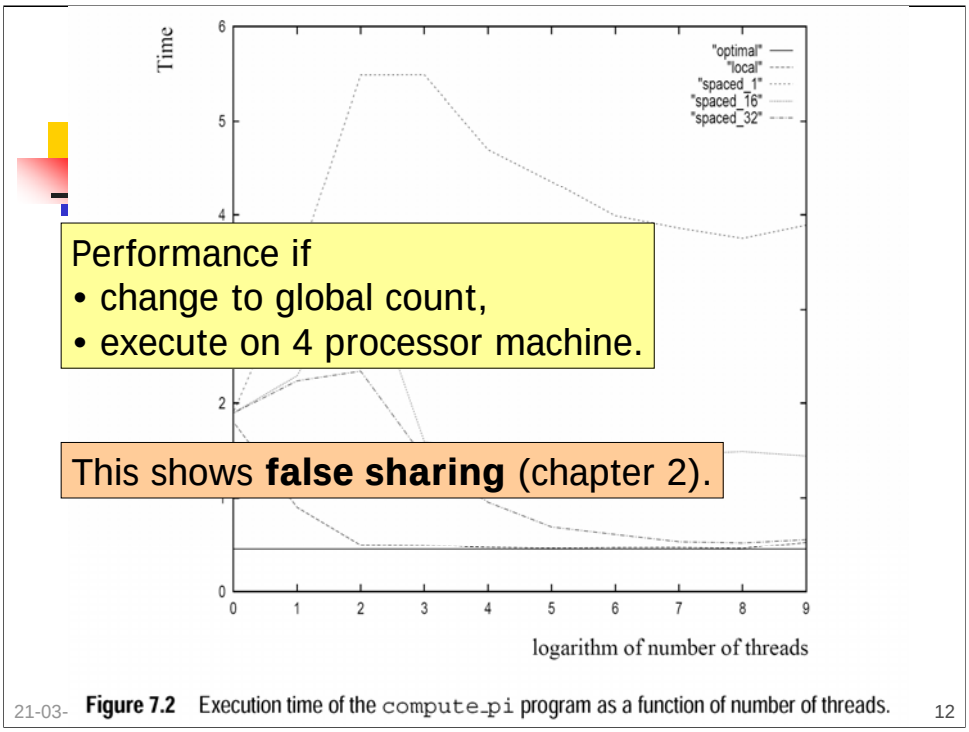
And returns success (0) or an error code.



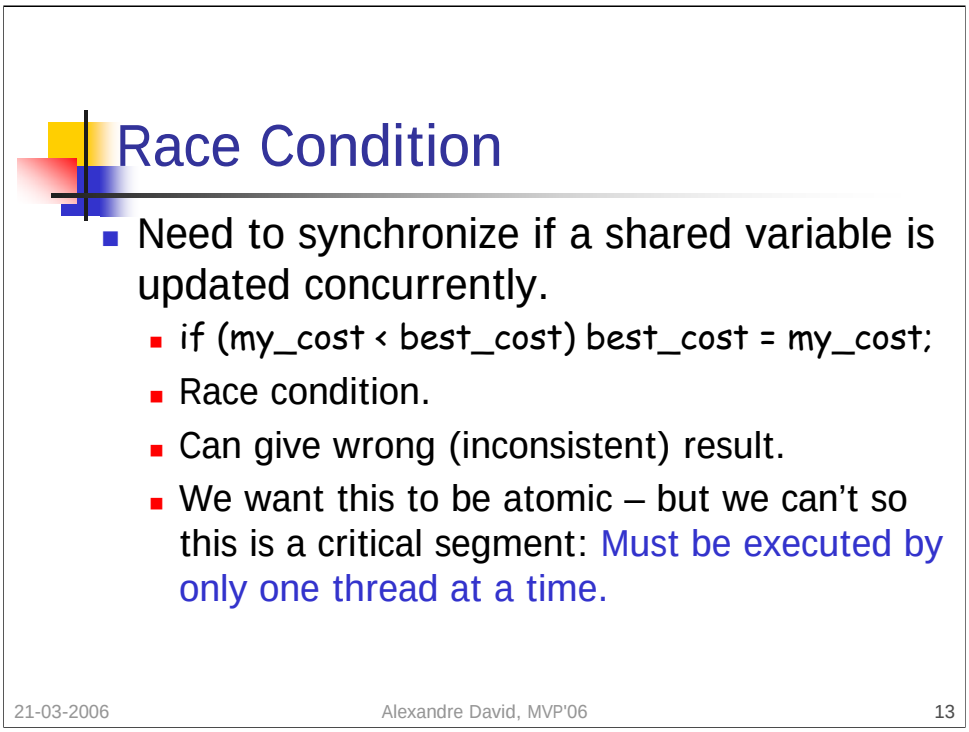
This is a lousy computation of pi. Based on area ratios. Take a square 1x1 and put a circle inside. Square area = 1, circle area = $\pi r^2 = \pi/4$ ($r = ½$). Choose many points randomly and the ratio hits/total will converge towards $\pi/4$.



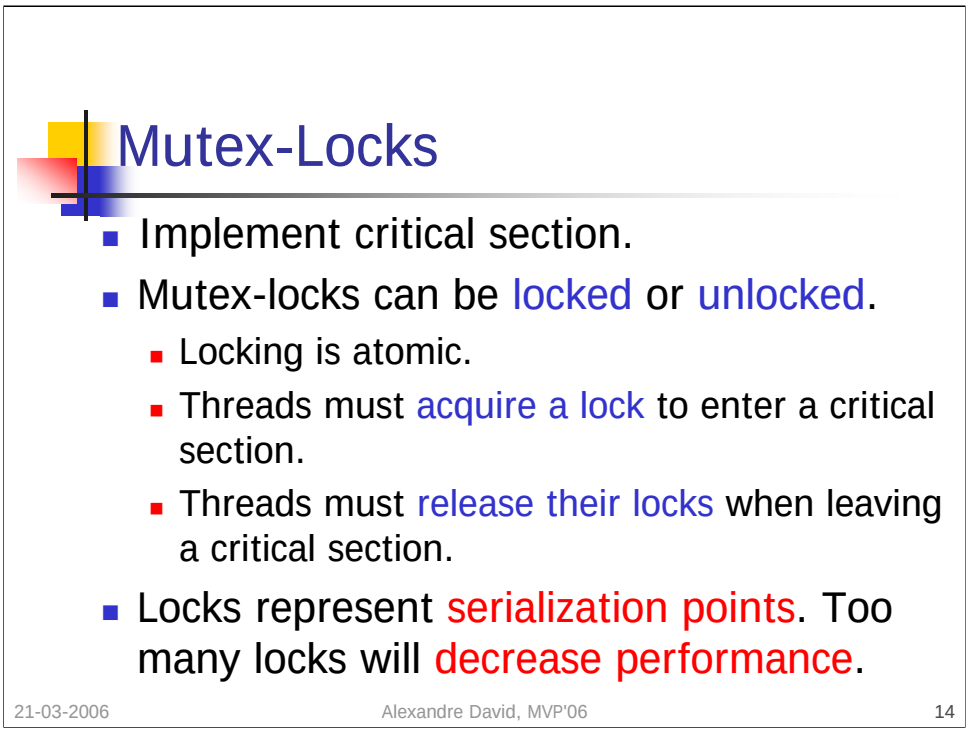
Call to **rand_r**, worse than drand48 or rand, because we need a **reentrant** function.



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



Race condition: 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).



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.

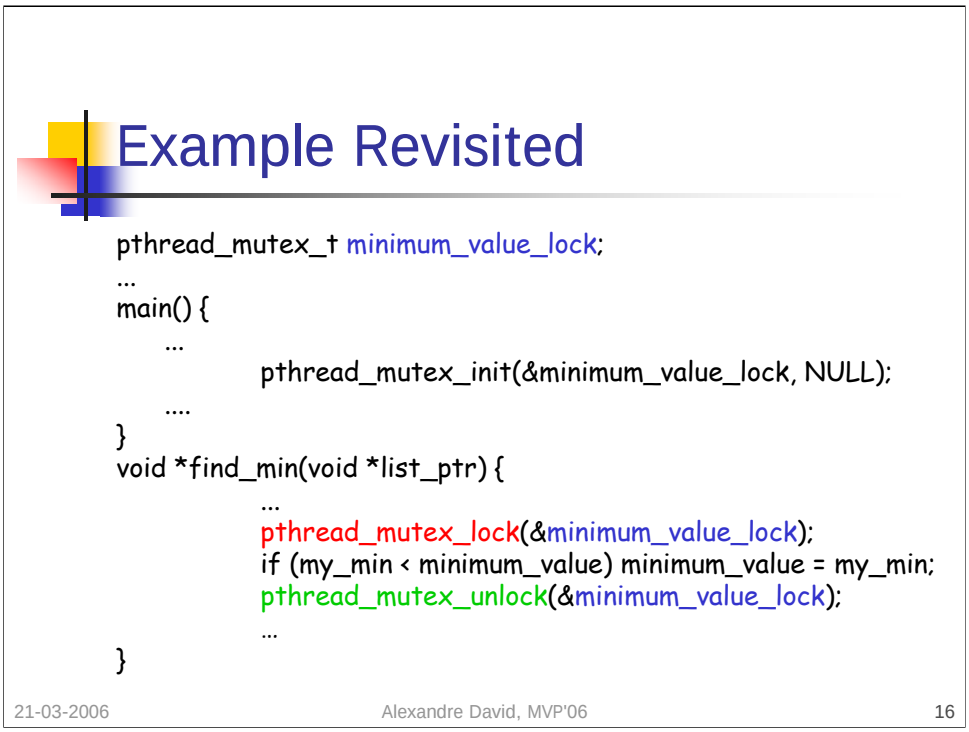
Be also careful on the granularity of what you lock. Locking big portions of code is bad since you are killing parallelism for the code you are locking.

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!



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.