POSIX Threads: a first step toward parallel programming

George Bosilca
bosilca@icl.utk.edu
Process vs. Thread

- A process is a collection of virtual memory space, code, data, and system resources.
- A thread (lightweight process) is code that is to be serially executed within a process.
- A process can have several threads.

Threads executing the same block of code maintain separate stacks. Each thread in a process shares that process's global variables and resources.

Possible to create more efficient applications?
Hardware Threads

• Hardware control switching between threads to hide latencies (memory accesses / operations)
  – Different switching policies: cache miss, after each operation
  – Hardware maintain independent state for each thread (registers)
  – Possible to switching threads in one cycle (TERA machine)
Terminology

• Lightweight Process (LWP): or kernel thread
• Contention Scope: how threads compete for system resources
• Thread-safe a program that protects the shared data for its threads (mutual exclusion)
• Reentrant code: a program that can have more than one thread executing concurrently.
• Async-safe means that a function is reentrant while handling a signal (i.e. can be called from a signal handler).
• Concurrency vs. Parallelism - They are not the same! Parallelism implies simultaneous running of code (which is not possible, in the strict sense, on uniprocessor machines) while concurrency implies that many tasks can run in any order and possibly in parallel.
Thread vs. Process

• Threads share the address space of the process that created it; processes have their own address space.
• Threads have direct access to the data segment of its process; processes have their own copy of the data segment of the parent process.
• Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.
• Threads have almost no overhead; processes have considerable overhead.
• New threads are easily created; new processes require duplication of the parent process.
• Threads can exercise considerable control over threads of the same process; processes can only exercise control over child processes.
• Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process; changes to the parent process does not affect child processes.

2.2.2 The Classical Thread Model in Modern Operating Systems 3e by Tanenbaum
Process vs. Thread

• Multithreaded applications must avoid two threading problems: deadlocks and race conditions.
• A deadlock occurs when each thread is waiting for the other to do something.
• A race condition occurs when one thread finishes before another on which it depends, causing the former to use a bogus value because the latter has not yet supplied a valid one.
The key is synchronization

- Ordering memory accesses
  - Flush / memory barrier
  - Volatile (poor-man solution in C)
- Synchronization = gaining access to a shared resource.
- Synchronization REQUIRE cooperation.

Thread 1

```c
x = 1;
barrier();
y = 5;
x = 2;
```

Thread 2

```c
y = 1;
barrier();
while (x==1) ;
printf( "%d\n", y );
```
POSIX Thread

• What’s POSIX?
  – Widely used UNIX specification
  – Most of the UNIX flavor operating systems

POSITX is the Portable Operating System Interface, the open operating interface standard accepted world-wide. It is produced by IEEE and recognized by ISO and ANSI.
# Pthread API

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_</td>
<td>Thread management (create/destroy/cancel/join/exit)</td>
</tr>
<tr>
<td>pthread_attr_</td>
<td>Thread attributes</td>
</tr>
<tr>
<td>pthread_mutex_</td>
<td>Mutexes</td>
</tr>
<tr>
<td>pthread_mutexattr_</td>
<td>Mutexes attributes</td>
</tr>
<tr>
<td>pthread_cond_</td>
<td>Condition variables</td>
</tr>
<tr>
<td>pthread_condattr_</td>
<td>Condition attributes</td>
</tr>
<tr>
<td>pthread_key_</td>
<td>Thread-specific data key (TLS)</td>
</tr>
<tr>
<td>pthread_rwlock_</td>
<td>Read/write locks</td>
</tr>
<tr>
<td>pthread_barrier_</td>
<td>Synchronization barriers</td>
</tr>
</tbody>
</table>
Thread Management (create)

int `pthread_create` (pthread_t *restrict thread,  
    const pthread_attr_t *restrict attr,  
    void *(*start_routine)(void *),  
    void *restrict arg)  

int `pthread_exit` (void *value_ptr)  

int `pthread_cancel` (pthread_t thread)  

int `pthread_attr_init` (pthread_attr_t *attr)  

int `pthread_attr_set*` (pthread_attr_t *restrict attr, *)  

int `pthread_attr_destroy` (pthread_attr_t *attr)

[OUT] thread id  
[IN] attributes  
[IN] thread function  
[IN] argument for thread function  
[OUT] Return to caller/joiner  
[IN] thread to be cancelled  
[OUT] attributes to be initialized  
[IN] set attributed (state, stack)  
[IN] attributed to be destroyed

Attributes: Detached or joinable state, Scheduling inheritance, Scheduling policy, Scheduling parameters, Scheduling contention scope, Stack size, Stack address, Stack guard (overflow) size

Questions:
- Once created what will be the status of the thread and how it will be scheduled by the OS ? (use `sched_setscheduler`)
- Where it will be run ? (use `sched_setaffinity` or HWLOC)
Thread Management (join)

#define _GNU_SOURCE
#include <pthread.h>

int pthread_join (pthread_t thread,
                  void **value_ptr)                        [OUT] thread return value

int pthread_detach (pthread_t thread)

int pthread_attr_setdetachstate (pthread_attr_t *attr,
                                  int detachstate)                            [IN/OUT] attribute
                                               [IN] state to be set

int pthread_attr_getdetachstate (const pthread_attr_t *attr,
                                  int *detachstate)                            [OUT] detach state value

Join blocks the calling thread until the target thread terminates, and returns it’s thread_exit argument.

It is impossible to join a thread in a detached state. It is also impossible to reattach it
Thread Management (state)

int pthread_attr_getstacksize (const pthread_attr_t *restrict attr, size_t *restrict stacksize)
int pthread_attr_setstacksize (pthread_attr_t *attr, size_t stacksize)
int pthread_attr_getstackaddr (const pthread_attr_t *restrict attr, void **restrict stackaddr)
int pthread_attr_setstackaddr (pthread_attr_t *attr, void *stackaddr)
pthread_t pthread_self (void)
int pthread_equal (pthread_t t1, pthread_t t2)

The POSIX standard does not dictate the size of a thread's stack!
Threading - example

• Numerical solution to Laplace’s equation

\[
U_{i,j}^{n+1} = \frac{1}{4} \left( U_{i-1,j}^n + U_{i+1,j}^n + U_{i,j-1}^n + U_{i,j+1}^n \right)
\]

for \( j = 1 \) to \( j_{\text{max}} \)

for \( i = 1 \) to \( i_{\text{max}} \)

\[
U_{\text{new}}(i,j) = 0.25 \times \left( U(i-1,j) + U(i+1,j) + U(i,j-1) + U(i,j+1) \right)
\]

end for

end for
Threading - example

• The approach to make it parallel is by partitioning the data
Threading - example

• The approach to make it parallel is by partitioning the data

Overlapping the data boundaries allow computation without communication for each superstep.

On the communication step each processor update the corresponding columns on the remote processors.
for $j = 1$ to $j_{\text{max}}$
  for $i = 1$ to $i_{\text{max}}$
    \[ u_{\text{new}}(i,j) = 0.25 \times (U(i-1,j) + U(i+1,j) + U(i,j-1) + U(i,j+1)) \]
  end for
end for
## Memory Consistency Models

<table>
<thead>
<tr>
<th>Memory Level</th>
<th>Size</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU registers</td>
<td>≈ 100B</td>
<td>0.5ns (1 cycles)</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>64KB – 1M</td>
<td>1ns (few cycles)</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>≈ 1-30 MB</td>
<td>10ns (tens of cycles)</td>
</tr>
<tr>
<td>Main Memory</td>
<td>≈ ? GB</td>
<td>150ns (hundreds of cycles)</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>≈ ? TB</td>
<td>10ms (thousands of cycles)</td>
</tr>
<tr>
<td>Network Storage</td>
<td>≈ ? PB</td>
<td>100ms to 1s (much more)</td>
</tr>
</tbody>
</table>

- Defining a consistent memory models is difficult and not necessarily required for correctness
  – Weaker definitions that are easier to implement and good enough to implement predictable and deterministic applications
Strict (atomic) consistency

- Definition: any read to a memory location X returns the value stored by the most recent write operation to X
  - the most recent covers all computing units in the system

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1 [W(x) 1]</td>
<td>?x ... [R(x) ?]</td>
</tr>
<tr>
<td>?x ... [R(x) ?]</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram showing valid and invalid sequence of operations]

- Valid sequence
  - P1: W(x) → 1
  - P2: R(x) → 1 (two reads, both return 1)

- Invalid sequence
  - P1: W(x) → 1
  - P2: R(x) → 0, then R(x) → 1 (conflict)

- Valid sequence
  - P1: W(x) → 1
  - P2: R(x) → 0, then R(x) → 1

- Invalid sequence
  - P1: W(x) → 1
  - P2: R(x) → 0, then R(x) → 1 (conflict)
Sequential Consistency

• Definition: the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program

  – Lamport ordering
  
  – expanding from the sets of reads and writes that *actually* happened to the sets that *could* have happened, we can reason more effectively about the program
Sequential Consistency

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1 [W(x) 1]</td>
<td>?x ... [R(x) ?]</td>
<td>x = 2 [W(x) 2]</td>
<td>?x ... [R(x) ?]</td>
</tr>
<tr>
<td>?x ... [R(x) ?]</td>
<td></td>
<td>?x ... [R(x) ?]</td>
<td></td>
</tr>
</tbody>
</table>

Cache coherency is **NOT** sequential consistency because seq. consistency requires a **globally** consistency view of memory operations while cache coherency only requires them **locally**.
Cache Coherence

Types of memory accesses:

• **Shared Access**: we can have shared access to variables vs. private access. But the questions we're considering are only relevant for shared accesses.

• **Competing vs. Non-Competing**: If we have two accesses from different processors, and at least one is a write, they are competing accesses. They are considered as competing accesses because the result depends on which access occurs first (if there are two accesses, but they're both reads, it doesn't matter which is first).

• **Synchronizing vs. Non-Synchronizing**: Ordinary competing accesses, such as variable accesses, are non-synchronizing accesses. Accesses used in synchronizing the processes are (of course) synchronizing accesses.

• **Acquire vs. Release**: Finally, we can divide synchronization accesses into accesses to acquire locks, and accesses to release locks.

Can’t happen with a snoopy-cache scheme but it can with a directory-based cache.
Weak Consistency

• Weak consistency results if we only consider competing accesses as being divided into synchronizing and non-synchronizing accesses, and require the following properties:
  – Accesses to synchronization variables are sequentially consistent.
  – No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
  – No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.

\[\begin{array}{c}
P1 & W(x) & 1 & W(x) & 2 & S \\
P2 & & R(x) & 0 & R(x) & 2 & S & R(x) & 2 \\
P3 & & R(x) & 1 & S & R(x) & 2 \\
\end{array}\]
Release consistency

• Weak Consistency (via synchronization) requires that when a synchronization occurs, all processors globally update memory – each local change must be propagated to all processors with a copy of the shared variable, and each processor need to obtain all changes from the others.

• Release consistency consider finer grain locks of memory regions, and only propagates the locked memory (as needed)
  – Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully.
  – Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.
  – The acquire and release accesses must be sequentially consistent.
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive

Active threads

Thread 1

```
... lock
unlock ...
```

Thread 2

```
... lock
unlock ...
```

Thread 3

```
... lock
unlock ...
```
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive
Mutual exclusion

• Spin vs. sleep?

• What’s the desired lock grain?
  – Fine grain – spin mutex
  – Coarse grain – sleep mutex

• Spin mutex: use CPU cycles and increase the memory bandwidth, but when the mutex is unlock the thread continue his execution immediately.
Shared/Exclusive Locks

- **ReadWrite Mutual exclusion**
- Extension used by the reader/writer model
- 4 states: write_lock, write_unlock, read_lock and read_unlock.
- multiple threads may hold a shared lock simultaneously, but only one thread may hold an exclusive lock.
- if one thread holds an exclusive lock, no threads may hold a shared lock.
Shared/Exclusive Locks

Legend
- Active thread
- Sleeping thread

Step 1
- Writer 1
  - rw_lock
  - rw_unlock

- Writer 2
  - rw_lock
  - rw_unlock

- Reader 1
  - rd_lock
  - rd_unlock

- Reader 2
  - rd_lock
  - rd_unlock

Step 2
- Writer 1
  - rw_lock
  - rw_unlock

- Writer 2
  - rw_lock
  - rw_unlock

- Reader 1
  - rd_lock
  - rd_unlock

- Reader 2
  - rd_lock
  - rd_unlock
Shared/Exclusive Locks

Legend
- Active thread
- Sleeping thread

Step 3

Step 4
Shared/Exclusive Locks

Legend

Active thread

Sleeping thread

Step 5

Step 6
Shared/Exclusive Locks

Legend

Active thread

Sleeping thread

Writer 1  

...  

rw_lock  

...  

rw_unlock  

...  

Writer 2  

...  

rw_lock  

...  

rw_unlock  

...  

Reader 1  

...  

rd_lock  

...  

rd_unlock  

...  

Reader 2  

...  

rd_lock  

...  

rd_unlock  

...  

Step 7
Condition Variable

- Block a thread while waiting for a condition
- Condition_wait / condition_signal
- Several thread can wait for the same condition, they all get the signal
Condition Variable

• Block a thread while waiting for a condition
• Condition_wait / condition_signal
• Several thread can wait for the same condition, they all get the signal
Condition Variable

- Block a thread while waiting for a condition
- `Condition_wait` / `condition_signal`
- Several threads can wait for the same condition, they all get the signal
Condition Variable

• Block a thread while waiting for a condition
• Condition_wait / condition_signal
• Several thread can wait for the same condition, they all get the signal
Condition Variable

- Block a thread while waiting for a condition
- `Condition_wait / condition_signal`
- Several threads can wait for the same condition, they all get the signal
Semaphores

- simple counting mutexes
- The semaphore can be held by as many threads as the initial value of the semaphore.
- When a thread gets the semaphore, it decreases the internal value by 1.
- When a thread releases the semaphore, it increases the internal value by 1.
Semaphores

Thread 1
... get
release...

Thread 2
... get
release...

Thread 3
... get
release...

Semaphore (2)

Semaphore (1)
Semaphores

Thread 1
...
get
release
...

Thread 2
...
get
release
...

Thread 3
...
get
release
...

Semaphore (0)

Semaphore (0)
Semaphores

Thread 1
...
get
release
...

Thread 2
...
get
release
...

Thread 3
...
get
release
...

Semaphore (1)

Semaphore (1)
Semaphores

Thread 1

... get
release ...

Thread 2

... get
release ...

Thread 3

... get
release ...

Semaphore (1)

Thread 1

... get
release ...

Thread 2

... get
release ...

Thread 3

... get
release ...

Semaphore (2)
Atomic instruction

- Is any operation that a CPU can perform such that all results will be made visible to each CPU at the same time and whose operation is safe from interference by other CPUs
  - TestAndSet
  - CompareAndSwap
  - DoubleCompareAndSwap
  - Atomic increment
  - Atomic decrement
Example: A Producer – Consumer Queue

- We have a bounded queue where producers store their output and from where consumers take their input
- Protect the structure against intensive unnecessary accesses
  - Detect boundary conditions: queue empty and queue full
Example: dot-product

Scalar product, inner product

In mathematics, the dot product or scalar product (sometimes inner product in the context of Euclidean space, or rarely projection product for emphasizing the geometric significance), is an algebraic operation that takes two equal-length sequences of numbers (usually coordinate vectors) and returns a single number.

Dot product - Wikipedia, the free encyclopedia

Scalar product, inner product

\[ \sum_{n=1}^{N} a_i \cdot b_i \]

- Divide the arrays between participants to load-balance the work
  - Each will then compute a partial sum
- Add all the partial sums together for the final result (reduce operation)
- Technical details: cost of managing the threads vs. cost of the algorithm? How to minimize the management cost?