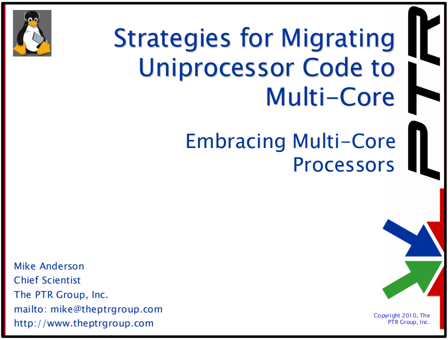
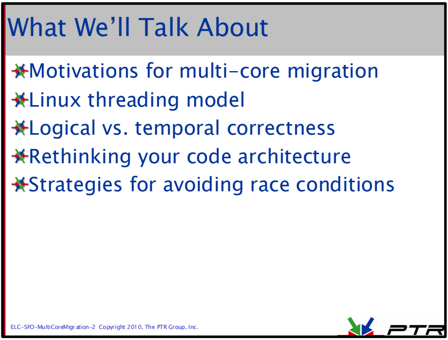


Strategies for Migrating Uniprocessor Code to Multi-Core

Embracing Multi-Core Processors

Mike Anderson
Chief Scientist
The PTR Group, Inc.
mailto: mike@theptrgroup.com
http://www.theptrgroup.com



What We’ll Talk About

- Motivations for multi-core migration
- Linux threading model
- Logical vs. temporal correctness
- Rethinking your code architecture
- Strategies for avoiding race conditions

What we won’t be Addressing

- The focus of this discussion is at the process/thread level
- We won’t be addressing:
  - Instruction-level parallelism (ILP)
  - OpenMP
  - Out-of-order, super-scalar processor issues and memory barriers
  - Simultaneous Multi-Threading (SMT)
  - SIMD instruction sets
- Each of these are worthy topics on their own, but I only have so much time…

Source: files.myopera.com

Why Multi-Core?

- The motivations for multi-core seem clear at this point in time
  - Lower thermal envelope
  - Lower power consumption
  - Ability to scale our code across multiple execution units
- However, there are “gotchas” as well
  - Each core is clocked slower
  - Cache misses and process migration issues can slow code execution

Source: weblog.sinteur.com

Single vs. Multi-Threaded Applications

- Much of the existing code today is single threaded
  - Only one execution path
- Single-threaded applications cannot utilize the additional cores
  - Lower frequencies of the cores means lower performance of the single-threaded application
    - Intel's "TurboBoost" is addressing this
- Multi-threaded code has multiple, simultaneous execution paths
  - Multi-threaded code often relies on priorities to ensure proper execution
    - Highest priority always wins in the scheduler

Scalability of Algorithms

- If an algorithm is perfectly scalable then adding N processors increases the speed N times
- This is represented in Amdahl's Law:

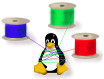
$$S_p = T_1 / T_p$$

where S is the speed up, T is the time to execute an algorithm and p is the number of processors

- Unfortunately, most code is rarely perfectly scalable due to IPCs, synchronization primitives and bus contention

The Linux Threading Model

- Linux supports a number of different threading models
  - GNU Pth, NPTL, SolarisThreads and more
- Most popular is NPTL
  - POSIX-based, 1-1 scheduling
- Each thread is independently schedulable
  - Blocking in one thread had no impact on other threads
- All share the address space of their parent process
  - I.e., memory is “flat” between threads



Source: acm.org

The Scheduler

- The scheduler runs on each core
  - Selects the highest priority thread ready to run at that time and dispatches it
- E.g.,. on a UP, priority 50 thread will run to completion before priority 0 thread
  - No problems with contention
- On a MP, priority 50 thread will run on one core while priority 0 thread runs simultaneously on the other
  - Race conditions will manifest themselves

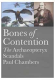
What is a Race Condition?

- When a program does the right set of steps, it’s considered to be logically correct
- When it does the right thing at the right time, it’s temporally correct
- Race conditions are violations of temporal correctness
  - Also known as “live-lock”

Source: dev.esl.eu

Where is the Contention?

- Most race conditions are caused due to contention over data structures or resources
  - Multiple threads accessing the same data at the same time from multiple cores
- Problem doesn’t manifest on a UP
  - Priority preemption prevents it
- Implies that there is a critical region of code that must have exclusive access for some period of time
  - Identifying the critical region takes practice



Source: wordpress.com

Detecting Race Conditions

- How could we go about detecting race conditions?
  - Static detection performed at compile time
    - Static detection is an NP-hard problem
      - Like the traveling salesman's problem
  - Heuristic detection techniques
    - Heuristic techniques can only detect potential race conditions
  - Dynamic detection at run time
    - We need to examine every memory access
      - We can only detect it after it happens
- All this being said, there are companies that sell automated tools that claim race-detection capabilities
  - Klocwork Insight™ and Coverity Prevent™ among others
  - YMMV

Source: informatik.uni-leipzip.de

Techniques for Avoiding Races #1

- Since most race conditions arise over contention for global data, simply eliminate the global data
- The stacks for each thread are unique
  - Store the data on the local stack
- Linux supports the use of thread local storage (TLS)
  - The `pthread_key_create(…)` and `pthread_getspecific(…)` calls allow for storage known only to the local thread
- Unfortunately, these approaches may require that algorithms be significantly re-written

Source: ehow.com

Techniques for Avoiding Races #2

- Contention can arise from threads on separate cores
  - Lock all of the threads to a single core
    - This reduces to the UP solution
  - Known as the “containment” approach
- This requires the use of processor affinity assignments
  - Also requires the use of priorities to ensure proper operation

Problems with Containment

- First, locking all threads to a single processor core defeats the scalability of MC systems
  - The reason you went to MC in the first place
- The requirement to use priorities is subtle
  - Time slicing can force preemption leaving the resource in an unknown state
  - Not a problem in preemptive, priority-based O/Ses like many RTOS solutions
  - Failure mode may not manifest itself frequently

Source: pet.imageg.net

A Brief Aside: Processor Affinity

- In Linux, the O(1) and CFS schedulers actually try to keep threads on the same processor when possible
  - Called “soft affinity”
  - Can conflict with load-balancing goals
- Even with soft affinity, threads can still migrate
- We can see the current core assignment for any thread in the **`ps`** command
  - Also visible in the /proc file system entry for the PID

Setting Hard Affinity

- In order for us to prevent thread migration, we must use hard affinity settings
  - We need to make sure that we have the schedutils package installed
- This allows us to use the **`taskset`** command to control a CPU migration mask for the PID
  - **`taskset -p [mask] pid`**
- We have a “1” bit in every allowed CPU core

Setting Hard Affinity in Code

Source: allfreightfree.com

- We can also set the affinity mask in our code
  - The `sched_setaffinity(…)` call allows us to set the processor the mask on a process basis
    - Does not include any threads
  - `pthread_setaffinity_np(…)` allows us to set the processor mask for pthreads
  - There are `sched_getaffinity(…)` and `pthread_getaffinity_np(…)` calls to retrieve the mask
- These calls also have an equivalent for kernel threads

Example Code

```
cpu_set_t cmask;
unsigned long len = sizeof(cmask);
pid_t p = 0;

CPU_ZERO(&cmask);
CPU_SET(0, &cmask);

if (!sched_setaffinity(0, len, &cmask)) {
  perror("Could not set cpu affinity for current process.\n");
}
```

- This would set the affinity for the calling process to core 0
- The mask allows for multiple CPUs to be set in the mask
  - E.g., a group of user-code cores and a group of interrupt cores

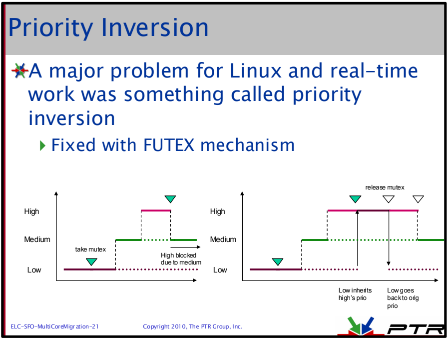
What About Encapsulation?

- You could place the resource in a class with access methods
  - Unless there is an kernel-enforced synchronization primitive involved, this is no better than containment
    - Time slicing can still leave resource in an unknown state
- You need to wrap access to the resource in a mutual exclusion mechanism

Source: nextmedia.com

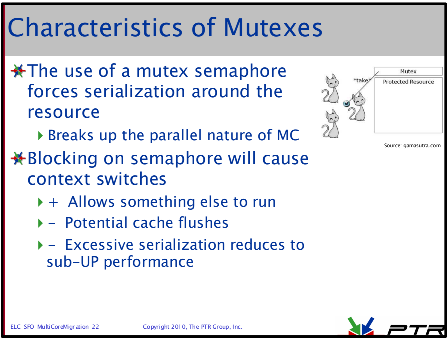
Mutual Exclusion Mechanisms #1

- The most common mutual exclusion technique is to use mutual exclusion (mutex) semaphores
  - Each code segment must acquire the semaphore before access
    - Release the semaphore after use
- Linux mutexes, via pthread calls, are based on the Linux fast, user-space mutex (FUTEX) mechanism
  - Adaptive in nature
    - Doesn't immediately sleep
  - If no contention, does not require kernel intervention
  - Priority inversion support
  - Has concept of ownership



Priority Inversion

- A major problem for Linux and real-time work was something called priority inversion
  - Fixed with FUTEX mechanism



Characteristics of Mutexes

- The use of a mutex semaphore forces serialization around the resource
  - Breaks up the parallel nature of MC
- Blocking on semaphore will cause context switches
  - \+ Allows something else to run
  - \- Potential cache flushes
  - \- Excessive serialization reduces to sub-UP performance

Source: gamasutra.com

Mutual Exclusion Mechanisms #2

- The Pthreads API also supports spin locks
    - A spin lock is a tight loop that checks for availability of the lock
- Burns CPU time
- Used in cases where context switch is undesirable
    - You expect that the resource will become available “soon”
- Might produce better performance on certain MC applications

Mutual Exclusion Mechanisms #3

- Another technique is to use message queues to pass data between threads
    - Decouples the production rate from the consumption rate
        - Threads become more “asynchronous”
- Unfortunately, requires multiple copies
    - One into the queue, one out for each direction
- Can pass pointers to data via the message queue to reduce copy overhead

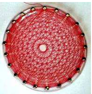
Beware of Binary Semaphores

- You might be tempted to use a traditional binary semaphore
  - It seems like it might work
- But, binary semaphores are subject to priority inversion
- Also, binary semaphores do not have a concept of ownership
  - Recursive calls to the `sem_wait()` function will cause deadlock
- Binary semaphores are designed for synchronization rather than mutual exclusion

Source: flickr.com

Threading Design Guidelines

- When developing applications, try to identify those activities that can run in parallel
- Identify data flow through the application
  - Determine what data must be shared between activities
- Identify the correct sequencing of the activities
  - Temporal correctness
- Identify relative importance of activities
  - These may need priority adjustments



Source: mathcats.com

Thread Design Guidelines #2

- Don’t assume that priorities will preclude race conditions
  - Remember, lower priority thread can run on other core!
- When designing your threads, keep them as separate as possible
  - Don’t share data unless necessary
  - Use synchronization primitives when needed
    - Mutexes, spin locks, message queues, etc.
- Try to keep data used by threads on separate cache lines
  - Create a cache_aligned_malloc/cache_aligned_free to make sure data is in separate cache lines to avoid false sharing
    - Avoid ping-ponging between processor caches

Summary

- Effective use of MC processors will require some thought on your part
  - You might need significant re-architecting to make your application MC aware
- Focus on data flow and identify critical regions of code
  - Try to keep the critical regions as short as possible to avoid excessive serialization
- Address processor affinity if you need to optimize the code to the next level