User–Space, Multi–Core Development Issues

What do we do with all of these processors?

Mike Anderson
Chief Scientist
The PTR Group, Inc.
http://www.theptrgroup.com
What We Will Talk About

- Motivations for multi-core processors
- Scaling issues
- Linux support for multi-processing
- Designing software for multi-processing
- Demo
Multi-Core Motivations

Unless you’ve been living under a rock, you’ve heard about the multi-core revolution

- Clock speeds couldn’t scale indefinitely
  - Power usage varies with the square of the voltage

2–16 Cores are now generally available for most of the popular CPU architectures

- Power, ARM, x86, MIPS, etc.

Both homogeneous and heterogeneous multi-core systems are available
Dual-Core PowerPC from Freescale

- MPC8641D
- Dual E600 cores can run SMP or detached mode
- MPX bus keeps the processor’s caches coherent

Source: Freescale Semiconductor
Quad-Core MPCore ARM–11

- Quad ARM–11 processors
- Specialized interrupt distribution for routing and interrupt balancing
- Bus snooping to improve cache coherency

Source: ARM Ltd
Dual-Core MIPS from Broadcom

- Dual MIPS-64 with Quad-issue, in-order pipeline
- 600-800 MHz cores
- Power dissipation of 8-10W @ 800 MHz

Source: Broadcom Corporation
Intel™ Nehalem™ and Atom™

- The i7 architecture starts with 4 cores and scales to 12
- Shared L3 cache to help mitigate code migration and data sharing effects
- The dual-core Atom 330 is also shipping
  - Supports hyperthreading as well
Embedded Heterogeneous MCP

- The TI OMAP is a good example of a heterogeneous multi-core processor
  - ARM and DSP processors on die
- The use in cell phones and reference boards like the Beagleboard show heterogeneous MCPs can meet varying embedded requirements
Special-Purpose Heterogeneous MCPs

The IBM Cell processor is another example of a heterogeneous multi-core processor

- Built for the PS/3 game console

But, it makes an excellent RADAR processing engine

- High-performance computing engine
AMP vs. SMP

Asymmetric Multi-Processing has been around for decades

- Separate CPUs with separate O/S tied together with LANs
- Message-passing programming paradigm

Symmetric Multi-Processing dates back to 1964

- One O/S to bind them all
Characteristics of SMP/MCP Machines

- All processors see everything
  - Memory, I/O, interrupts, etc.
- There is only one kernel
  - The scheduler determines which applications are assigned to which processor
- Applications can migrate between processors
- They do not typically share caches
  - This model changes when we shrink SMP to the chip level for MCPs
Advantages of SMP/MCPs

- Given multiple processors, applications can each run on their own processor
  - This can be coupled with Simultaneous Multi-Threaded (SMT or hyperthreading) as well
    - 2 processors look like 4 processors
- This tends to favor applications that have threads that are independently schedulable
  - I.e., the 1–1 threading model found in Linux
- Interrupt latency is minimized
  - The interrupt runs on any free processor
- Processor cores can be partitioned
  - Alternate O/S can be run on other cores
  - Hypervisors can help in offloading work
Problems with SMP/MCPs

- SMP/MCPs do not scale perfectly
  - Because the memory is shared between CPUs and the memory has a finite bandwidth, SMP/MCP machines can develop “hot spots” where multiple applications must serialize on a single piece of data.

- Thread/ISR migration can lead to poor cache utilization
  - We need to flush the caches if a thread or ISR migrates.

- Multiple processors can lead to race conditions
  - We need to provide for multi-processor synchronization.
Multi-Core Performance Issues

Assuming a shared bus architecture:

- Dual core runs at about 180% of single core of same speed
  - Quad core runs 50% faster than the dual
    - 270% faster than the single core

Remember, multi-core is typically clocked slower than a single core

- Lower heat production and power consumption
- But, poorer performance for single-threaded applications
Multi-Processor Use Cases

SMP

- O/S manages applications transparently
- Good for control plane
- Bus bandwidth a limit for data plane

Partitioning

- Typically AMP
- Frequently implemented via light-weight executives or hypervisors
- Works for both control & data plane
- Partitioned processors can run alternate O/S or thin layers
- Partitioned processors are data shufflers
- Data plane cores can be simpler and cheaper
  - But, deep packet inspection suffers if they’re too simple
Multi-Processing Use Cases #2

Offloading

- CPU-intensive work is sent to alternate core(s) with thin executive
- Used in deep packet inspection and security applications

Standby

- Idle cores are held in reserve for redundancy
- Supports adding more capacity in the field via software
- Load updates to idle core and switch
- Rapid S/W upgrade with little downtime
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:

\[ S_p = \frac{T_1}{T_p} \]

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
Processor Affinity

- The term processor affinity relates to the tendency for an application to run on a particular processor and resist migration

- The scheduler will prefer not to migrate a process to another CPU unless needed
  - This is referred to as soft affinity
  - This can be overridden with hard affinity assignments in source code

- Hard affinity APIs allow the developer to make explicit assignments to a processor or a group of processors
  - You decide where your code runs by setting a CPU bit mask for each thread via calls like Linux’s `sched_setaffinity()` and `sched_getaffinity()`
The Kernel’s Knowledge of SMP

- In both SMP and SMT, the kernel needs to be compiled with SMP enabled
  - This informs various subsystems of the presence of multiple processors
- The Linux scheduler supports process affinity
  - A means of assigning threads to particular processors to avoid cache flushes
- IRQ load balancing subsystem allows interrupt lines to be directed to particular processors
  - Fortunately, Linux also supports IRQ affinity to ensure fast ISR response
Example of Interrupt Load Balancing

```
mike@defiant:~> more /proc/interrupts

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 21427467</td>
<td>21403917</td>
<td>IO-APIC-edge</td>
</tr>
<tr>
<td>1: 13217</td>
<td>14317</td>
<td>IO-APIC-edge</td>
</tr>
<tr>
<td>8: 5</td>
<td>0</td>
<td>IO-APIC-edge</td>
</tr>
<tr>
<td>9: 2</td>
<td>0</td>
<td>IO-APIC-fasteoi</td>
</tr>
<tr>
<td>12: 17306</td>
<td>23786</td>
<td>IO-APIC-edge</td>
</tr>
<tr>
<td>14: 205456</td>
<td>206781</td>
<td>IO-APIC-edge</td>
</tr>
<tr>
<td>15: 158807</td>
<td>158750</td>
<td>IO-APIC-edge</td>
</tr>
<tr>
<td>16: 2291422</td>
<td>2290748</td>
<td>IO-APIC-fasteoi</td>
</tr>
<tr>
<td>17: 405909</td>
<td>400991</td>
<td>IO-APIC-fasteoi</td>
</tr>
<tr>
<td>18: 3</td>
<td>0</td>
<td>IO-APIC-fasteoi</td>
</tr>
<tr>
<td>19: 168320</td>
<td>164332</td>
<td>IO-APIC-fasteoi</td>
</tr>
<tr>
<td>20: 2174764</td>
<td>2176161</td>
<td>IO-APIC-fasteoi</td>
</tr>
<tr>
<td>21: 0</td>
<td>0</td>
<td>IO-APIC-fasteoi</td>
</tr>
<tr>
<td>22: 0</td>
<td>0</td>
<td>IO-APIC-fasteoi</td>
</tr>
<tr>
<td>23: 0</td>
<td>0</td>
<td>IO-APIC-fasteoi</td>
</tr>
<tr>
<td>NMI: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC: 42831216</td>
<td>42830668</td>
<td></td>
</tr>
<tr>
<td>ERR: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIS: 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interrupt Balancing: timer
Interrupt Affinity: i8042, rtc, acpi, nvidia, ipw3945, eth0, uhci_hcd:usb1, ehci_hcd:usb5, uhci_hcd:usb2, HDA Intel, uhci_hcd:usb3, uhci_hcd:usb4, sdhci:slot0
```
Threads and Processes

The classic process model has a single thread of control with a dedicated virtual memory address (VMA) space.

If we allow for more than one thread of control in a single VMA, we have a multi-threaded process.

- A key factor is how the scheduler treats these different threads of control.

Linux works like many of the RTOSes with respect to the scheduler.

- Each thread is independently schedulable.
Threading Example

A reasonable example of processes vs. threads would be an application like MS Word

- Word is the process that anchors the VMA

- Word is comprised of multiple threads
  - Repagination
  - Background printing
  - WYSIWYG formatting
  - Spell checking
  - Popping up that annoying paper clip thingy
  - And more...

Source: Microsoft
Many developers are intimidated by threading in their applications
- They are not quite sure what a thread is
- O/S APIs can be difficult to understand

Essentially, if you can think of a piece of code a separate sequence of steps from the main, then its probably a candidate to be a thread
- A thread can be thought of as a subroutine with a life of its own
Fine-Grained Threading via OpenMP

- Open standard focused on extending compilers to support fine-grained parallelism via threading
  - Goal is high-performance by splitting up algorithms and running them as parallel threads
- Targeted at simultaneous multi-threading (SMT a.k.a. hyperthreaded) and multi-core CPUs
  - Compiler is responsible for creating parallel threads
  - Compilers require hints from the developer for what to parallelize

http://www.openmp.org
OpenMP Usage

To use OpenMP, you may need to restructure your code:

```c
for (j=0; j < num_elements; j++) {
    my_array[j] = startval;
    startval++;
}
```

This loop cannot be parallelized because of the data dependency on `startval`.

- We need to rewrite the code like this:

```c
#pragma omp parallel for
for (j=0; j < num_elements; j++) {
    my_array[j] = startval + j;
}
startval += num_elements;
```
Programming for OpenMP

- OpenMP is only supported by certain compilers
  - E.g., Intel compilers for C/C++ and FORTRAN
  - GNU gcc 4.2.1+
- Requires the use of various \#pragma directives to provide hints for the compiler
  - You need to know where they might apply
- May require you to recode your program to make it more parallelizable
Stepping up a Level – pThreads

Of all of the threading APIs, the POSIX pThreads API has arguably the largest number of implementations

- A non-proprietary API that can be implemented in virtually any O/S

The threads all live in the global address space of the parent process VMA

- Threads can each have their own priority
  - Different scheduling policies are also supported

However, pThreads have a reputation for being difficult to understand
pThread Example #1 of 3

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

int global;

void * thread(void *joiner) {
    void *status;
    global = pthread_self();
    sleep(1);
    printf("Parent PID is %d, TID is %d, global = %d\n",
           getpid(), pthread_self(), global);
    if (joiner) {
        if (pthread_join((pthread_t)joiner, &status)) {
            exit(1);
        }
    }
    pthread_exit((void*) 0);
}
int main(void) {
    void *status;
    int x;
    pthread_attr_t attr;
    pthread_t curr_thr_id;
    pthread_t prev_thr_id;

    pthread_attr_init(&attr);
    if (pthread_attr_setschedpolicy(&attr, SCHED_RR)) {
        exit(1);
    }

    /* Start 3 threads */
    prev_thr_id = 0;
    for (x=0; x<3; x++) {
        if (pthread_create(&curr_thr_id, &attr, thread, (void*)prev_thr_id)) {
            exit(1);
        }
        prev_thr_id = curr_thr_id;
    }
}
pThread Example #3 of 3

/* Join last thread */
   pthread_join(curr_thr_id, &status);
}

This example shows the same piece of code being used to create three different threads:

- Each thread is independent, but shares the VMA of main
- Each could have its own priority and processor affinity assigned
- In a 1:1 threading model, each would be independently schedulable
Reentrancy and Synchronization

- Thread APIs like POSIX support semaphores, mutexes, message queues, spin locks and a host of other IPC mechanisms
  - Due to the flat address space within the VMA, critical sections need to be protected to avoid reentrancy issues
- If a resource is shared, it *must* be protected
- Use of semaphores can enforce ordering of threads
  - Blocking one thread does not block all threads in the same process in 1:1 thread models
Simplifying Writing Thread Code

Most threading APIs, although fairly straightforward, have been wrapped in class libraries

- C++, Java, Python, Ruby, etc.

Some, like Intel’s Thread Building Blocks are open source and run in multiple O/Ses

Migrating to Multi-Core

If your applications is single-threaded, simply recompile for the platform and run

- Don’t be surprised if the performance actually drops from that of a single core due to clock-speed issues

If the application is multi-threaded, try a containment approach first

- Use affinity settings to lock the threads to a single core
- Then start enhancing with mutual exclusion to enable threads running on multiple cores
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
Thread Design Guidelines #2

- Don’t assume that priorities will preclude race conditions
  - 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
    • Semaphores, mutexes, 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
Thread Affinity Guidelines

- If your hardware is SMP/Multi-Core, run the application without adjusting the affinity to see if there is a problem
  - Don’t try to solve a problem if it doesn’t exist
- If there is an issue, look at processor loading to see if one processor is bearing most of the effort
  - If yes, then adjusting affinity comes next
Summary

- Multi-core can be thought of as SMP on a chip
- Make sure you understand and use affinity mechanisms
  - Provides the most flexibility
  - Use SCHED_FIFO/SCHED_RR and priorities when needed
  - Don’t forget interrupt affinity as well
- We must consider application redesign to take advantage of multi-core processors
  - The use of threads becomes important
  - POSIX pThreads API
    - Good documentation, good place to start