Linux and symmetric multiprocessing

Unlock the power of Linux on SMP systems

M. Tim Jones

March 14, 2007

As evidenced by major central processing unit (CPU) vendors, multi-core processors are poised to dominate the desktop and embedded space. With multiprocessing comes greater performance but also new problems. This article explores the ideas behind multiprocessing and developing applications for Linux® that exploit SMP.

You can increase the performance of a Linux system in various ways, and one of the most popular methods is increasing the performance of the processor. An obvious solution is to use a processor with a faster clock rate, but for any given technology there exists a physical limit where the clock simply can't go any faster. When you reach that limit, you can use the more-is-better approach and apply multiple processors. Unfortunately, performance doesn't scale linearly with the aggregate performance of the individual processors.

Before discussing the application of multiprocessing in Linux, let's take a quick look back at the history of multiprocessing.

History of multiprocessing

Flynn's classification of multi-CPU architectures

Single Instruction, Single Data (SISD) is the typical uniprocessor architecture. The *Multiple Instruction, Multiple Data (MIMD)* multiprocessing architecture has separate processors operating on independent data (control parallelism). Finally, *Single Instruction, Multiple Data (SIMD)* has a number of processors operating on different data (data parallelism).

See the Related topics section below for detail on Flynn's original paper.

Multiprocessing originated in the mid-1950s at a number of companies, some you know and some you might not remember (IBM, Digital Equipment Corporation, Control Data Corporation). In the early 1960s, Burroughs Corporation introduced a symmetrical MIMD multiprocessor with four CPUs and up to sixteen memory modules connected via a crossbar switch (the first SMP architecture). The popular and successful CDC 6600 was introduced in 1964 and provided a CPU with ten subprocessors (peripheral processing units). In the late 1960s, Honeywell delivered the first Multics system, another symmetrical multiprocessing system of eight CPUs.

While multiprocessing systems were being developed, technologies also advanced the ability to shrink the processors and operate at much higher clock rates. In the 1980s, companies like Cray Research introduced multiprocessor systems and UNIX®-like operating systems that could take advantage of them (CX-OS).

The late 1980s, with the popularity of uniprocessor personal computer systems such as the IBM PC, saw a decline in multiprocessing systems. But now, twenty years later, multiprocessing has returned to these same personal computer systems through symmetric multiprocessing.

Amdahl's law

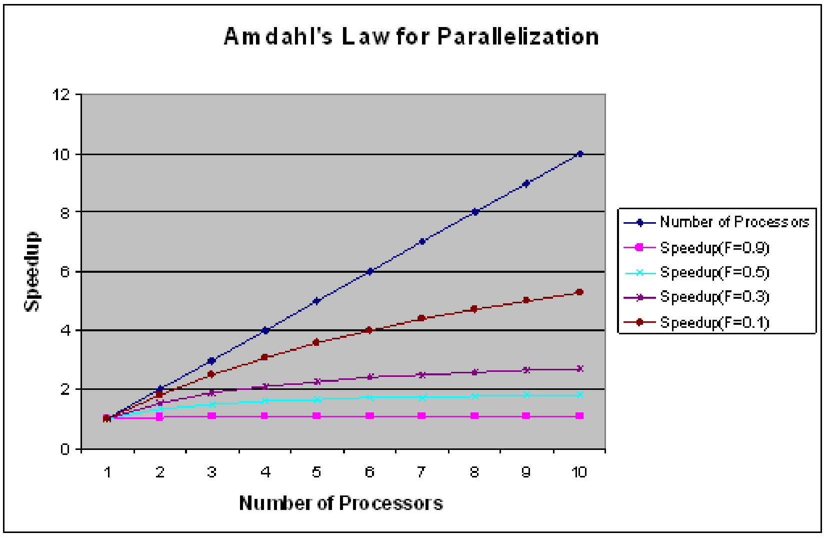
Gene Amdahl, a computer architect and IBM fellow, developed computer architectures at IBM, his namesake venture, Amdahl Corporation, and others. But he is most famous for his law that predicts the maximum expected system improvement when a portion of the system is improved. This is used predominantly to calculate the maximum theoretical performance improvement when using multiple processors (see Figure 1).

Figure 1. Amdahl's law for processor parallelization

$$Speedup = \frac{1}{F + (1 - F) / N}$$

Using the equation shown in Figure 1, you can calculate the maximum performance improvement of a system using *N* processors and a factor *F* that specifies the portion of the system that cannot be parallelized (the portion of the system that is sequential in nature). The result is shown in Figure 2.

Figure 2. Amdahl's law for up to ten CPUs



In Figure 2, the top line shows the number of processors. Ideally, this is what you'd like to see when you add additional processors to solve a problem. Unfortunately, because not all of the problem can be parallelized and there's overhead in managing the processors, the speedup is quite a bit less. At the bottom (purple line) is the case of a problem that is 90% sequential. In the best case for this graph, the brown line shows a problem that's 10% sequential and, therefore, 90% parallelizable. Even in this case, ten processors perform only slightly better than five.

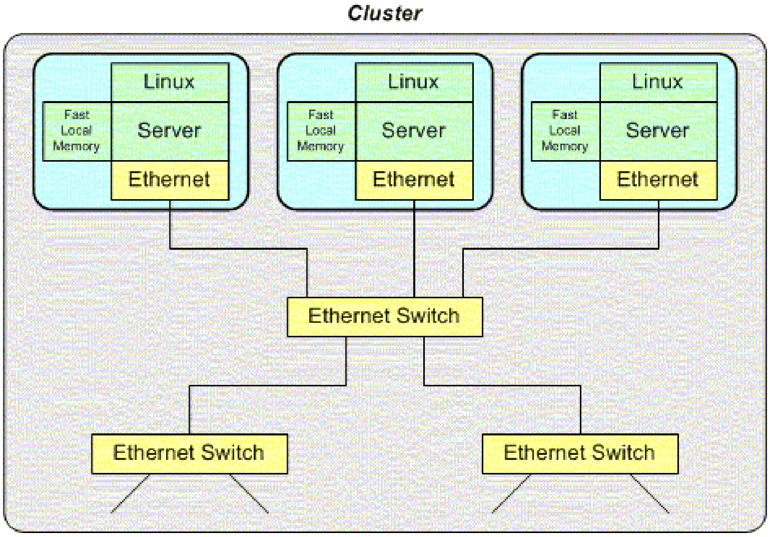
Multiprocessing and the PC

An SMP architecture is simply one where two or more identical processors connect to one another through a shared memory. Each processor has equal access to the shared memory (the same access latency to the memory space). Contrast this with the Non-Uniform Memory Access (NUMA) architecture. For example, each processor has its own memory but also access to shared memory with a different access latency.

Loosely-coupled multiprocessing

The earliest Linux SMP systems were loosely-coupled multiprocessor systems. These are constructed from multiple standalone systems connected by a high-speed interconnect (such as 10G Ethernet, Fibre Channel, or Infiniband). This type of architecture is also called a cluster (see Figure 3), for which the Linux Beowulf project remains a popular solution. Linux Beowulf clusters can be built from commodity hardware and a typical networking interconnect such as Ethernet.

Figure 3. Loosely-coupled multiprocessing architecture



Building loosely-coupled multiprocessor architectures is easy (thanks to projects like Beowulf), but they have their limitations. Building a large multiprocessor network can take considerable space and power. As they're commonly built from commodity hardware, they include hardware that isn't relevant but consumes power and space. The bigger drawback is the communications fabric. Even with high-speed networks such as 10G Ethernet, there are limits to the scalability of the system.

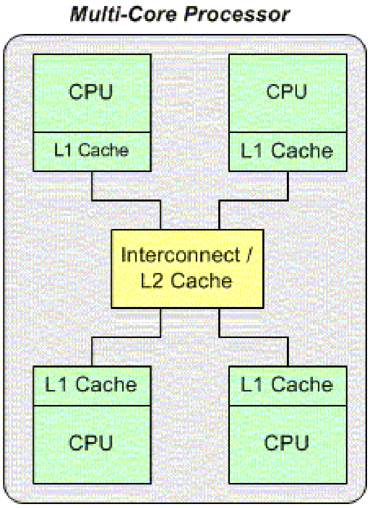
Tightly-coupled multiprocessing

Tightly-coupled multiprocessing refers to chip-level multiprocessing (CMP). Think about the loosely-coupled architecture being scaled down to the chip level. That's the idea behind tightly-coupled multiprocessing (also called multi-core computing). On a single integrated circuit, multiple chips, shared memory, and an interconnect form a tightly integrated core for multiprocessing (see Figure 4).

Processor interconnects

Another interconnect option (bus for a system fabric) is AMD's HyperTransport. Intel® is also planning a new interconnect called the Common System Interface, which is expected in 2008.

Figure 4. Tightly-coupled multiprocessing architecture



In a CMP, multiple CPUs are connected via a shared bus to a shared memory (level 2 cache). Each processor also has its own fast memory (a level 1 cache). The tightly-coupled nature of the CMP allows very short physical distances between processors and memory and, therefore, minimal memory access latency and higher performance. This type of architecture works well in multi-threaded applications where threads can be distributed across the processors to operate in parallel. This is known as thread-level parallelism (TLP).

Given the popularity of this multiprocessor architecture, many vendors produce CMP devices. Table 1 lists some of the popular variants with Linux support.

Table 1. Sampling of CMP devices

Vendor	Device	Description
IBM	POWER4	SMP, dual CPU
IBM	POWER5	SMP, dual CPU, four simultaneous threads
AMD	AMD X2	SMP, dual CPU
Intel®	Xeon	SMP, dual or quad CPU
Intel	Core2 Duo	SMP, dual CPU
ARM	MPCore	SMP, up to four CPUs
IBM	Xenon	SMP, three Power PC CPUs
IBM	Cell Processor	Asymmetric multiprocessing (ASMP), nine CPUs

Kernel configuration

To make use of SMP with Linux on SMP-capable hardware, the kernel must be properly configured. The `CONFIG_SMP` option must be enabled during kernel configuration to make the kernel SMP aware. With an SMP-aware kernel running on a multi-CPU host, you can identify the number of processors and their type using the proc filesystem.

First, you retrieve the number of processors from the cpuinfo file in /proc using `grep`. As shown in Listing 1, you use the count option (`-c`) for lines that begin with the word `processor`. The content of the `cpuinfo` file is then presented. The example shown is from a two-chip Xeon motherboard.

Listing 1. Using the proc filesystem to retrieve CPU information

```
mtj@camus:~$ grep -c ^processor /proc/cpuinfo
8
mtj@camus:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 6
model name      : Intel(R) Xeon(TM) CPU 3.73GHz
stepping        : 4
cpu MHz         : 3724.219
cache size      : 2048 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 2
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 6
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
pbe nx lm pni monitor ds_cpl est cid xtpr

bogomips        : 7389.18

...

processor       : 7
vendor_id       : GenuineIntel
cpu family      : 15
model           : 6
model name      : Intel(R) Xeon(TM) CPU 3.73GHz
stepping        : 4
cpu MHz         : 3724.219
cache size      : 2048 KB
physical id     : 1
siblings        : 4
core id         : 3
cpu cores       : 2
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 6
```

```
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
pbe nx lm pni monitor ds_cpl est cid xtpr

bogomips        : 7438.33

mtj@camus:~$
```

SMP and the Linux kernel

In the early days of Linux 2.0, SMP support consisted of a "big lock" that serialized access across the system. Advances for support of SMP slowly migrated in, but it wasn't until the 2.6 kernel that the power of SMP was finally revealed.

The 2.6 kernel introduced the new O(1) scheduler that included better support for SMP systems. The key was the ability to load balance work across the available CPUs while maintaining some affinity for cache efficiency. For cache efficiency, recall from Figure 4 that when a task is associated with a single CPU, moving it to another CPU requires the cache to be flushed for the task. This increases the latency of the task's memory access until its data is in the cache of the new CPU.

SMP in the kernel

To understand how SMP is initialized for a given architecture, check out the smp.c or smpboot.c files within the kernel at ./linux/arch/<arch>/kernel/ (for most architectures and platforms).

The 2.6 kernel maintains a pair of runqueues for each processor (the expired and active runqueues). Each runqueue supports 140 priorities, with the top 100 used for real-time tasks, and the bottom 40 for user tasks. Tasks are given time slices for execution, and when they use their allocation of time slice, they're moved from the active runqueue to the expired runqueue. This provides fair access for all tasks to the CPU (and locking only on a per CPU basis).

With a task queue per CPU, work can be balanced given the measured load of all CPUs in the system. Every 200 milliseconds, the scheduler performs load balancing to redistribute the task loading to maintain a balance across the processor complex. For more information on the Linux 2.6 scheduler, see the Related topics section.

User space threads: Exploiting the power of SMP

A lot of great work has gone into the Linux kernel to exploit SMP, but the operating system by itself is not enough. Recall that the power of SMP lies in TLP. Single monolithic (non-threaded) programs can't exploit SMP, but SMP can be exploited in programs that are composed of many threads that can be distributed across the cores. While one thread is delayed awaiting completion of an I/O, another thread is able to do useful work. In this way, the threads work with one another to hide each other's latency.

Portable Operating System Interface (POSIX) threads are a great way to build threaded applications that are able to take advantage of SMP. POSIX threads provide the threading mechanism as well as shared memory. When a program is invoked that creates some number of threads, each thread is provided its own stack (local variables and state) but shares the data space of the parent. All threads created share this same data space, but this is where the problem lies.

To support multi-threaded access to shared memory, coordination mechanisms are necessary. POSIX provides the mutex function to create *critical sections* that enforce exclusive access to an object (a piece of memory) by a single thread. Not doing so can lead to corrupted memory due to unsynchronized manipulation by multiple threads. Listing 2 illustrates creating a critical section with a POSIX mutex.

Listing 2. Using pthread_mutex_lock and unlock to create critical sections

```
pthread_mutex_t crit_section_mutex = PTHREAD_MUTEX_INITIALIZER;

...

pthread_mutex_lock( &crit_section_mutex );

/* Inside the critical section. Memory access is safe here
 * for the memory protected by the crit_section_mutex.
 */

pthread_mutex_unlock( &crit_section_mutex );
```

If multiple threads attempt to lock a semaphore after the initial call above, they block and their requests are queued until the `pthread_mutex_unlock` call is performed.

Kernel variable protection for SMP

When multiple cores in a processor work concurrently for the kernel, it's desirable to avoid sharing data that's specific to a given core. For this reason, the 2.6 kernel introduced the concept of *per-CPU* variables that are associated with a single CPU. This permits the declaration of variables for a CPU that are most commonly accessed by that CPU, which minimizes the locking requirements and improves performance.

Defining a per-CPU variable is done with the `DEFINE_PER_CPU` macro, to which you provide a type and variable name. Since the macro behaves like an l-value, you can also initialize it there. The following example (from ./arch/i386/kernel/smpboot.c) defines a variable to represent the state for each CPU in the system.

```
/* State of each CPU. */
DEFINE_PER_CPU(int, cpu_state) = { 0 };
```

The macro creates an array of variables, one per CPU instance. To access the per-CPU variable, the `per_cpu` macro is used along with `smp_processor_id`, which is a function that returns the current CPU identifier for which the code is currently executing.

```
per_cpu( cpu_state, smp_processor_id() ) = CPU_ONLINE;
```

The kernel provides other functions for per-CPU locking and dynamic allocation of variables. You can find these functions in ./include/linux/percpu.h.

Summary

As processor frequencies reach their limits, a popular way to increase performance is simply to add more processors. In the early days, this meant adding more processors to the motherboard or clustering multiple independent computers together. Today, chip-level multiprocessing provides more CPUs on a single chip, permitting even greater performance due to reduced memory latency.

You'll find SMP systems not only in servers, but also desktops, particularly with the introduction of virtualization. Like most cutting-edge technologies, Linux provides support for SMP. The kernel does its part to optimize the load across the available CPUs (from threads to virtualized operating systems). All that's left is to ensure that the application can be sufficiently multi-threaded to exploit the power in SMP.

Related topics

- "Inside the Linux scheduler" (developerWorks, June 2006) details the new Linux scheduler introduced in the 2.6 kernel.
- "Basic use of Pthreads" (developerWorks, January 2004) introduces Pthread programming with Linux.
- "Access the Linux kernel using the /proc filesystem" (developerWorks, March 2006) introduces the /proc filesystem, including how to build your own kernel module to provide a /proc filesystem file.
- In "The History of Parallel Processing" (1998), Mark Pacifico and Mike Merrill provide a short but interesting history of five decades of multiprocessing.
- The IBM POWER4 and POWER5 architectures provide symmetric multiprocessing. The POWER5 also provides symmetric multithreading (SMT) for even greater performance.
- The Cell processor is an interesting architecture for asymmetric multiprocessing. The Sony Playstation 3, which utilizes the Cell, clearly shows how powerful this processor can be.
- IBM provides clustering technologies in High-Availability Cluster Multiprocessing (HACMP). In addition to multiprocessing through clustering, HACMP also provides higher reliability through complete online system monitoring.
- Flynn's original taxonomy defined what was possible for multiprocessing architectures. His paper was entitled "Some Computer Organizations and Their Effectiveness." This paper was published in the IEEE Transactions on Computing, Vol. C-21, 1972. Wikipedia provides a great summary of the four classifications.
- The ARM11 MPCore is a synthesizable processor that implements up to four ARM11 CPUs for an aggregate 2600 Dhrystone million instructions per second (MIPS) performance.
- The Beowulf cluster is a great way to consolidate commodity Linux servers to build a high-performance system.
- Standards such as HyperTransport, and the upcoming Common System Interconnect provide efficient chip-to-chip interconnects for next-generation systems.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.