Multi-Processor Systems

Much of the discussion in this course has considered the operating system to be running on a time-shared uni-processor ... and this perspective is adequate to fully understand most of those topics. But increasingly many modern computer systems are now multi-processor:

> Multiple general purpose CPUs (as opposed to GPUs) that are capable of running unrelated programs or threads (unlike SIMD array processors) and (to some degree) share memory and I/O devices.

These systems are interesting because they are independent enough to encounter many of the problems associated with distributed computing ... but (because they share memory and I/O devices) do things that push the distributed systems envelope. As people develop applications to exploit these platforms, it is important that they understand the issues they present.

Why Build Multi-Processor Systems

We continue to find applications that require ever more computing power. Sometimes these problems can be solved by horizontally scaled systems (e.g. thousands of web servers). But some problems demand, not more computers, but faster computers. Consider a single huge database, that each year, must handle twice as many operations as it served the previous year. Distributed locking, for so many parallel transactions on a single database, could be prohibitively expensive. The (seemingly also prohibitively expensive) alternative would be to buy a bigger computer every year.

Long ago it was possible to make computers faster by shrinking the gates, speeding up the clock, and improving the cooling. But eventually we reach a point of diminishing returns where physics (the speed of light, information theory, thermodynamics) makes it ever more difficult to build faster CPUs. Recently, most of our improvements in processing speed have come from:

- smarter pipe-lining and increasingly parallel and speculative execution
- putting more cores per chip, more chips per board, and more boards per computer system.

But it is reasonable to ask whether or not 16x3B instructions per second is actually equivalent to 48B instructions per second? The answer (see Amdahl's Law) depends on whether or not your application can be divided into 16 or more parallely executable sub-tasks. Fortunately, modern operating systems tend to run large numbers of processes, and expensive computations are increasingly designed to be executable in multiple parallel threads..

For these reasons, multi-processor is the dominant architecture for powerful servers and desktops. And, as the dominant architecture, operating systems must do a good job of exploiting them.

Multi-Processor Hardware

The above general definition covers a wide range of architectures, that actually have very different characteristics. And so it is useful, to overview the most prominent architectures.

Hyper-Threading

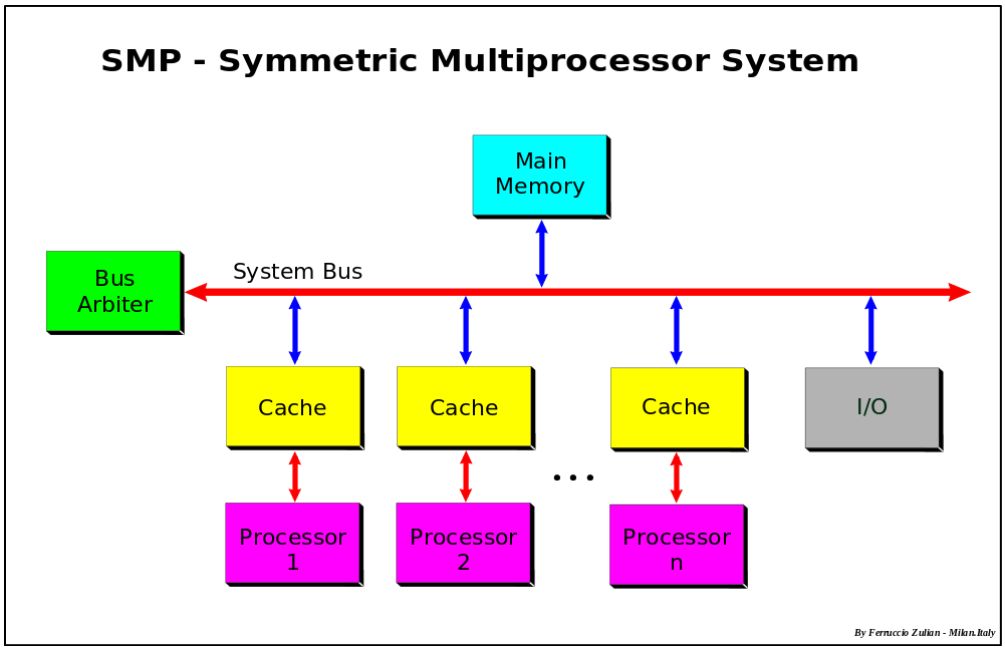
CPUs are much faster than memory. A 2.5GHz CPU might be able to execute more than 5 Billion instructions for second. Unfortunately, 80ns memory can only deliver 12 Million fetches or stores per second. This is almost a 1000x mismatch in performance. The CPU has multiple levels of cache to ensure that we seldom have to go to memory, but even so, the CPU spends a great deal of time waiting for memory.

The idea of hyper-threading is to give each core two sets of general registers, and the ability to run two independent threads. When one of those threads is blocked (waiting for memory) the other thread can be using the execution engine. Think of this as non-preemptive time-sharing at the micro-code level. It is common for a pair of hyper-threads to get 1.2-1.8 times the instructions per second that a single thread would have gotten on the same core. It is theoretically possible to get 2x hyper-threading, but a thread might run out of L1 cache for a long time without blocking, or perhaps both hyper-threads are bocked waiting for memory.

From a performance point-of-view, it is important to understand that both hyper-threads are running in the same core, and so sharing the same L1 and L2 cache. Thus hyper-threads that use the same address space will exhibit better locality, and hence run much better than hyper-threads that use different address spaces.

Symmetric Multi-Processors

A Symmetric Multi-Processor has some number of cores, all connected to the same memory and I/O busses. Unlike hyper-threads these cores are completely independent execution engines, and (modulo limitations on memory and bus throughput) N cores should be able to execute N times as many instructions per second as a single core.

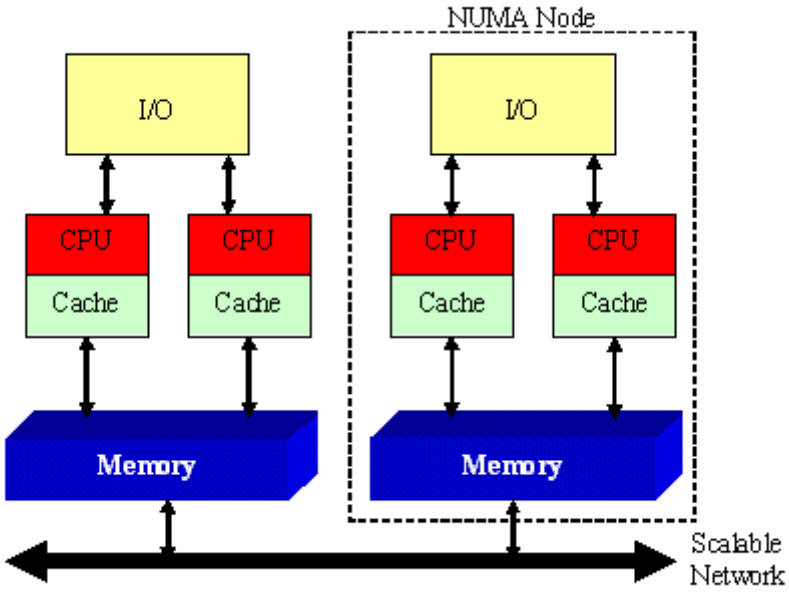


Cache Coherence

As mentioned previously, much of processor performance is a result of caching. In most SMP systems, each processor has its own L1/L2 caches. This creates a potential *cache-coherency* problem if (for instance) processor 1 updates a memory location whose contents have been cached by processor 2. Program execution based on stale cache entries would result in incorrect results, and so must be prevented. There are a few general approaches to maintaining cache coherency (ensuring that there are no disagreements about the current contents of any cache line), and most SMP systems with per-processor caching incorporate some Cache Coherency Mechanism to address this issue.

Cache Coherent Non-Uniform Memory Architectures

It is not feasible to create fast memory controllers that can provide concurrent access to large numbers of cores, and eventually memory bandwidth becomes the bottleneck that prevents scaling to larger numbers of CPUs. A Non-Uniform Memory Architecture addresses this problem by giving each node or CPU its own high-speed local memory, and interconnecting all of the memory busses with a slower but more scalable network.



Operations to local memory may be several times faster than operations to remote memory, and the maximum throughput of the scalable network may be a small fraction of the per-node local memory bandwidth. Such an architecture might provide nearly linear scaling to much larger numbers of processors, but only if we can ensure that most of the memory references are local. The Operating System might be able to deal with the different memory access speeds by trying allocate memory for each process from the CPU on which that process is running. But there will still be situations where multiple CPUs need to access the same memory. To ensure correct execution, we must maintain coherency between all of the per-node/per-CPU caches. This means that, in addition to servicing remote memory read and write requests, the scalable network that interconnects the nodes must also provide cache coherency. Such architectures are called *Cache Coherent Non-Uniform Memory Architectures* (CC-NUMA), and the implementing networks are called *Scalable Coherent Interconnects*. The best known Scalable Coherent Interconnects are probably Intel's Quick Path Interconnect (QPI), and AMD's HyperTransport.

Power Management

The memory and cache interconnections are probably the most interesting part of a multi-processor system, but power management is another very important feature. A multi-core system can consume a huge amount of power ... and most of the time it does not need most of the cores. Many multi-processor systems include mechanisms to slow (or stop) the clocks on unneeded cores, which dramatically reduces system power consumption. This is not a slow process, like a sleep and reboot. A core can be returned to full speed very quickly.

Multi-Processor Operating Systems

To exploit a multi-processor system, the operating system must be able to concurrently manage multiple threads/processes on each of the available CPU cores. One of the earliest approaches was to run the operating system on one core, and applications on all of the others. This works reasonably for a small number of cores, but as the number of cores increases, the OS becomes the primary throughput bottleneck. Scaling to larger numbers of cores requires the operating system itself to run on multiple cores. Running efficiently on multiple cores requires the operating system to carefully choose which threads/processes to run on which cores and what resources to allocate to them.

When we looked at distributed systems, we saw (e.g. Deutsch's Seven Fallacies) that the mere fact that a network is capable of distributing every operation to an arbitrary node does not make doing so a good idea. It will be seen that the same caveat applies to multi-processor systems.

Scheduling

If there are threads (or processes) to run, we would like to keep all of the cores busy. If there are not threads (or processes) to run, we would like to put as many cores as possible into low power mode. It is tempting to think that we can just run each thread/process on the next core that becomes available (due to a process blocking or getting a time-slice-end). But some cores may be able to run some threads (or processes) far more efficiently than others.

- dispatching a thread from the same process as the previous thread (to occupy that core) may be much less expensive because re-loading the page table (and flushing all cached TLB entries) is a very expensive operation.
- a thread in the same process may run more efficiently because shared code and data may exploit already existing L1/L2 cache entries.
- threads that are designed to run concurrently (e.g. parallel producer and consumer communicating through shared memory) should be run on distinct cores.

Thus, the choice of when to run which thread in which core is not an arbitrary one. The scheduler must consider what process was last running in each core. It may make more sense to leave one core idle, and delay executing some thread until its preferred core becomes available. The operating system will try to make intelligent decisions, but if the developers understand how work can best be allocated among multi-processor cores, they can advise the operating system with operations like *sched_setaffinity(2)* and *pthread_setaffinity_np(3)*.

Synchronization

Sharing data between processes is relatively rare in user mode code. But the operating system is full of shared data (process table entries, file descriptors, scheduling and I/O queues, etc). In a uni-processor, the primary causes of race conditions are preemptive scheduling and I/O interrupts. Both of these problems can be managed by disabling (selected) interrupts while executing critical sections ... and many operating systems simply declare that preemptions cannot occur while executing in the operating system.

These techniques cease to be effective once the operating system is running on multiple CPUs in a multi-processor system. Disabling interrupts cannot prevent another core from performing operations on a single global object (e.g. I-node). Thus multi-processor operating systems require some other means to ensure the integrity global data structures. Early multi-processor operating systems tried to create a single, global, kernel lock ... to ensure that only one process at a time could be executing in the operating system. But this is essentially equivalent to running the operating system on a single CPU. As the number of cores increases, the (single threaded) operating system becomes the scalability bottleneck.

The Solution to this problem is finer grained locking. And as the number of cores increased, the granularity required to achieve high parallelism became ever finer. Depending on the particular shared resource and operations, different synchronizations may have to be achieved with different mechanisms (e.g. compare and swap, spin-locks, interrupt disables, try-locks, or blocking mutexes). Moreover for every resource (and combination of resources) we need a plan to prevent deadlock. Changing complex code that involves related updates to numerous data structures to implement fine-grained locking is difficult to do (often requiring significant design changes) and relatively brittle (as maintainers make changes that fail to honor all of the complex synchronization rules). If a decision is made to transition the operating system to finer grained locking, it becomes much more difficult for third party developers to build add-ons (e.g. device drivers and file systems) that will work with the finer grained locking schemes in the hosting operating system.

Because of this complexity, there are relatively few operating systems that are able to efficiently scale to large numbers of multi-processor cores. Most operating systems have chosen simplicity and maintainability over scalability.

Device I/O

If an I/O operation is to be initiated, does it matter which CPU initiates it? When an I/O interrupt comes in to a multi-processor system, which processor should be interrupted? There are a few reasons we might want to choose carefully which cores handle which I/O operations:

- as with scheduling, sending all operations for a particular device to a particular core may result in more L1/L2 cache hits and more efficient execution.
- syncrhonization between the synchronous (resulting from system calls) and asynchronous (resulting from interrupts) portions of a device driver if they are all executing in the same CPU.
- each CPU has a limited I/O throughput, and we may want to balance activity among the available cores.
- some CPUs may be bus-wise closer to some I/O devices, so that operations go more quickly initiated from some cores.

Many multi-processor architectures have interrupt controllers that are configurable for which interrupts should be delivered to which processors.

Non-Uniform Memory Architectures

CC-NUMA is only viable if we can ensure that the vast majority of all memory references can be satisfied from local memory. Doing this turns out to create a lot of complexity for the operating system.

When we were discussing uni-processor memory allocation, we observed that significant savings could be achieved if we shared a single copy of a (read only) load module among all processes that were running that program. This ceases to be true when those processes are running on distinct NUMA nodes. Code and other read-only data should be have a separate copy (in local memory) on each NUMA node. The cost (in terms of wasted memory) is negligible in comparison performance gains from making all code references local.

When a program calls *fork(2)* to create a new process, *exec(2)* to run a new program, or *sbrk(2)* to expand its address space, the required memory should always be allocated from the node-local memory pool. This creates a very strong affinity between processes and NUMA nodes. If it is necessary to migrate a process to a new NUMA node, all of its allocated code and data segments should be copied into local memory on the target node.

As noted above, the operating system is full of data structures that are shared among many cores. How can we reduce the number or cost of remote memory references associated with those shared data structures? If the updates are few, sparse and random, there may be little we can do to optimize them ... but their costs will not be high. When more intensive use of shared data structures is required, there are two general appoaches:

1. move the data to the computation
    - lock the data structure.
    - copy it into local memory.
    - update the global pointer to reflect its new location.
    - free the old (remote) copy.
    - perform all subsequent operations on the (now) local copy.
2. move the computation to the data
    - look up the node that owns the resource in question.
    - send a message requesting it to perform the required operations.
    - await a response.

In practice, both of these techniques are used ... the choice determined by the particulars of the resource and its access.

As with fine-grained synchronization of kernel data structures, this turns out to be extremely complicated. Relatively few operating systems have been willing to pay this cost, and so (again) most opt for simplicity and maintainability over performance and scalability.