Unit IV

Chapter 2: Multiple Processor Systems

Introduction

- Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them.
- Applications in a multi-processing system are broken to smaller routines that run independently. The operating system allocates these threads to the processors improving performance of the system.
- A multiprocessor is a computer system having two or more processing units (multiple processors) each sharing main memory and peripherals, in order to simultaneously process programs.
- At the operating system level, *multiprocessing* is sometimes used to refer to the execution of multiple concurrent processes in a system, with each process running on a separate CPU or core, as opposed to a single process at any one instant.
- When used with this definition, multiprocessing is sometimes contrasted with multitasking, which may use just a single processor but switch it in time slices between tasks (i.e. a time-sharing system).

Multiprocessor Hardware

- Although all multiprocessors have the property that every CPU can address all of memory, some multiprocessors have the additional property that every memory word can be read as fast as every other memory word. These machines are called UMA (Uniform Memory Access) multiprocessors.
- NUMA (Nonuniform Memory Access) multiprocessors do not have this property.

UMA Multiprocessors with Bus-Based Architectures

- The simplest multiprocessors are based on a single bus as illustrated in Fig.
- Two or more CPUs and one or more memory modules all use the same bus for communication. When a CPU wants to read a memory word, it first checks to see if the bus is busy. If the bus is idle, the CPU puts the address of the word it wants on the bus, asserts a few control signals, and waits until the memory puts the desired word on the bus.
- If the bus is busy when a CPU wants to read or write memory, the CPU just waits until the bus becomes idle. Herein lies the problem with this design. With two or three CPUs, contention for the bus will be manageable; with 32 or 64 it will be unbearable. The system will be totally limited by the bandwidth of the bus, and most of the CPUs will be idle most of the time.

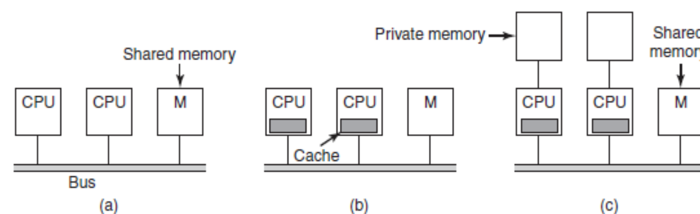


Figure 8-2. Three bus-based multiprocessors. (a) Without caching. (b) With caching. (c) With caching and private memories.

- The solution to this problem is to add a cache to each CPU, as depicted in Fig.
- The cache can be inside the CPU chip, next to the CPU chip, on the processor board, or some combination of all three. Since many reads can now be satisfied out of the local cache, there will be much less bus traffic, and the system can support more CPUs.
- When a word is referenced, its entire block, called a cache line, is fetched into the cache of the CPU touching it.

UMA Multiprocessors Using Crossbar Switches

- Even with the best caching, the use of a single bus limits the size of a UMA multiprocessor to about 16 or 32 CPUs. To go beyond that, a different kind of interconnection network is needed. The simplest circuit for connecting *n* CPUs to *k* memories is the crossbar switch, shown in Fig.

- Crossbar switches have been used for decades in telephone switching exchanges to connect a group of incoming lines to a set of outgoing lines in an arbitrary way.
- At each intersection of a horizontal (incoming) and vertical (outgoing) line is a crosspoint. A crosspoint is a small electronic switch that can be electrically opened or closed, depending on whether the horizontal and vertical lines are to be connected or not.

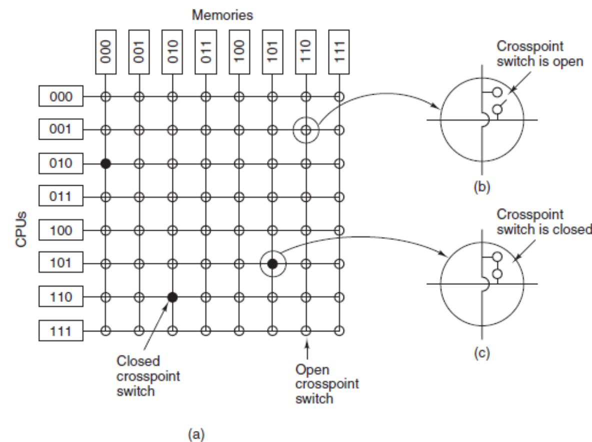


Figure 8-3. (a) An 8 × 8 crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.

UMA Multiprocessors Using Multistage Switching Networks

- A completely different multiprocessor design is based on the humble 2 × 2 switch shown in Fig.
- This switch has two inputs and two outputs. Messages arriving on either input line can be switched to either output line. For our purposes, messages will contain up to four parts, as shown in Fig.
- The *Module* field tells which memory to use.
- The *Address* specifies an address within a module.
- The *Opcode* gives the operation, such as READ or WRITE.
- Finally, the optional *Value* field may contain an operand, such as a 32-bit word to be written on a WRITE. The switch inspects the *Module* field and uses it to determine if the message should be sent on *X* or on *Y*. Our 2 × 2 switches can be arranged in many ways to build larger multistage switching networks

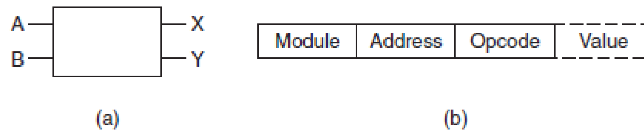


Figure 8-4. (a) A 2 × 2 switch with two input lines, *A* and *B*, and two output lines, *X* and *Y*. (b) A message format.

NUMA Multiprocessors

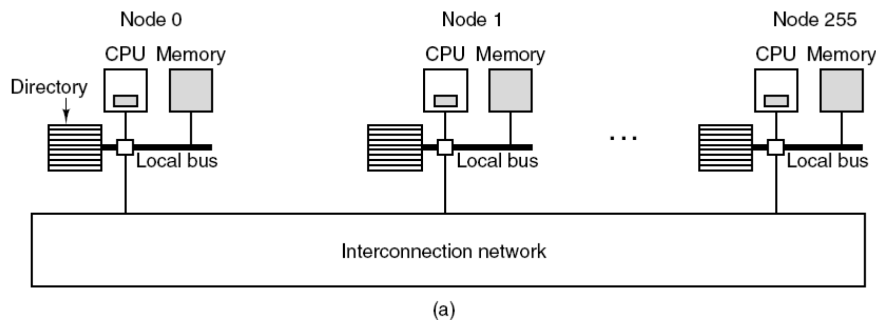
- Single-bus UMA multiprocessors are generally limited to no more than a few dozen CPUs, and crossbar or switched multiprocessors need a lot of (expensive) hardware and are not that much bigger.
- NUMA machines have three key characteristics that all of them possess and which together distinguish them from other multiprocessors:

  1. There is a single address space visible to all CPUs.

  2. Access to remote memory is via LOAD and STORE instructions.

  3. Access to remote memory is slower than access to local memory.
- When the access time to remote memory is not hidden (because there is no caching), the system is called NC-NUMA (Non Cache-coherent NUMA). When the caches are coherent, the system is called CC-NUMA (Cache-Coherent NUMA).
- A popular approach for building large CC-NUMA multiprocessors is the directory-based multiprocessor. The idea is to maintain a database telling where each cache line is and what its status is.

NUMA Multiprocessors



Multicore Chips

- As chip manufacturing technology improves, transistors are getting smaller and smaller and it is possible to put more and more of them on a chip. This empirical observation is often called Moore’s Law, after Intel co-founder Gordon Moore, who first noticed it.
- To put two or more complete CPUs, usually called cores, on the same chip. Dual-core, quad-core, and octacore chips are already common; and you can even buy chips with hundreds of cores.
- While the CPUs may or may not share caches, they always share main memory.
- Special hardware circuitry makes sure that if a word is present in two or more caches and one of the CPUs modifies the word, it is atomically removed from all the caches in order to maintain consistency. This process is known as snooping.
- The result of this design is that multicore chips are just very small multiprocessors.
- In fact, multicore chips are sometimes called CMPs (Chip MultiProcessors).

Manycore Chips

- Multicore simply means ‘‘more than one core,’’ but when the number of cores grows well beyond the reach of finger counting, we use another name. Manycore chips are multicores that contain tens, hundreds, or even thousands of cores.
- Thousands of cores are not even that special any more. The most common manycores today, graphics processing units, are found in just about any computer system that is not embedded and has a monitor. A GPU is a processor with dedicated memory and, literally, thousands of itty-bitty cores. Compared to general-purpose processors, GPUs spend more of their transistor budget on the circuits that perform calculations and less on caches and control logic. They are very good for many small computations done in parallel, like rendering polygons in graphics applications. They are not so good at serial tasks. They are also hard to program.
- While GPUs can be useful for operating systems (e.g., encryption or processing of network traffic), it is not likely that much of the operating system itself will run on the GPUs.

What does *Graphics Processing Unit (GPU)* mean?

- A Graphics Processing Unit (GPU) is a single-chip processor primarily used to manage and boost the performance of video and graphics. GPU features include
- 2-D or 3-D graphics
- Digital output to flat panel display monitors
- Texture mapping
- Application support for high-intensity graphics software such as AutoCAD
- Rendering polygons
- Support for YUV color space
- Hardware overlays
- MPEG decoding

These features are designed to lessen the work of the CPU and produce faster video and graphics. A GPU is not only used in a PC on a video card or motherboard; it is also used in mobile phones, display adapters, workstations and game consoles. This term is also known as a visual processing unit (VPU).

Heterogeneous Multicores

- Some chips integrate a GPU and a number of general-purpose cores on the same die.
- Systems that integrate multiple different breeds of processors in a single chip are collectively known as heterogeneous multicore processors.
- An example of a heterogeneous multicore processor is the line of IXP network processors originally introduced by Intel in 2000 and updated regularly with the latest technology. The network processors typically contain a single general purpose control core (for instance, an ARM processor running Linux) and many tens of highly specialized stream processors that are really good at processing network packets and not much else.

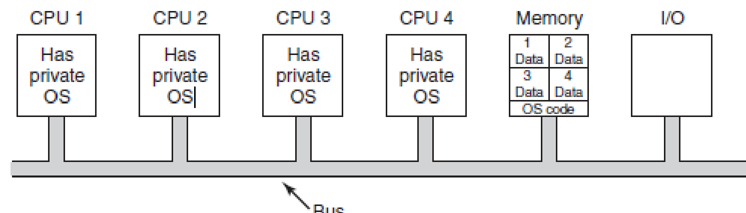
Programming with Multiple Cores

- Current programming languages are poorly suited for writing highly parallel programs and good compilers and debugging tools are scarce on the ground. Few programmers have had any experience with parallel programming and most know little about dividing work into multiple packages that can run in parallel.
- Synchronization, eliminating race conditions, and deadlock avoidance are such stuff as really bad dreams are made of, but unfortunately performance suffers horribly if they are not handled well.

Multiprocessor Operating System Types

Each CPU Has Its Own Operating System:

- The simplest possible way to organize a multiprocessor operating system is to statically divide memory into as many partitions as there are CPUs and give each
- CPU its own private memory and its own private copy of the operating system. In effect, the *n CPUs then operate as n independent computers.*



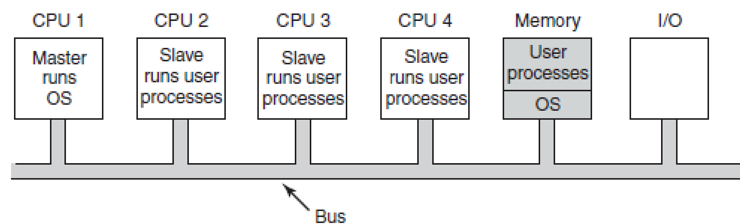
- This scheme is still better than having *n separate computers since it allows all* the machines to share a set of disks and other I/O devices, and it also allows the memory to be shared flexibly.
- Processes can efficiently communicate with one another by allowing a producer to write data directly into memory and allowing a consumer to fetch it from the place the producer wrote it.
- First, when a process makes a system call, the system call is caught and handled on its own CPU using the data structures in that operating system's tables.
- Second, since each operating system has its own tables, it also has its own set of processes that it schedules by itself. There is no sharing of processes. If a user logs into

CPU 1, all of his processes run on CPU 1. As a consequence, it can happen that CPU 1 is idle while CPU 2 is loaded with work.

- Third, there is no sharing of physical pages. It can happen that CPU 1 has pages to spare while CPU 2 is paging continuously. There is no way for CPU 2 to borrow some pages from CPU 1 since the memory allocation is fixed.
- Fourth, and worst, if the operating system maintains a buffer cache of recently used disk blocks, each operating system does this independently of the other ones.

Master-Slave Multiprocessors

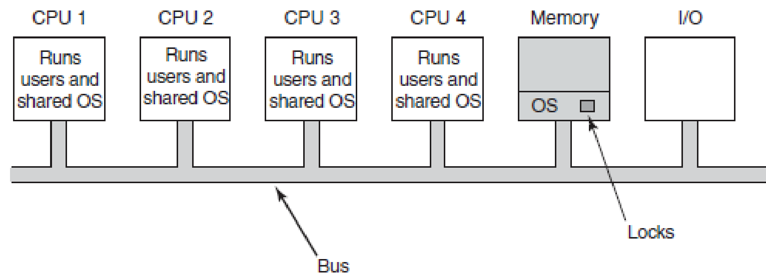
- Here, one copy of the operating system and its tables is present on CPU 1 and not on any of the others. All system calls are redirected to CPU 1 for processing there. CPU 1 may also run user processes if there is CPU time left over. This model is called master-slave since CPU 1 is the master and all the others are slaves.



- The master-slave model solves most of the problems of the first model. There is a single data structure (e.g., one list or a set of prioritized lists) that keeps track of ready processes. When a CPU goes idle, it asks the operating system on CPU 1 for a process to run and is assigned one. Thus it can never happen that one CPU is idle while another is overloaded. Similarly, pages can be allocated among all the processes dynamically and there is only one buffer cache, so inconsistencies never occur.
- The problem with this model is that with many CPUs, the master will become a bottleneck. After all, it must handle all system calls from all CPUs. If, say, 10% of all time is spent handling system calls, then 10 CPUs will pretty much saturate the master, and with 20 CPUs it will be completely overloaded. Thus this model is simple and workable for small multiprocessors, but for large ones it fails.

Symmetric Multiprocessors

- Our third model, the SMP (Symmetric MultiProcessor), eliminates this asymmetry. There is one copy of the operating system in memory, but any CPU can run it. When a system call is made, the CPU on which the system call was made traps to the kernel and processes the system call.



- This model balances processes and memory dynamically, since there is only one set of operating system tables. It also eliminates the master CPU bottleneck, since there is no master, but it introduces its own problems. In particular, if two or more CPUs are running operating system code at the same time, disaster may well result. Imagine two CPUs simultaneously picking the same process to run or claiming the same free memory page.

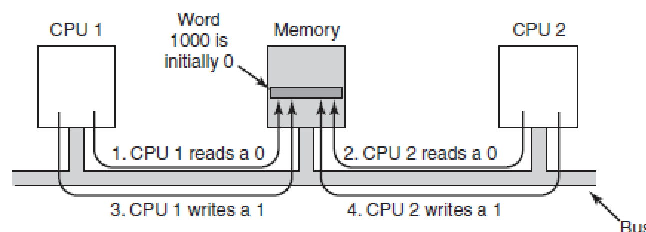
- The simplest way around these problems is to associate a mutex (i.e., lock) with the operating system, making the whole system one big critical region. When a CPU wants to run operating system code, it must first acquire the mutex. If the mutex is locked, it just waits. In this way, any CPU can run the operating system, but only one at a time. This approach is somethings called a big kernel lock.

- This observation leads to splitting the operating system up into multiple independent critical regions that do not interact with one another. Each critical region is protected by its own mutex, so only one CPU at a time can execute it. In this way, far more parallelism can be achieved.

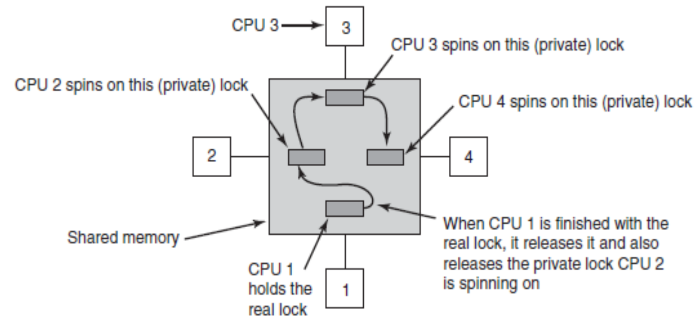
- Each table that may be used by multiple critical regions needs its own mutex. In this way, each critical region can be executed by only one CPU at a time and each critical table can be accessed by only one CPU at a time.

Multiprocessor Synchronization

- The CPUs in a multiprocessor frequently need to synchronize. We just saw the case in which kernel critical regions and tables have to be protected by mutexes.
- To start with, proper synchronization primitives are really needed. If a process on a uniprocessor machine (just one CPU) makes a system call that requires accessing some critical kernel table, the kernel code can just disable interrupts before touching the table.
- On a multiprocessor, disabling interrupts affects only the CPU doing the disable. Other CPUs continue to run and can still touch the critical table. As a consequence, a proper mutex protocol must be used and respected by all CPUs to guarantee that mutual exclusion works.
- The heart of any practical mutex protocol is a special instruction that allows a memory word to be inspected and set in one indivisible operation. We saw how TSL (Test and Set Lock) was used in Fig. to implement critical regions. In step 1, CPU 1 reads out the word and gets a 0. In step 2, before CPU 1 has a chance to rewrite the word to 1, CPU 2 gets in and also reads the word out as a 0. In step 3, CPU 1 writes a 1 into the word. In step 4, CPU 2 also writes a 1 into the word. Both CPUs got a 0 back from the TSL instruction, so both of them now hav e access to the critical region and the mutual exclusion fails.



- To prevent this problem, the TSL instruction must first lock the bus, preventing other CPUs from accessing it, then do both memory accesses, then unlock the bus. An even better idea is to give each CPU wishing to acquire the mutex its own private lock variable to test, as illustrated in Fig



Spinning vs. Switching

- So far we have assumed that a CPU needing a locked mutex just waits for it, by polling continuously, polling intermittently, or attaching itself to a list of waiting CPUs. Sometimes, there is no alternative for the requesting CPU to just waiting.
- For example, suppose that some CPU is idle and needs to access the shared ready list to pick a process to run. If the ready list is locked, the CPU cannot just decide to suspend what it is doing and run another process, as doing that would require reading the ready list. It *must* wait until it can acquire the ready list.