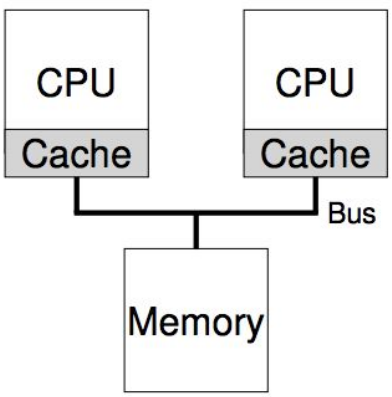
Chapter 10: Multiprocessor Scheduling (Advanced)

Crux: How to schedule jobs on multiple CPUs: How should the OS schedule jobs on multiple CPUs? What new problems arise? Do the same old techniques work?

10.1: Background: Multiprocessor Architecture

- In a system with a single CPU, there are a hierarchy of **hardware caches** that in general help the processor run programs faster
    - Caches are small, fast memories that hold pieces of popular data that are found in the main memory of the system
- Caches are based on the notion of **locality**
    - **temporal locality**
        - when a piece of data is accessed, it is likely to be used again in the near future -> instructions/variables being accessed over and over in a loop
    - **spatial locality**
        - If the program accesses a data item at address x, it is likely to access a data item near x
- However, what happens when you have multiple CPUs in a single system, with a single shared main memory?
    - 

- Ex: we have a program running on CPU 1 that reads a data item with value D at address A
    - because the data is not yet in the cache on CPU 1, the system fetches it from main memory and gets value D
    - The program modifies the value at address A, just updating its cache with the new value D’
        - writing the data through to main memory is slow so the system will usually do that later
    - Assume that the OS decides to stop running the program and move it to CPU 2
    - The program then re-reads the value at address A, there is no such data in CPU 2’s cache and thus the system fetches the value from main memory and gets the old value D instead of the correct value D’ -> OOPS!

- **Cache coherence**
    - basic solution is provided by the hardware -> by monitoring memory access, hardware can ensure that basically the “right thing” happens
- Use **bus-snooping**
    - each cache pays attention to memory updates by observing the bus that connects them to main memory
    - When a CPU then sees an update for a data item it holds in the cache, it will notice the change and **invalidate** its copy -> remove from its own cache or update it (put new value on the cache)
        - Write-back caches make this more difficult because the write to main memory isn’t visible till later

10.2: Don’t Forget Synchronization

- When accessing shared data items or structures across CPU,s mutual exclusion primitives should be used to guarantee correctness

10.3: One Final Issue: Cache Affinity

- **Cache affinity:** a process, when run on a particular CPU, builds up a fair bit of state in the caches (and TLBs) of the CPU.
    - Next time the process runs, it is often advantageous to run it on the same CPU, as it will run faster if some state is already present on the cache
    - If we run the process on another CPU, the performance will be much worse because a cache miss will occur and we would have to store the data directly in the cache
- Thus, it is important for the to consider cache affinity when scheduling decisions

10.4: Single-Queue Scheduling

- Using the same technique for single processor scheduling by putting all jobs that need to be scheduled into a single queue -> **SQMS (single-queue multiprocessor scheduling)**
- Advantages
    - Simplicity: Chooses the best job to run next and adapts it to work on more than one CPU
- Shortcomings of SQMS
    - **Lack of scalability**
        - to ensure the scheduler works correctly on multiple CPUs, the developers will have inserted some form of locking into the code
        - This means that (locks) performance is reduced, particularly as the number of CPUs grow
    - **Cache affinity**
        - Does not preserve cache affinity as it just picks the next best job -> isn’t cache aware

10.5: Multi-Queue Scheduling (MQMS/multi-queue multiprocessor scheduling)

- Each queue will likely follow a scheduling discipline such as round robin, though of course any algorithm can be used
- When a job enters the system, it is placed on exactly one scheduling queue, according to some heuristic such as random.
- Scheduled essentially independently, avoiding problems of information sharing and synchronization
    - you are going to have different jobs in different queues -> no overlap
- Advantages
    - Inherently more scalable -> number of queues grows along with the number of CPUs
        - lock and contention should not be a problem
    - Provides cache affinity
        - jobs stay on the same queue and thus have inherent cache affinity
- Disadvantages
    - **Load imbalance**
        - **CPU time for each process can vary depending on the queue partitions**
        - **Solution: Migration**
            - Move jobs from one CPU to another to achieve balance
        - **Work stealing**
            - a source queue that is low on jobs will occasionally peek at another queue to see how full it is
                - the source queue can steal a job from the target queue
        - this can result in high overhead from switching and stealing too much

10.7: Linux Multiprocessor Schedulers

- In the Linux community, no common solution has approached to building a multiprocessor scheduler

Eventual Consistency

- Eventual consistency is a consistency model used in distributed computing to **achieve high availability that informally guarantees** that, **if no new updates are made to a given item, eventually accesses to that item will return the last updated value**
- Eventually-consistent services are often classified as providing BASE (**Basically Available, Soft State, Eventual Consistency) semantics** rather than ACID
- Eventual Consistency is purely a liveness guarantee and does not make safety guarantees
- **Key point:** eventual consistency is a weak guarantee that reads eventually return the same value and does not make the safety guarantee as an eventually consistent system can return any value before it converges

Conflict Resolution

- In order to ensure replica convergence, a system must reconcile differences between multiple copies of distributed data -> consists of two parts
  - Exchanging versions or updates of data between servers (anti-entropy)
  - **Choosing an appropriate final state when concurrent updates have occurred, called reconciliation**
- the most appropriate approach to reconciliation depends on the application
  - A widespread approach is “last writer wins”
  - Another is to invoke a user-specified conflict handler
  - Timestamps and vector clocks
- Reconciliation of concurrent writes must occur sometime before the next read, and can be scheduled at different instants
  - **Read repair**: The correction is done when a read finds an inconsistency
  - **Write repair:** The correction takes place during a write operation, if an inconsistency has been found, slowing down the write operation
  - **Asynchronous repair:** the correction is not part of a read or write operation

Strong Eventual Consistency
- Strong eventual consistency guarantees safety on top of liveness in that any two nodes that have received the same (unordered) set of updates will be in the same state

Multi-Processor Systems

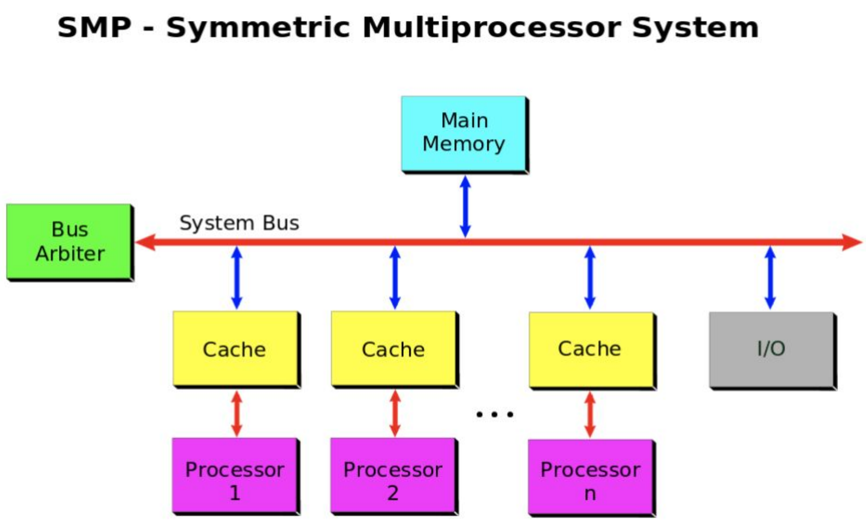
Why Build Multi-Processor Systems
- We continue to find applications that require more and more computing power
- Sometimes these problems can be solved by horizontally scaled systems (e.g. thousands of web servers)
- Some problems demand not more computers, but faster computers
  - Consider a single huge database, that each year, must handle twice as many operations as it served the previous year
  - Distributed locking could be prohibitively expensive

Multi-Processor Hardware

Hyper Threading
- CPUs are much faster than memory
  - e.g. a 2.5GHz CPU might be able to execute more than 5 billion instructions for second
  - CPU has multiple levels of cache to ensure that we seldom have to go to memory
- Idea of hyper-threading
  - give each core two sets of general registers and the ability to run independent threads
  - When one of those threads is blocked (waiting for memory), the other thread can be using the execution engine -> like non preempted time sharing

- Both hyper-threads are running in the same core, and so they share the same L1 and L2 cache
- Thus hyperthreads that use the same address space will exhibit better locality, and hence run much better than hyperthreads which use different address spaces

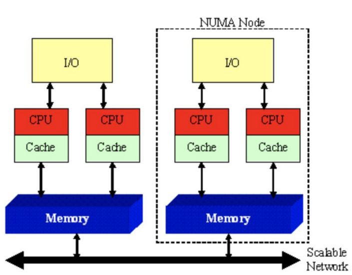
Symmetric Multi-Processors

- has some number of cores, all connected to the same memory and I/O busses
    - Unlike hyperthreads, these cores are completely independent execution engines
- 

Cache Coherence

- Much of the processor performance is a result of caching
- In most SMP systems, each processor has its own L1/L2 cache
- because one CPU can update a memory location whose contents have been cached by another CPU, we need some sort of cache coherency mechanism

Cache Coherent Non-Uniform Memory Architectures

- Non-Uniform Memory Architectures address the issues that memory bandwidth becomes the bottleneck that prevents scaling to large number of CPUs
    - **gives 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
- Maximum throughput of the scalable network may be a small fraction of the per-node local memory
- There will be situations where multiple CPUs need to access the same memory
    - Maintain coherency between all of the per-node/per-CPU caches
    - the scalable network that connects the memory busses must also provide cache coherence

Power Management

- A multi-core system can consume a huge amount of power and most of the time it doesn't even 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

Multi-Processor Operating System

- To exploit a multi-processor system, the OS must be able to concurrently manage multiple threads/processes on each of the available CPU cores

Scheduling

- If there are threads (or processes) to run, we would like to keep all of the cores busy.
    - If there are threads that do not need to run, we would like to put as many cores as possible into low power mode
- It may be tempting to think that we can just run a thread on the next core that becomes available, but some cores may be able to run threads or processes much more efficiently than others
    - dispatching a thread from the same process as the previous thread may be much less expensive because re-loading the page table and flushing all TLB entries is an expensive operation
    - a thread in the same process may run more efficiently because of shared code and data may exploit already existing L1/L2 cache entries
    - threads that are designed to run concurrently (e.g. parallel producer/consumer communication through shared memory) should be run on one distinct core

Synchronization

- Sharing data between processes is relatively rare in user mode -> however, OS is full of shared data such as process table entries, file descriptors, scheduling and I/O queues, etc.
- Disabling interrupts cannot prevent another core from performing operations on a single global object such as an inode
- Solution: finer grained locking

- 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, blocking mutexes)
- However, finer grained locking means that it is more difficult for third party developers to build add-ons that will work with finer grained locking schemes

Device I/O

- There are a few reasons we might want to choose carefully which cores handle which I/O operations
  - sending all operations for a particular device to a particular core may result in more L1/L2 cache hits
  - synchronization between the synchronous (resulting from sys 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, so we would want to balance activity
  - some CPUs may be bus-wise closer to some I/O devices
- Many multi-processor architectures have interrupt controllers that are configurable for which interrupts should be delivered to which processors

Non-Uniform Memory Architecture

- CC-NUMA (mentioned above) is only viable if we can ensure that the vast majority of all memory references can be satisfied from local memory
- When we were discussing single processor memory allocation, we observed that **significant savings can be achieved through sharing a single copy of a read only load module among all processes that were running the same program**
  - This ceases to be true in multiprocessor systems
  - Code and read-only code that are shareable should have a separate copy (in local memory) on each NUMA node
- When we call fork(2), to create a new process, exec(2) to run a new program, or sbrk(2) to expand the address space, the required memory should always be located from the node-local memory pool
  - creates strong affinity between nodes and memory pools/processes
  - 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
- How can we reduce the number or cost of remote memory references associated with shared data structures in the OS -> so that we don't have to copy allocated code and data segments into the local memory of target nodes
  - 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

Cluster Concepts

Cluster

- Common types of clusters include
  - load sharing clusters, which divide work among members
  - high availability clusters, where backup nodes take over when primary nodes fail
  - information sharing clusters, which ensure the dissemination of information throughout a network

Membership

- If a cluster is defined as a networked connection of nodes who consider themselves to be participants in the cluster, then obviously "membership" is key
- two types of membership
  - potential, eligible, or designated members
  - active or currently participating members
- only active members can communicate with one another
- In most clusters, a node has to be explicitly configured or provisioned into the cluster so that the set of potential members is well known and perhaps even close to being new members
- There are still nodes that welcome any members at any time

Degree of Coupling

- Horizontally scaled systems generally seek maximum independence between the participating nodes -> if they share no resources, there should be little need for them to coordinate their activities with one another -> **loosely coupled**
- **How loose coupling is a good thing**
  - if there are no shared resources, there is no danger of conflicting updates from other servers
    - can safely cache frequently used data, without fear that it will be invalidated by other servers
  - no need to synchronize shared resource use -> makes code simpler and eliminates potential bottlenecks
  - if there is little communication between nodes, they can operate completely in parallel
    - with good scalability!
  - little coordination between nodes -> unlikely that a bug or failure on one node will affect others

- However, sometimes sharing is inevitable
  - Consider a database server which must service many thousands of requests per second to a single, shared, database
  - Distributed systems that share resources and coordinate activities with one another are said to be **tightly coupled**
  - Ultimate extreme: **single system image** -> shares all state and resources so perfectly that application cannot tell that they are all running on a single computer

Node Redundancy

- In a clustered system, work is divided among the active members. To reduce distributed synchronization, it is common to partition the work (e.g. designate each server responsible for a certain subset (e.g. a file system, a range of keys) of requests, and route all requests to the designated user
- Two approaches to take to high availability
  - Active/Stand-By
    - The system is divided into active and stand-by nodes
    - The incoming requests are partitioned among the active nodes
    - standby nodes are idle until an active node fails
  - Active/Active
    - the incoming requests are partitioned among all of the available nodes -> if one node fails, then the work will be distributed amongst the other nodes
- an active/active architecture achieves better utilization, and so may be more cost-effective
  - when a failure occurs, the load on the surviving nodes is increased and so performance may suffer
- an active/standby architecture normally has idle capacity, but may not suffer any performance degradation after a failure

Heart Beat