Multiprocessor Systems

Operating Systems

Oleg Goldshmidt

ogoldshmidt@computer.org

Lecture 7

Single CPU Computers

- the CPU can execute only one instruction at a time
- program execution is purely sequential
- multiprogramming is possible thanks to time division
- increasing performance means making the clock faster
- fundamental limit #1: $c \approx 20\,\mathrm{cm/\,ns}$ in wire or fiber
  - $10\,\mathrm{GHz}$ system must be smaller than $2\,\mathrm{cm}$
- fundamental limit #2: heat dissipation
  - the smaller the system the more heat it generates
  - the smaller the system the harder it is to dissipate
  - (Intel say) the melting point of (doped) silicon is not so far away

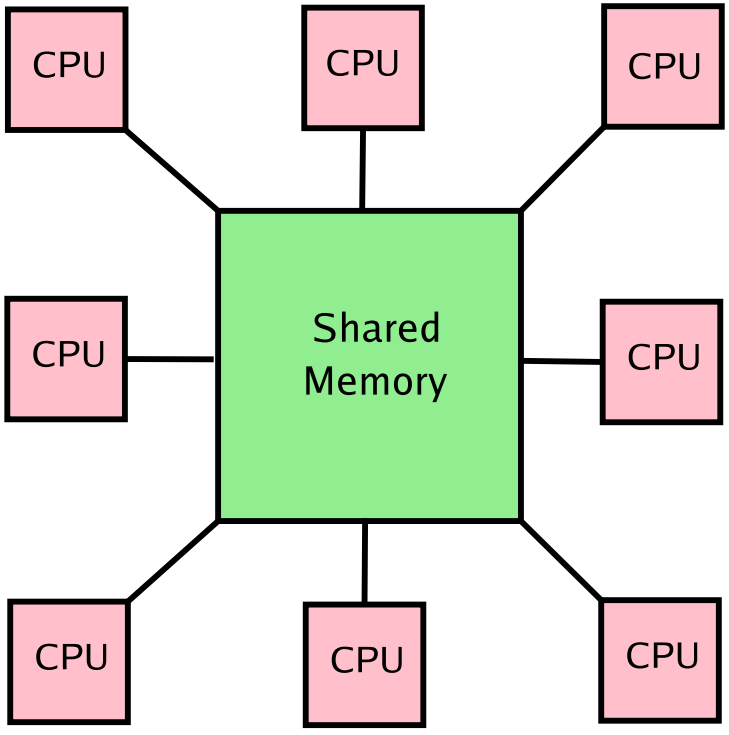
Solution: Parallelization

- many CPUs running at “normal” speed, for some definition of “normal”
- speed up computations
  - at least those that can be parallelized
- deal with heavier loads
  - different CPUs deal with different transactions, users
- enormous range of systems:
  - single servers with 2, 4, 8, 16, and more CPUs
  - supercomputers and clusters ($10 \div 10^5$ CPUs)
  - internet-wide computations (e.g. SETI@home)
  - grid computing

Stored-Program Multiprocessors

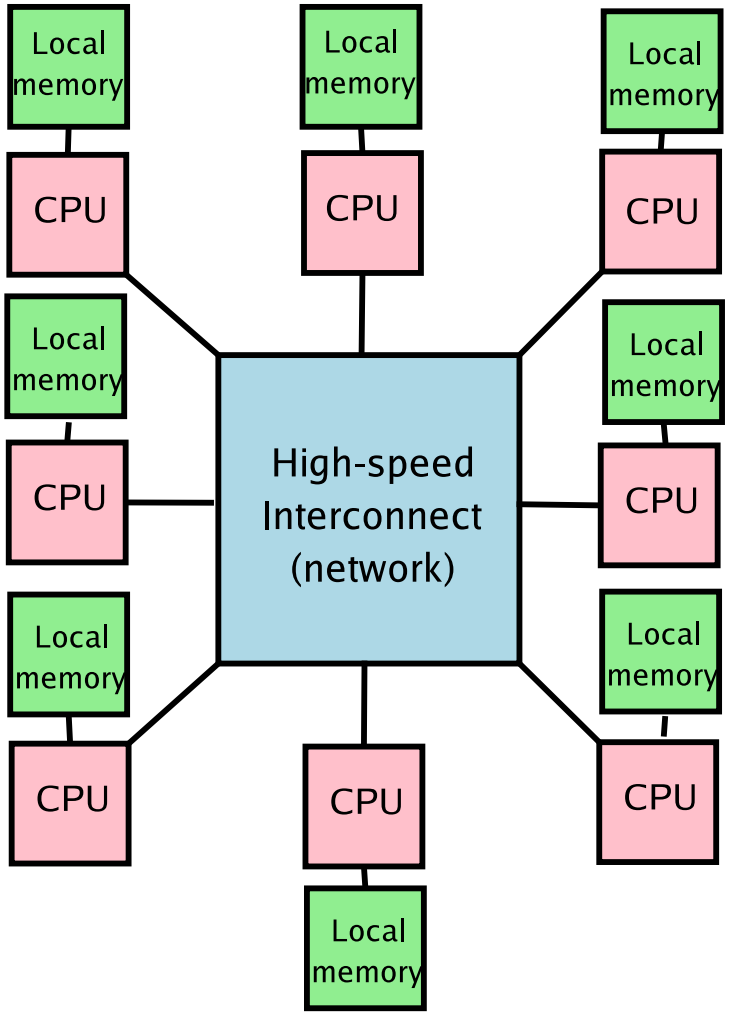
- separation of CPUs and memory
- how CPUs access memory is central to what follows
  - where istructions and data live
  - how fast can the CPU access the data
  - caching
  - shared memory
- interconnect between CPUs and memory, its properties are crucial
  - topology and connectivity
  - bandwidth and latencies
  - blocking properties
  - routing

Shared-Memory Multiprocessors



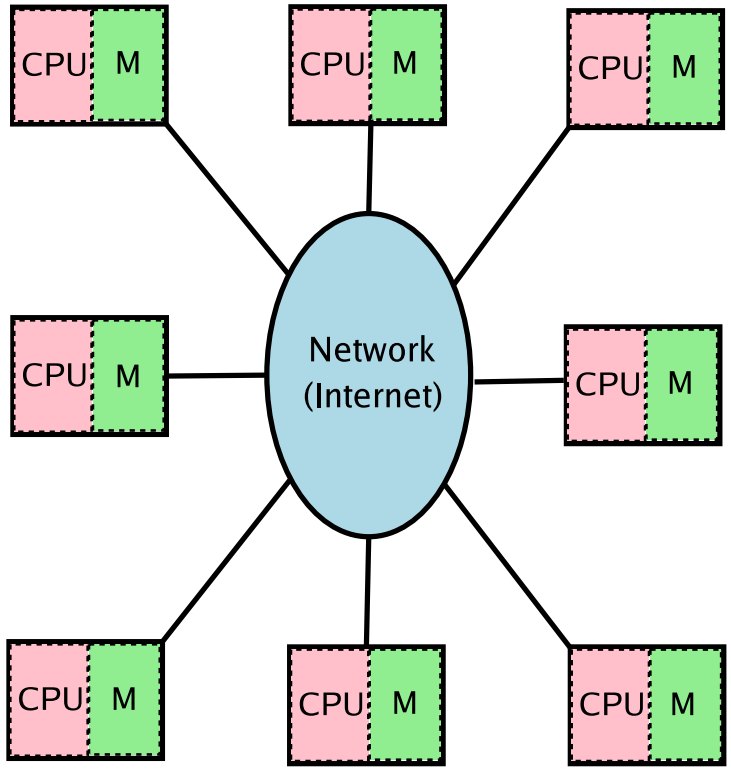
- between 2 and hundreds of CPUs
- all have access to the entire physical memory
- LOAD and STORE individual words
- memory access times $10 \div 50\,\mathrm{ns}$
- looks simple
- difficult to implement
- much message-passing under the covers

Message-Passing Multiprocessors



- $10^2 \div 10^5$ CPUs
- each CPU has its own memory, inaccessible by others
- communicate by sending messages over the interconnect
- latencies $1 \div 50\ \mu s$
- much easier to build than shared-memory ones
- harder to program
- each node can be a shared-memory multiprocessor

Distributed Multiprocessors



- potentially huge number of nodes
- each node is a complete system
- very similar to message-passing multicomputers
- but long latencies $1 \div 50\,\mathrm{ms}$
- often no QoS (e.g. over Internet)
- “loosely-coupled” vs. “tightly-coupled”

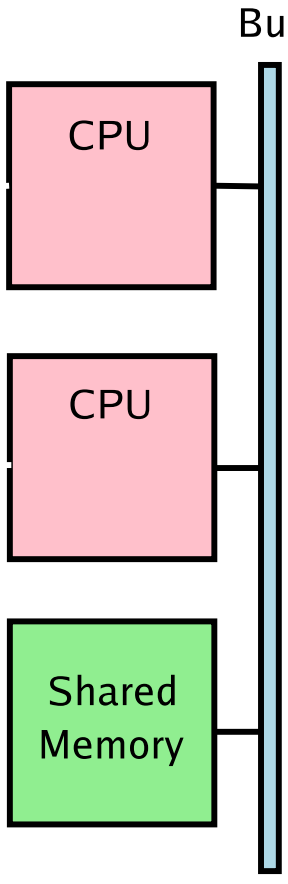
Shared-Memory MP Quirks

- multiple CPUs share full access to a common RAM
- a process running on a CPU sees a normal virtual address space
- the memory is usually paged
- things really can happen simultaneously
- unusual property: a CPU can STORE a word in memory and then LOAD it, and get a different value
  - because another CPU changed it
  - allows a form of interprocessor communication
  - must be very careful indeed with locking

Shared-Memory Multiprocessor OS

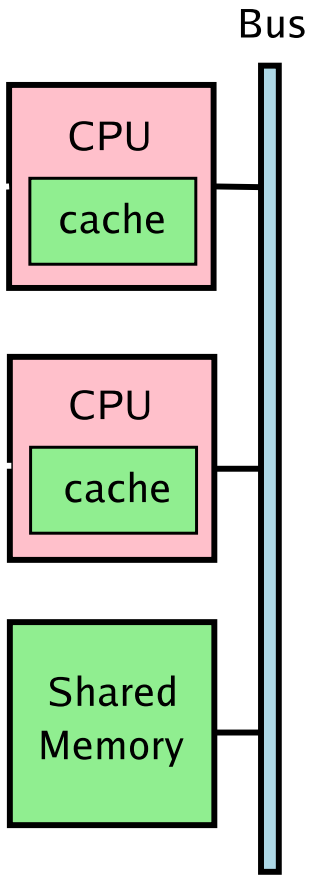
- a regular OS for the most part
  - system calls
  - memory management
  - file systems
  - I/O
- but not quite ordinary
  - process synchronization
  - resource management
  - scheduling

Shared-Memory MP Hardware: UMA



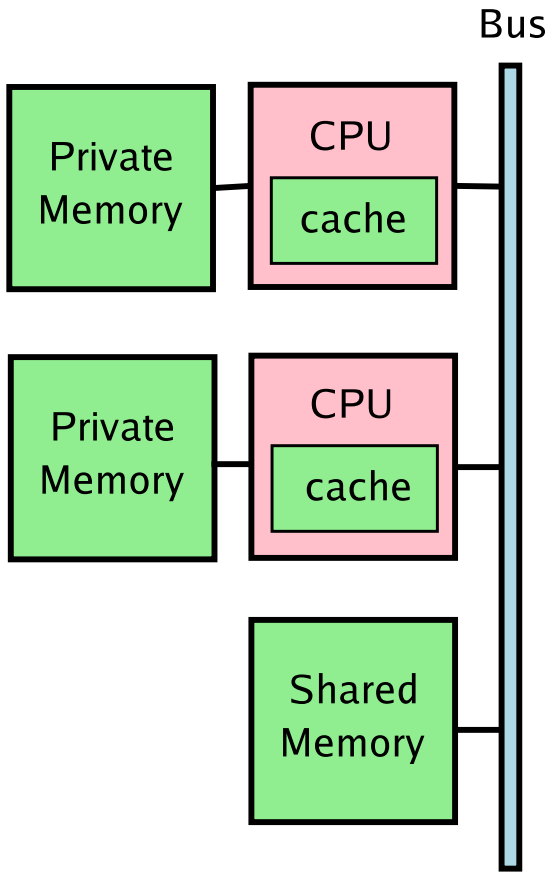
- CPUs share a bus for communication
  - contention limited by the bus bandwidth — non-scalable beyond a few CPUs

Shared-Memory MP Hardware: UMA



- CPUs share a bus for communication
  - contention limited by the bus bandwidth — non-scalable beyond a few CPUs
- solution: add caches to CPUs

Shared-Memory MP Hardware: UMA

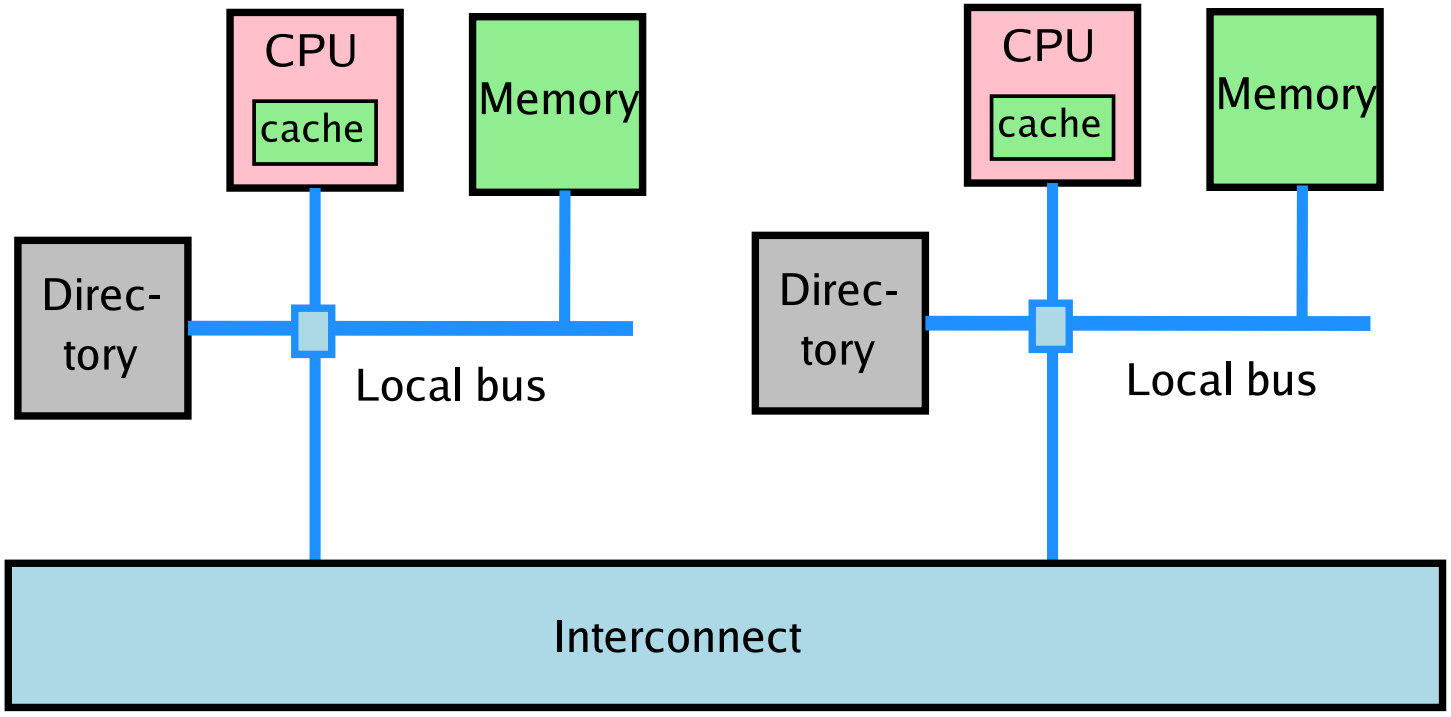


- CPUs share a bus for communication
  - contention limited by the bus bandwidth — non-scalable beyond a few CPUs
- solution: add caches to CPUs
- optimization: place all text sections, read-only data, stacks, local variables in private memory, access the bus for writeable shared data
  - requires co-operation from compilers

NUMA Multiprocessors

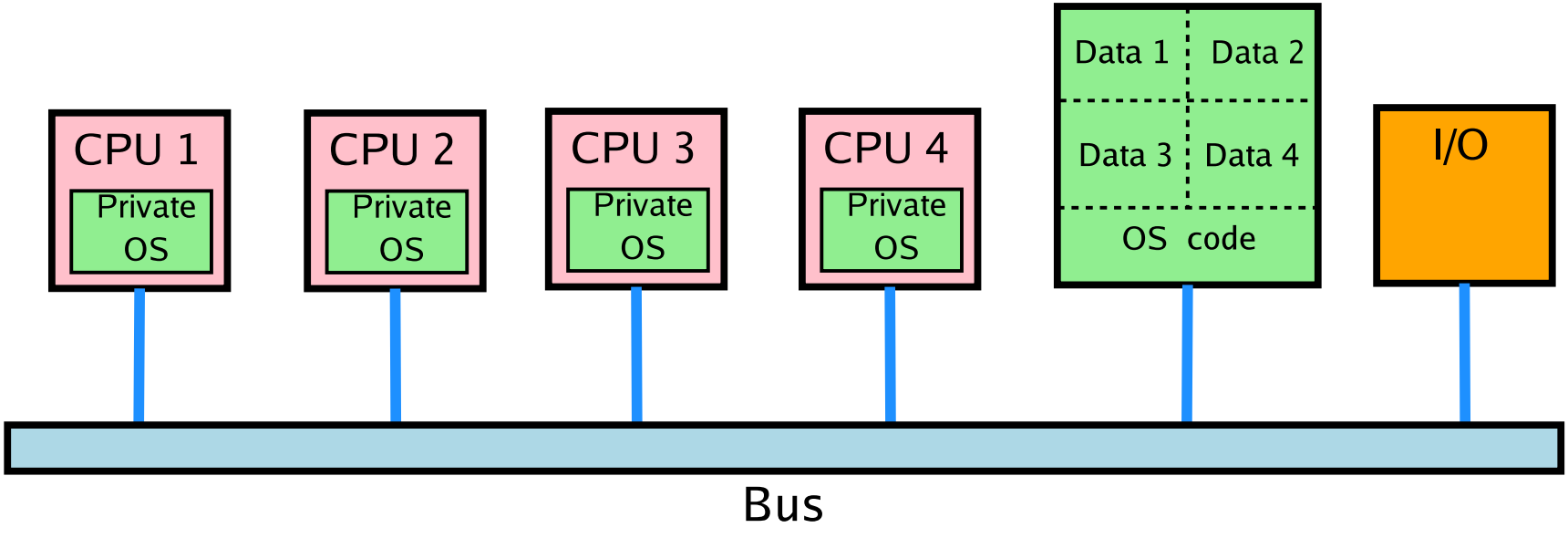
- single-bus UMA multiprocessors are not very scalable
- crossbar and multistage switching help, but not very much
- to go beyond 100 CPUs something has got to give
- NUMA — give up the uniform memory access time
- all CPUs still see all the RAM and use a single address space, but local memory access is faster than remote
- all UMA programs will run on NUMA machines, but slower
- NC-NUMA — no caching
- CC-NUMA — coherent caches are present

Directory-Based NUMA



- maintain a DB (in very fast HW) that knows where each cache line is and what its status (clean or dirty) is
- query the DB for every memory access

Multiprocessors with Private OS

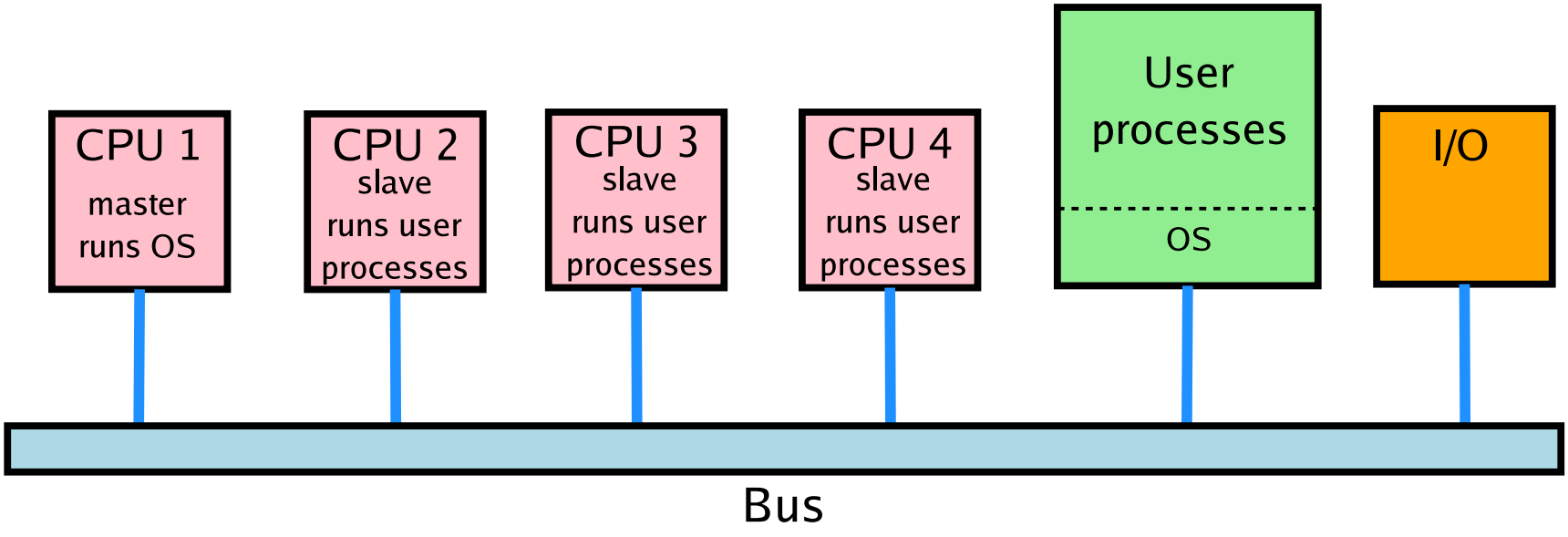


- each CPU has private memory and a private copy of OS
- effectively n independent computers
- optimization: share OS code

Multiprocessors with Private OS II

- better than n independent computers
  - shared I/O
  - flexible memory allocation
  - effective inter-processor communication
- system calls are handled locally — private tables etc
- no process sharing: CPU 1 idle while CPU 2 overloaded
- no page sharing: CPUs cannot borrow/loan pages
- local buffer caches (of recently used disk blocks)
  - if a block is present and dirty in multiple buffer caches the system is in inconsistent state
  - eliminating buffer caches hurts performance

Master-Slave Multiprocessors



- only one CPU (“master”) has OS, tables, etc
- all system calls are redirected to the master CPU
- master can also run user processes, if it is not loaded

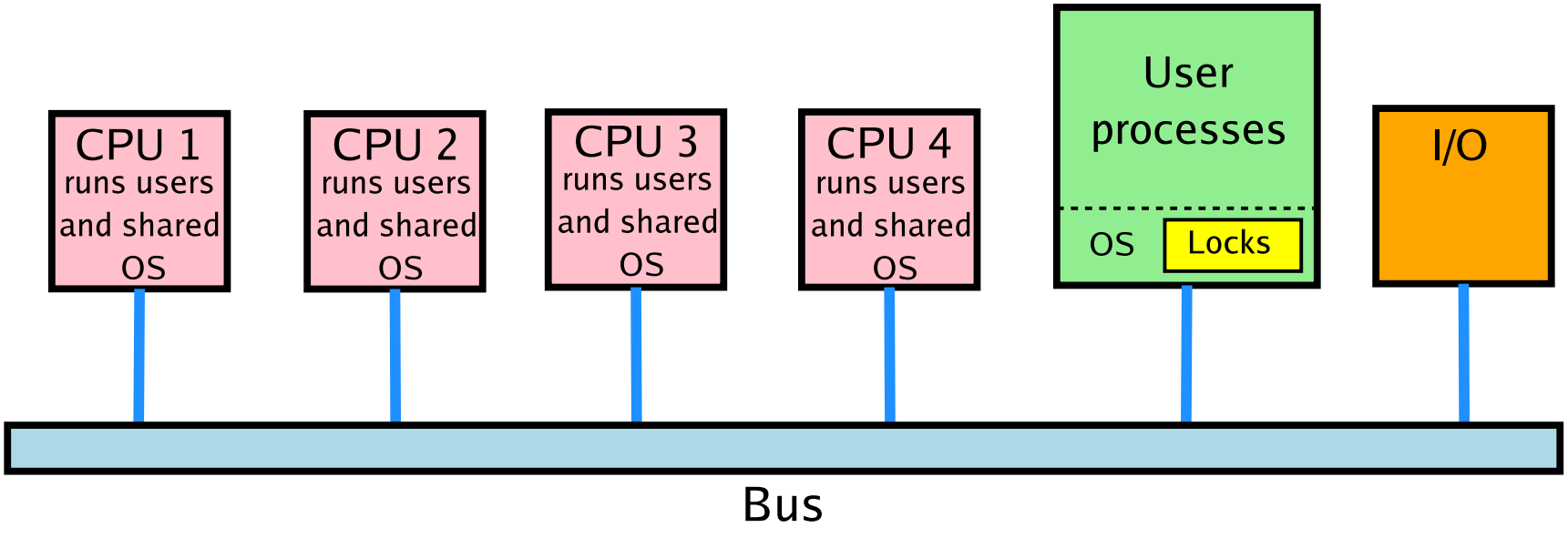
Master-Slave Multiprocessors II

- solves most of the problems of the private OS scheme
  - there is a single set of OS data structures
  - a CPU will never stay idle when another is overloaded
  - pages can be allocated among all the processes dynamically
  - there is one buffer cache, so no inconsistencies will occur
- problem: the master CPU is a bottleneck
  - must handle all the system calls from all the slaves
  - example: if 10% of the time is spent in system calls, the master will be saturated by 10 CPUs

Master-Slave Multiprocessors III

- usefulness depends on the workload
  - OK for workloads with few system calls
  - very important application: heavy number crunching
  - used in HPC, supercomputers
- not scalable for workloads with a lot of system activity
- need another model...

Symmetric Multiprocessors (SMP)



- there is one copy of OS, but any CPU can run it
- when a CPU needs to perform a system call it does

Symmetric Multiprocessors II

- balances processes and memory dynamically
- there is only one set of OS tables
- eliminates the master CPU bottleneck
- problem: need to synchronize the CPUs
  - imagine 2 CPUs scheduling the same process to run
  - or claiming the same free memory page
- solution: protect the OS with a mutex
  - any CPU can run the OS, but only one at a time can do it
  - almost as bad as master-slave: CPUs will queue to get the OS

SMP Synchronization

- solution: split the OS into independent critical regions, protect each with its own mutex
- some tables may be used by multiple critical sections
  - e.g. process table is used by
    - scheduler
    - fork()
    - signal handling
  - such tables need their own mutexes
- such organization is hard to design...
- ... and is even harder to program

No Simple Solutions

- on uniprocessor machines simple, direct, brute-force solutions are possible in some cases
  - the only thing we need to take care of is interrupts
  - in principle, one can disable interrupts while in critical region
- disabling interrupts on SMP affects only the disabling processor
- one big lock for the OS (BKL in the Linux world) reduces SMP performance to that of master-slave: CPUs queue to acquire the lock
- solution: lock different data structures separately
  - NB: always lock data, not code!
- what are the mechanisms?

Test and Set Lock (TSL)

- TSL is a HW instruction that lies in the heart of any practical mutex protocol
- atomic operation on a memory word
  - reads a word from memory
  - stores the word in a register
  - writes 1 (or a non-zero value) into the memory word
- two memory operations, really: `LOAD` and `STORE`
- takes two bus cycles
- always works as expected on uniprocessor machines

TSL on SMP

- Both CPUs may acquire the lock, can proceed into the critical region
- mutual exclusion is broken, unless SL locks the bus
- need a special locking line and protocol on the bus

CPU 1	CPU 2
`READ 0`	`...`
`...`	`READ 0`
`WRITE 1`	`...`
`...`	`WRITE 1`

What Does the Waiting CPU Do?

- keeps testing the lock (“spins”)
- wastes cycles
- accesses the bus all the time, slowing everybody else down!
- but what about cache?
- in theory, it should work
  - the CPU should be able to test the lock in its own cache
  - when the lock is released, all the caches will be invalidated

Avoiding Bus Thrashing

- in practice, caches work in 32-bit or 64-bit blocks
- TSL brings a whole block into a cache, and invalidates it, since it is a write
- result: “bus thrashing”
- need to get rid of TSL writes for requesters
- make the requesting CPU do a pure read to see if the lock is free
- every CPU polls its own cache in read-only mode
- when the lock is released, caches are invalidated
- the next attempts to TSL does not guarantee lock acquisition!

Binary Exponential Backoff

- enhancement to further reduce bus traffic
- same algorithm as in Ethernet
- insert a delay loop between successive polls
  - initial delay — 1 instruction
  - if the lock is busy — increase delay to 2 instructions
  - continue until some maximal delay is reached
- tradeoff:
  - low maximal delay — faster acquisition on release, more thrashing
  - high maximal delay — reduced thrashing, slow acquisition

Private Lock Variables

- a CPU that fails to acquire a lock allocates a private lock variable
  - private lock resides in its own cache block
- adds itself to the list of CPUs waiting for the (real) lock
- when the holder releases the real lock it also releases the private lock of the first CPU on the list
  - remember: memory is shared
- further complication: can 2 CPUs attach themselves to the list simultaneously?

To Spin Or Not To Spin?

- on UP spinning does not make sense — no one else can release the lock

To Spin Or Not To Spin?

- on UP spinning does not make sense — no one else can release the lock
- on SMP sometimes spinning cannot be avoided
  - an idle CPU needs to pick a process from a (locked) ready queue

To Spin Or Not To Spin?

- on UP spinning does not make sense — no one else can release the lock
- on SMP sometimes spinning cannot be avoided
  - an idle CPU needs to pick a process from a (locked) ready queue
- spinning wastes CPU cycles
- so does switching contexts (at least twice):
  - save current process state
  - pick a process to run from ready queue (another lock etc.)
  - load and start the new process
  - suffer from many cache misses, TLB faults etc.

Designing for Concurrency

- What causes concurrency issues?
  - HW and SW interrupts, sleeping, preemption, SMP
- design protection and locking from the start
  - not too difficult, if you realize that you need to protect the data, not code paths (have I said this before?)
  - retrofitting locks in is really difficult, and the results are not pretty
- always design and develop for the worst case: SMP, preemption, etc
- always test on SMP

Linux Spinlocks

- by far the most common type of lock
- does what it says: if a lock is contended the waiting thread “spins”, i.e., busy-waits until the lock is released
- it is not wise to hold a spinlock for a long time
- architecture-dependent, assembly implementation
- useless on single-processor machines
  - are compiled away if `CONFIG_SMP` is not defined
- can be used in interrupt handlers (cannot sleep)
  - must disable local (this CPU) interrupts
  - otherwise there is a deadlock (can you see it?)
  - why only local interrupts must be disabled?

Spinlock Interface

```
spinlock_t lck = SPIN_LOCK_UNLOCKED;

/* generic interface */
spin_lock(&lck);
/* critical section */
spin_unlock(&lck);

/* disable interrupts and lock */
unsigned long flags;
spin_lock_irqsave(&lck,flags);
/* critical section */
spin_unlock_irqrestore(&lck,flags);
```

Semaphores

- not very relevant to SMP — SMP protection is done by spinlocks
- spinlocks vs. semaphores:
  - low overhead is important — spinlocks preferred
  - short lock holding time — spinlocks preferred
  - another CPU may access the data — spinlocks required
  - in interrupt context — spinlocks required
  - long lock holding time — semaphores preferred
  - may sleep while holding lock — semaphores required

Practical OS Coding Rules

- if a function acquires a lock and calls another function that may attempt to acquire the same lock, your machine will hang
- avoid taking multiple locks
- if you asolutely have to, determine the scope of the lock
  - take more local lock first
  - e.g., if you lock a device you are writing a driver for — obtain this lock first
  - never hold a spinlock if you can sleep
  - if you need both a semaphore and a spinlock, obtain the semaphore first
    - semaphores can sleep, spinlocks cannot!

SMP And Preemption

- the kernel is preemptive
  - can stop at any time to allow another process to run
- a task can begin running in the same critical region as another task that was preempted
- the kernel uses spinlocks as markers for non-preemptive regions
  - while a spinlock is held the kernel is not preemptive
- same as SMP: SMP-safe means preemption-safe!
- not always...

Disabling Preemption

- some situations do not require SMP spinlocks, but preemption must be disabled
- example: per-processor data
  - no need to protect from SMP, since the data are CPU-specific
  - still need a spinlock to disable preemption
- preemption counter: zero if preemptive

```
preempt_disable();
preempt_enable();
preempt_count();
get_cpu();
put_cpu();
```

Side Note: Superthreading

- a.k.a. time-slice multithreading
- interleave instructions from different threads
- each pipeline stage can contain instructions from one thread only
- scheduling logic switches between threads
- helps alleviate memory latency
  - if one thread requests data from main memory that is not in cache it stalls for several cycles
  - another thread can proceed with execution, keeping pipelines full
- does not help with instruction-level parallelism
  - if on a given cycle not enough instructions can be parallelized, there will still be waste

Side Note: Hyperthreading

- removes the “one thread per time slot” restriction
- Intel Pentium 4 Xeon: 2 threads per CPU
- not very complicated: adds about 5% to the die area for Xeon
- from the OS perspective: 2 logical processors, equivalent to 2-way SMP
  - installing an OS on a Xeon means installing an SMP kernel
  - can be disabled (why?)
- both logical CPUs share the same cache

Side Note: SMT and Caching

- no cache coherence problems that plague SMPs
- but higher potential for cache conflicts
- each thread can monopolize the caches — no cooperation
  - potential for cache thrashing
  - can be bad for memory-intensive workloads
  - remember: hyperthreading can be disabled