Multiprocessor Systems

Operating Systems

Oleg Goldshmidt

ogoldshmidt@computer.org

Lecture 8

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: $c \approx 20\,\mathrm{cm}/\,\mathrm{ns}$ 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

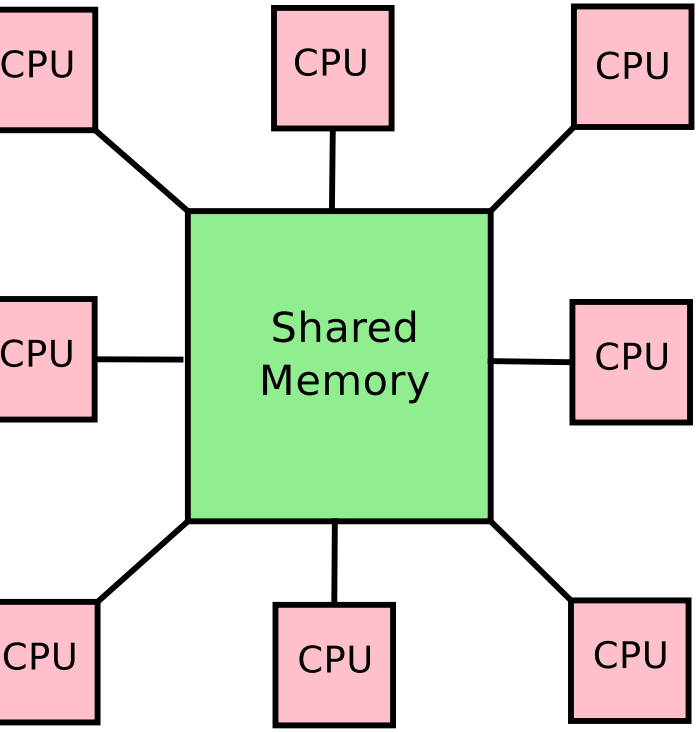
Solution: Parallelization

- 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
- difficulty: parallel programming

Stored-Program Multiprocessors

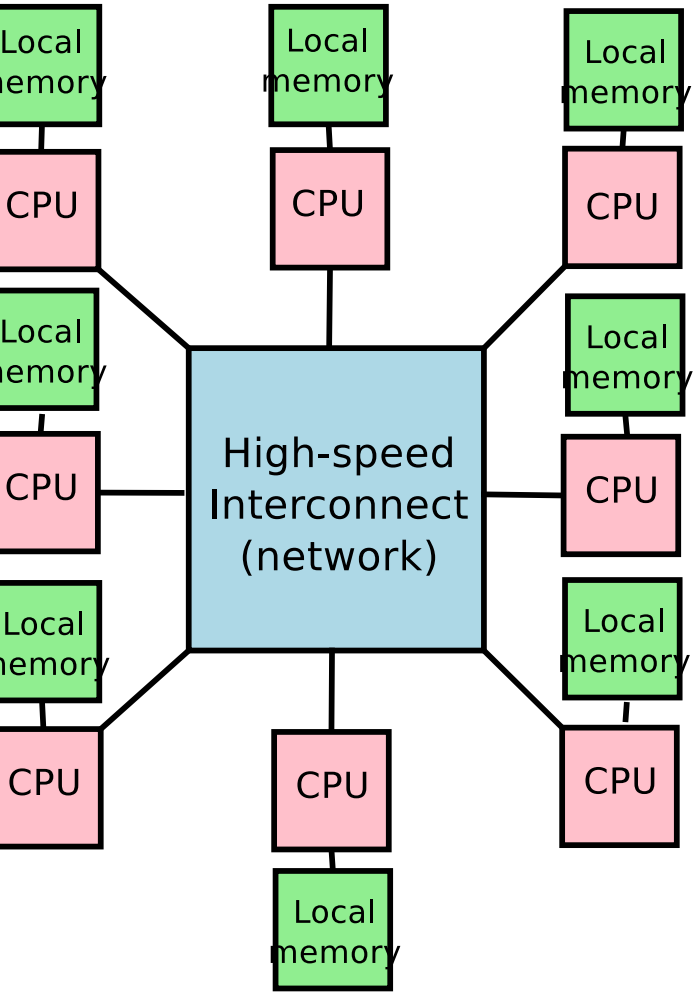
- separation of CPUs and memory
- how CPUs access memory is central to what follows
  - where istructions and data live
  - how fast can the CPU access the data
  - caching
  - shared memory
- interconnect between CPUs and memory, its properties are crucial
  - topology and connectivity
  - bandwidth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

Shared-Memory Multiprocessors



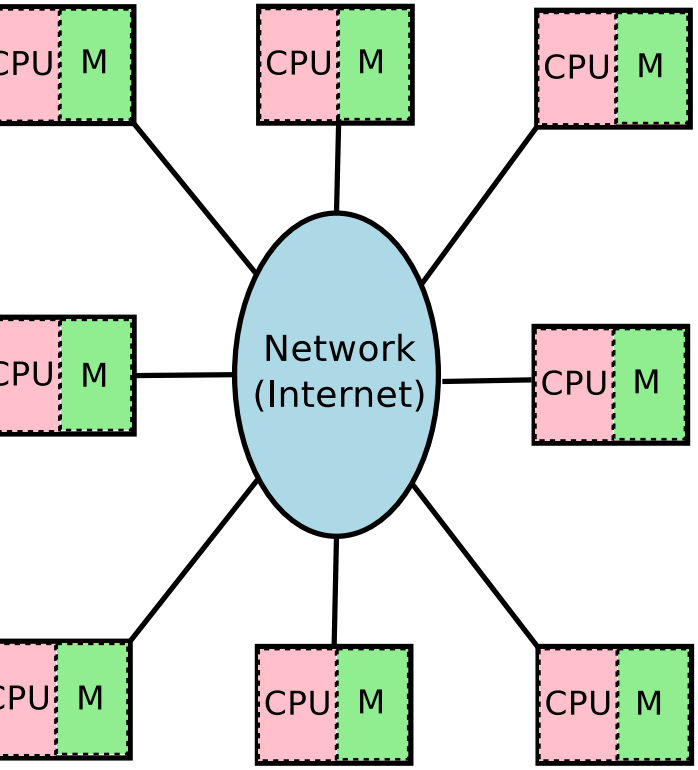
- between 2 and hundreds of CPUs
- all have access to the entire physical memory
- LOAD and STORE individual words

Message-Passing Multiprocessors



- $10^2 \div 10^5$ CPUs
- each CPU has its own memory, inaccessible by others

Distributed Multiprocessors



- potentially huge number of nodes
- each node is a complete system
- very similar to message-passing

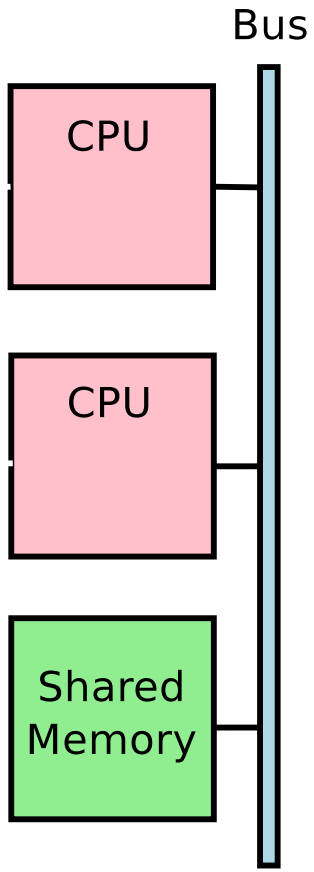
Shared-Memory MP Quirks

- 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

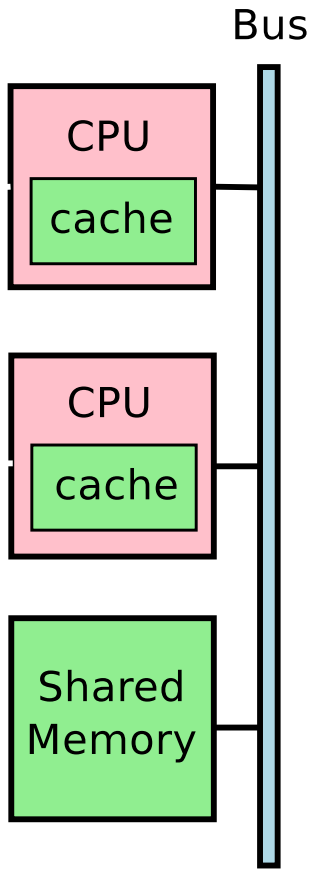
- a regular OS for the most part
  - system calls
  - memory management
  - file systems
  - I/O
- but not quite ordinary
  - process synchronization
  - resource management
  - scheduling

hared-Memory MP Hardware: UMA



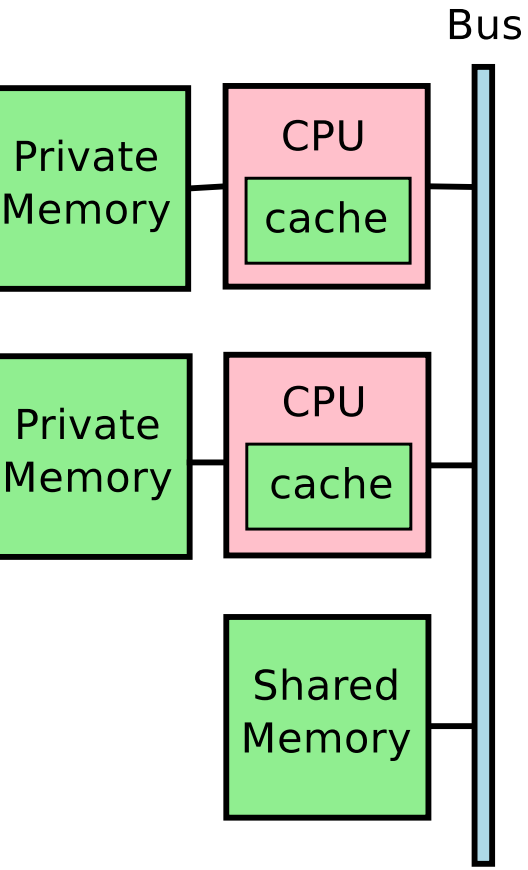
- CPUs share a bus for communication
  - contention limited by the bus bandwidth — non-scalable beyond a few CPUs

hared-Memory MP Hardware: UMA



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

hared-Memory MP Hardware: UMA

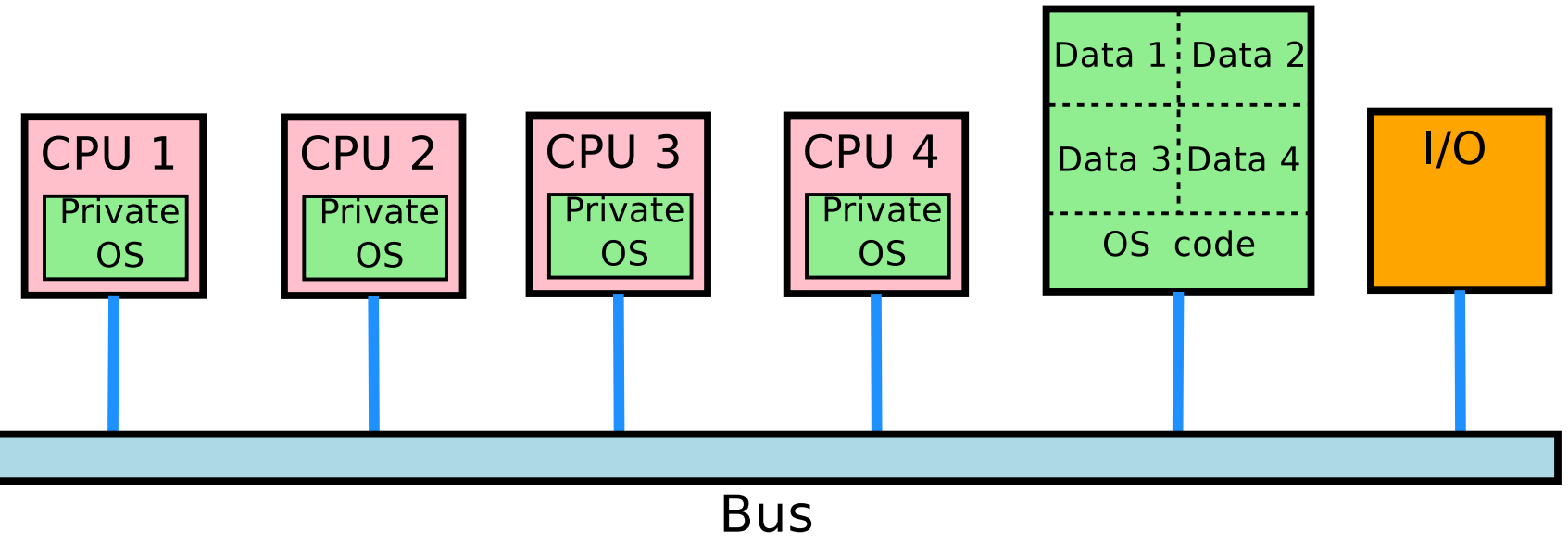


- CPUs share a bus for communication
  - contention limited by the bus bandwidth — non-scalable beyond a few CPUs
- solution: add caches to CPUs

NUMA Multiprocessors

- single-bus UMA multiprocessors are not very scalable
  - NUMA — give up the uniform memory access time
- to go beyond 100 CPUs something has got to give
- 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

Multiprocessors with Private OS

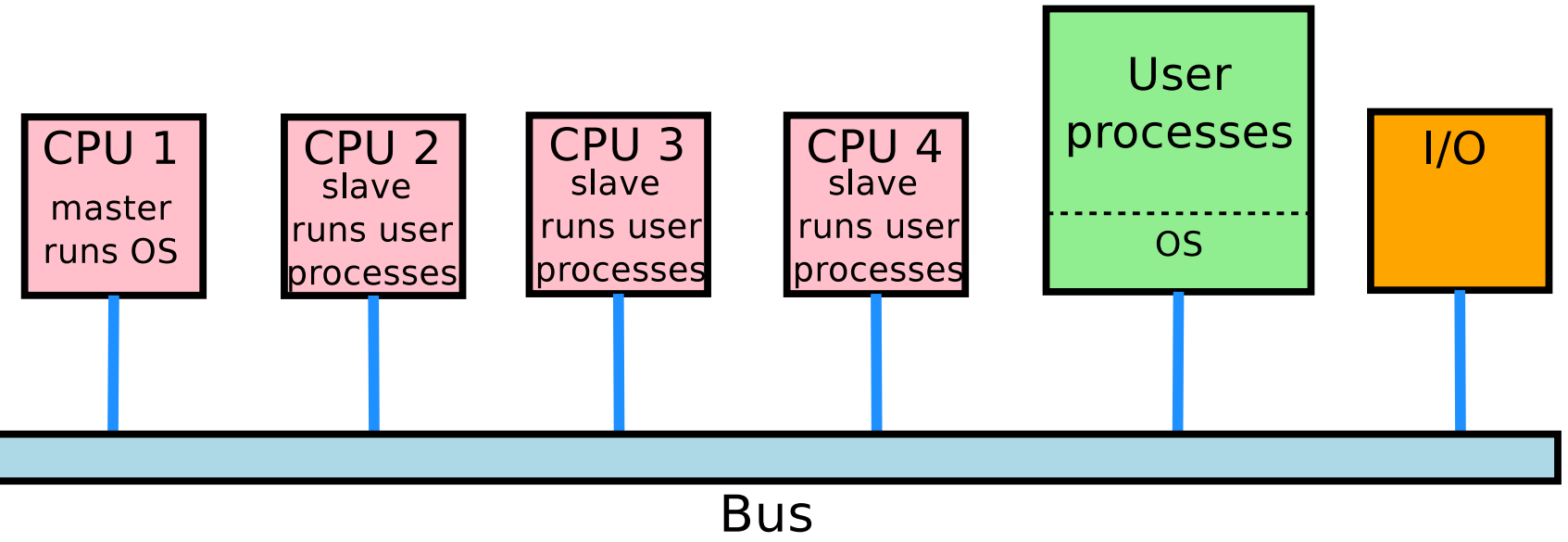


- each CPU has private memory and a private copy of OS
- effectively n independent computers
- optimization: share OS code

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

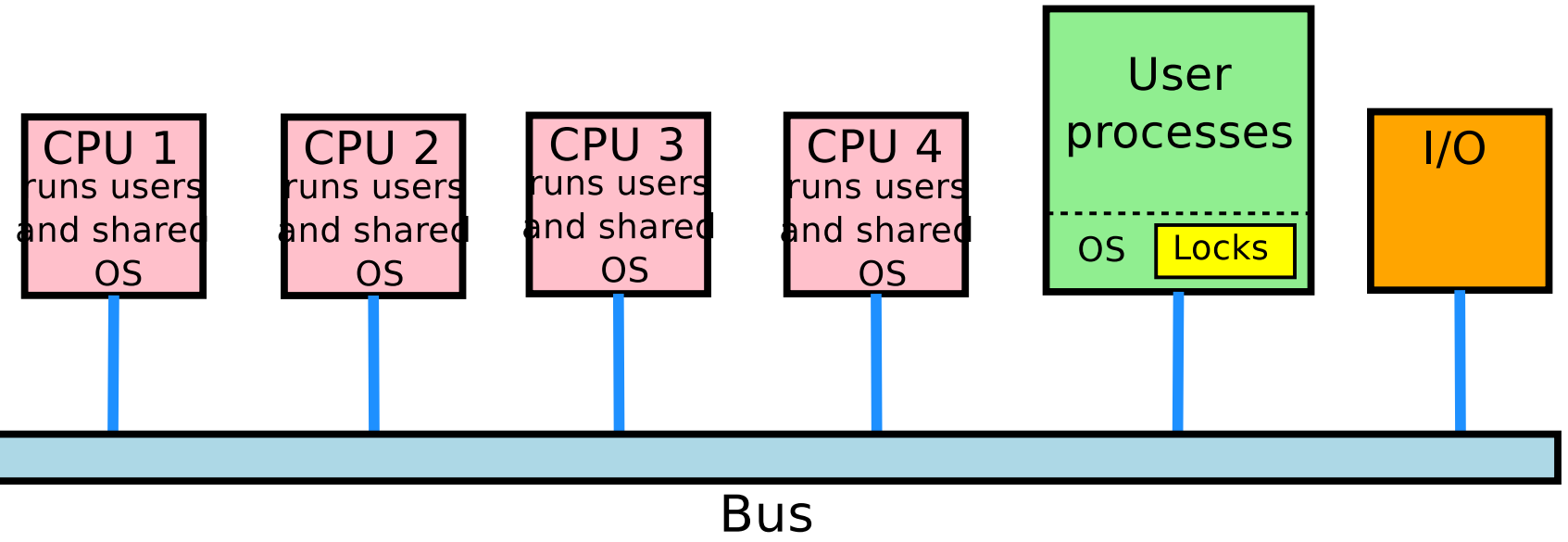
Master-Slave Multiprocessors II

- 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 (Linux: BKL)
  - 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, brute-force solutions (e.g., disabling interrupts) as on UP
  - interrupts are disabled on the current CPU only

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

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 on UP machines
- 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?

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

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