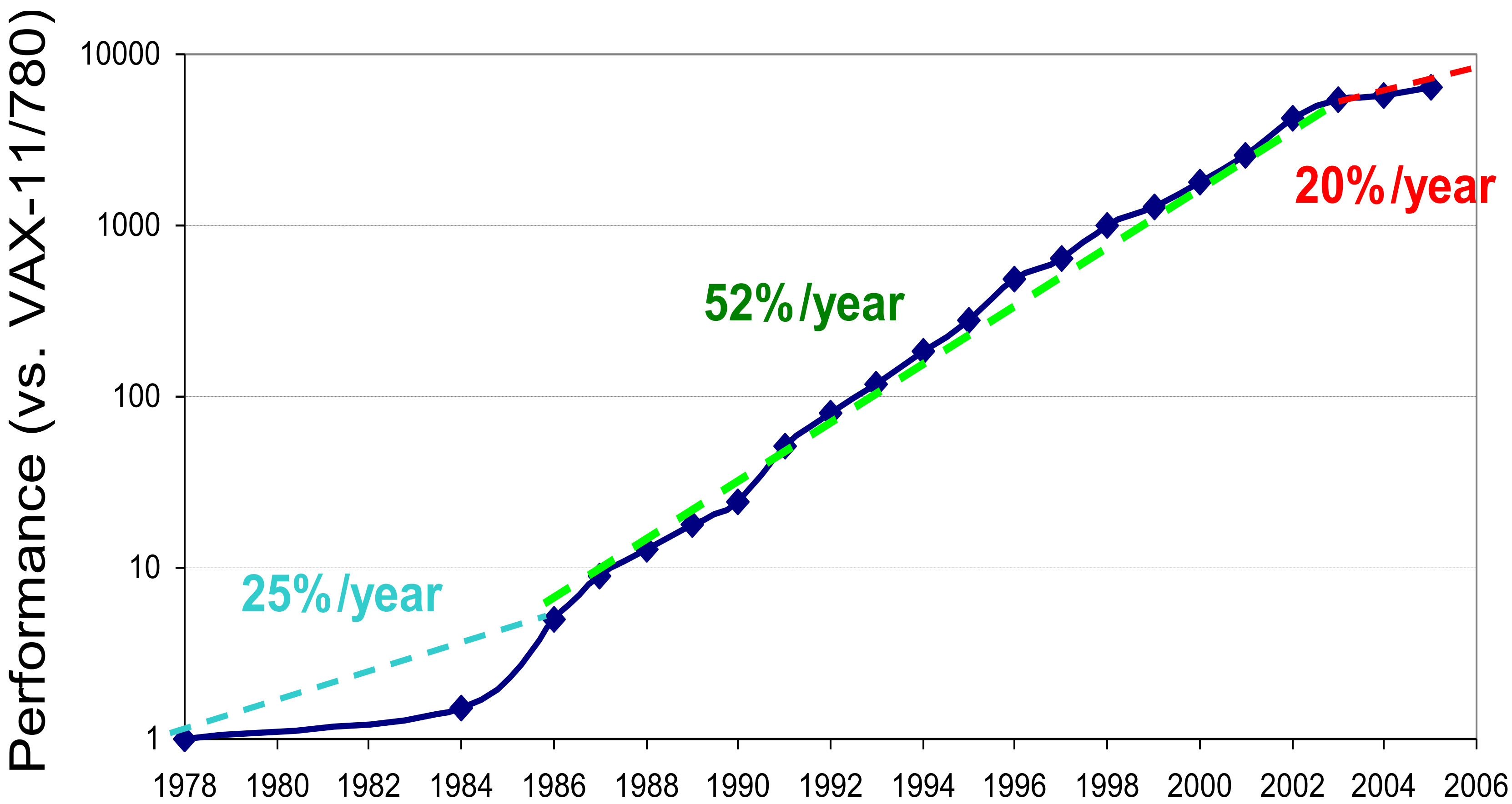
# Concurrent Programing: Why you should care, deeply

Don Porter
Portions courtesy Emmett Witchel

# Uniprocessor Performance Not Scaling

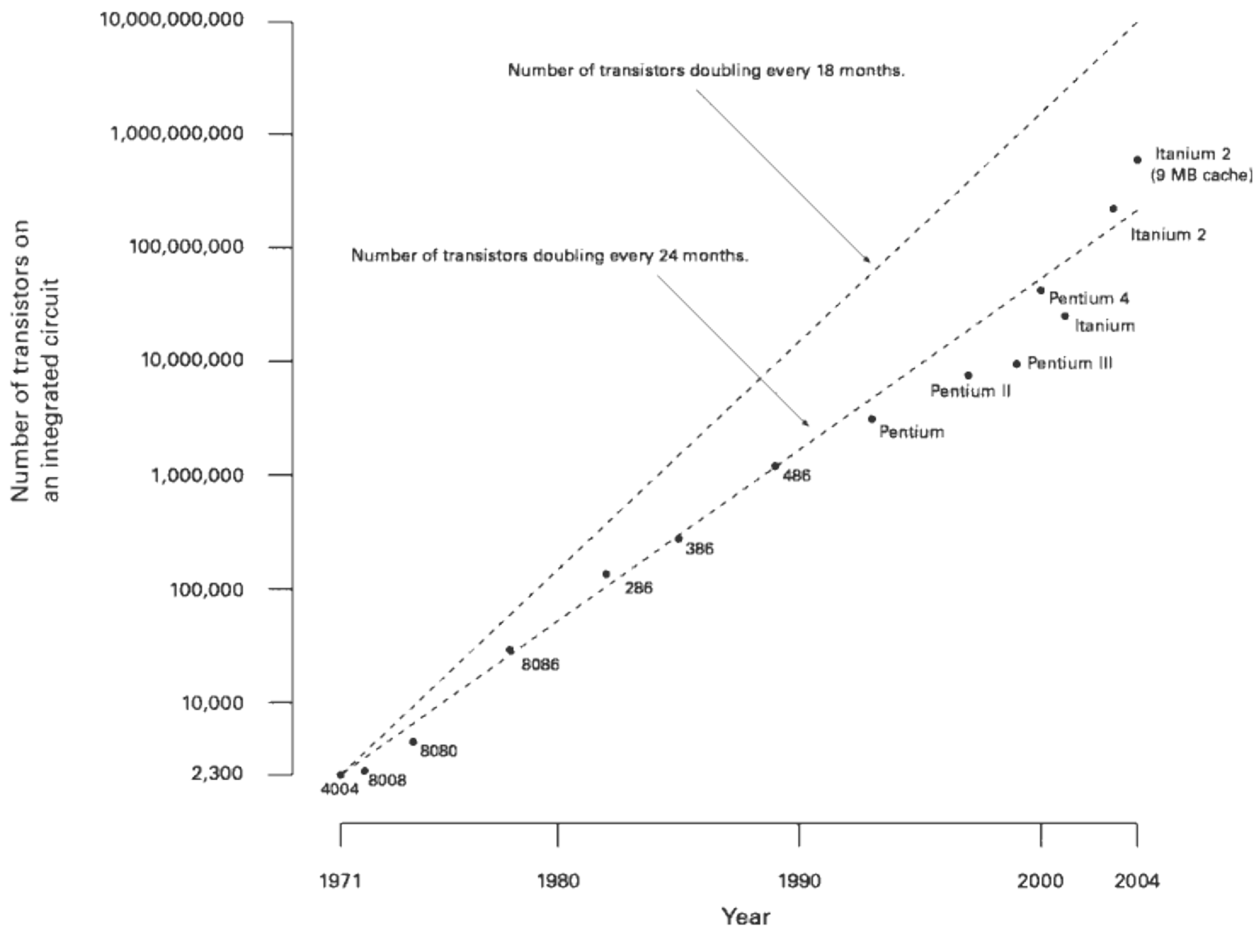



Graph by Dave Patterson

# Power and heat lay waste to processor makers

- Intel P4 (2000-2007)
  - 1.3GHz to 3.8GHz, 31 stage pipeline
  - "Prescott" in 02/04 was too hot. Needed 5.2GHz to beat 2.6GHz Athalon
- Intel Pentium Core, (2006-)
  - 1.06GHz to 3GHz, 14 stage pipeline
  - Based on mobile (Pentium M) micro-architecture
    - Power efficient
- 2% of electricity in the U.S. feeds computers
  - Doubled in last 5 years

# What about Moore's law?



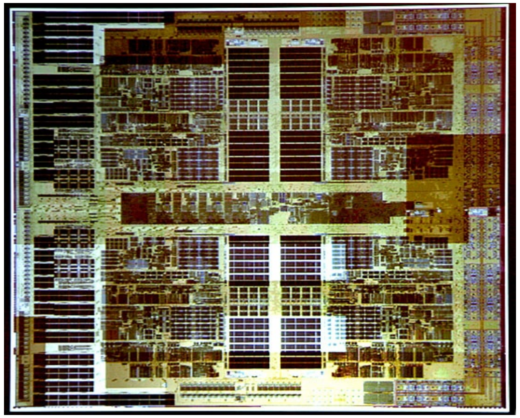

- Number of transistors double every 24 months
  - Not performance!

# Architectural trends that favor multicore

- Power is a first class design constraint
  - Performance per watt the important metric
- Leakage power significant with small transisitors
  - Chip dissipates power even when idle!
- Small transistors fail more frequently
  - Lower yield, or CPUs that fail?
- Wires are slow
  - Light in vacuum can travel ~1m in 1 cycle at 3GHz
  - Motivates multicore designs (simpler, lower-power cores)
- Quantum effects
- Motivates multicore designs (simpler, lower-power cores)

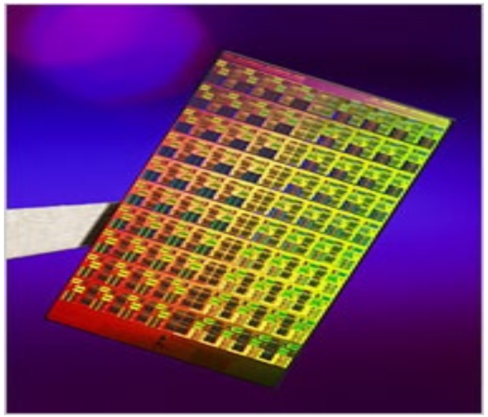
# *Multicores are here, and coming fast!*

4 cores in 2007 | 16 cores in 2009 | 80 cores in 20??



AMD Quad Core


Sun Rock



Intel TeraFLOP

"[AMD] quad-core processors … are just the beginning…."
http://www.amd.com

"Intel has more than 15 multi-core related projects underway"
http://www.intel.com

# Multicore programming will be in demand

- Hardware manufacturers betting big on multicore
- Software developers are needed
- Writing concurrent programs is not easy
- You will learn how to do it in this class

# Concurrency Problem

- Order of thread execution is non-deterministic
  - Multiprocessing
    - A system may contain multiple processors ➔ cooperating threads/processes can execute simultaneously
  - Multi-programming
    - Thread/process execution can be interleaved because of time-slicing
- Operations often consist of multiple, visible steps
  - Example: x = x + 1 is not a single operation
    - read x from memory into a register
    - increment register
    - store register back to memory

Thread 2
read
increment
store

- Goal:
  - Ensure that your concurrent program works under ALL possible interleaving

# Questions

- Do the following either completely succeed or completely fail?
- Writing an 8-bit byte to memory
  - A. Yes B. No
- Creating a file
  - A. Yes B. No
- Writing a 512-byte disk sector
  - A. Yes B. No

# Sharing among threads increases performance…

```
int a = 1, b = 2;
main() {
    CreateThread(fn1, 4);
    CreateThread(fn2, 5);
}
fn1(int arg1) {
    if(a) b++;
}
fn2(int arg1) {
    a = arg1;
}
```

What are the values of a & b at the end of execution?

# Sharing among theads increases performance, but can lead to problems!!

```
int a = 1, b = 2;
main() {
    CreateThread(fn1, 4);
    CreateThread(fn2, 5);
}
fn1(int arg1) {
    if(a) b++;
}
fn2(int arg1) {
    a = 0;
}
```

What are the values of a & b at the end of execution?

# Some More Examples

- What are the possible values of x in these cases?

```
Thread1: x = 1;                Thread2: x = 2;
```

```
Initially y = 10;
Thread1: x = y + 1;            Thread2: y = y * 2;
```

```
Initially x = 0;
Thread1: x = x + 1;            Thread2: x = x + 2;
```

# Critical Sections

- A critical section is an abstraction
  - Consists of a number of consecutive program instructions
  - Usually, crit sec are mutually exclusive and can wait/signal
    - Later, we will talk about atomicity and isolation
- Critical sections are used frequently in an OS to protect data structures (e.g., queues, shared variables, lists, …)
- A critical section implementation must be:
  - Correct: the system behaves as if only 1 thread can execute in the critical section at any given time
  - Efficient: getting into and out of critical section must be fast. Critical sections should be as short as possible.
  - Concurrency control: a good implementation allows maximum concurrency while preserving correctness
  - Flexible: a good implementation must have as few restrictions as practically possible

# The Need For Mutual Exclusion

- Running multiple processes/threads in parallel increases performance
- Some computer resources cannot be accessed by multiple threads at the same time
  - E.g., a printer can't print two documents at once
- Mutual exclusion is the term to indicate that some resource can only be used by one thread at a time
  - Active thread excludes its peers
- For shared memory architectures, data structures are often mutually exclusive
  - Two threads adding to a linked list can corrupt the list

# Exclusion Problems, Real Life Example

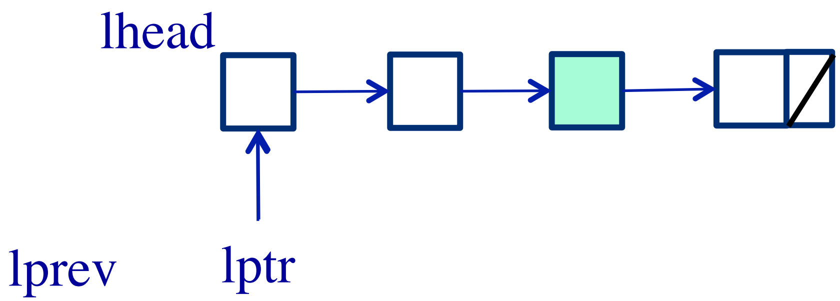
- Imagine multiple chefs in the same kitchen
  - Each chef follows a different recipe
- Chef 1
  - Grab butter, grab salt, do other stuff
- Chef 2
  - Grab salt, grab butter, do other stuff
- What if Chef 1 grabs the butter and Chef 2 grabs the salt?
  - Yell at each other (not a computer science solution)
  - Chef 1 grabs salt from Chef 2 (preempt resource)
  - Chefs all grab ingredients in the same order
    - Current best solution, but difficult as recipes get complex
    - Ingredient like cheese might be sans refrigeration for a while

# The Need To Wait

- Very often, synchronization consists of one thread waiting for another to make a condition true
  - Master tells worker a request has arrived
  - Cleaning thread waits until all lanes are colored
- Until condition is true, thread can sleep
  - Ties synchronization to scheduling
- Mutual exclusion for data structure
  - Code can wait (await)
  - Another thread signals (notify)

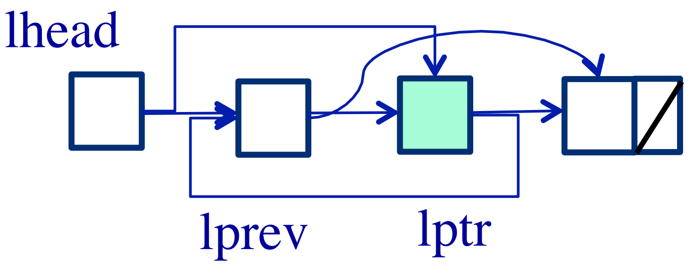
# Example 2: Traverse a singly-linked list

- Suppose we want to find an element in a singly linked list, and move it to the head
- Visual intuition:



# Example 2: Traverse a singly-linked list

- Suppose we want to find an element in a singly linked list, and move it to the head
- Visual intuition:



# Even more real life, linked lists

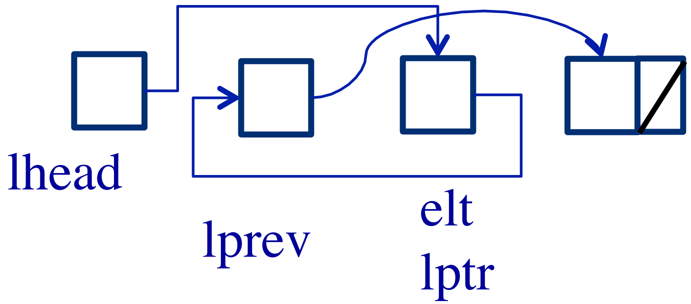
```
lprev = NULL;
for(lptr = lhead; lptr; lptr = lptr->next) {
   if(lptr->val == target){
      // Already head?, break
      if(lprev == NULL) break;
      // Move cell to head
      lprev->next = lptr->next;
      lptr->next = lhead;
      lhead = lptr;
      break;
   }
   lprev = lptr;
}
```

- Where is the critical section?

# Even more real life, linked lists

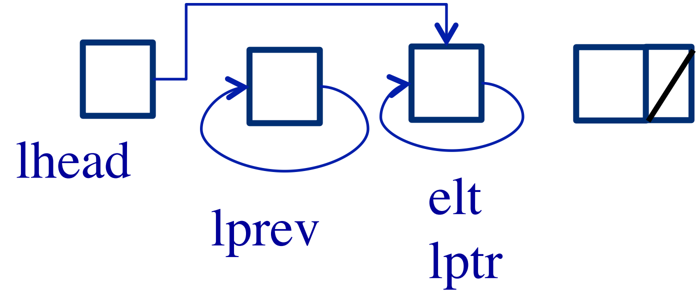
Thread 1

```
// Move cell to head
lprev->next = lptr->next;
lptr->next = lhead
lhead = lptr;
```




Thread 2

```
lprev->next = lptr->next;
lptr->next = lhead;
lhead = lptr;
```




- A critical section often needs to be larger than it first appears
  - The 3 key lines are not enough of a critical section

# Even more real life, linked lists

Thread 1

```
if(lptr->val == target){
      elt = lptr;
      // Already head?, break
      if(lprev == NULL) break;
      // Move cell to head
      lprev->next = lptr->next;
      // lptr no longer in list
```

Thread 2

```
for(lptr = lhead; lptr;
   lptr = lptr->next) {
   if(lptr->val == target){
```

- Putting entire search in a critical section reduces concurrency, but it is safe.

# Safety and Liveness

- *Safety property* : "nothing bad happens"
  - holds in every finite execution prefix
    - Windows™ never crashes
    - a program never terminates with a wrong answer

- *Liveness property*: "something good eventually happens"
  - no partial execution is irremediable
    - Windows™ always reboots
    - a program eventually terminates

- Every property is a combination of a safety property and a liveness property - (Alpern and Schneider)

# Safety and liveness for critical sections

- At most k threads are concurrently in the critical section
  - A. Safety
  - B. Liveness
  - C. Both
- A thread that wants to enter the critical section will eventually succeed
  - A. Safety
  - B. Liveness
  - C. Both
- Bounded waiting: If a thread *i* is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section (only 1 thread is alowed in at a time) before thread *i*'s request is granted.
  - A. Safety B. Liveness C. Both