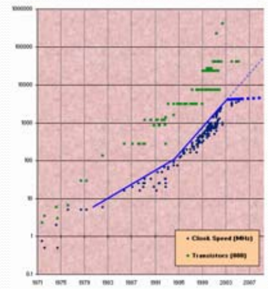
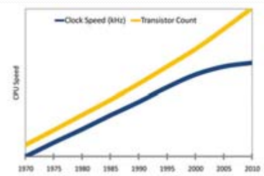
Threads

Ken Birman

Moore's Law and CPU speed



- Transistor count still rising according to Moore's Law
- Clock speed flattening



What this tells us?

- It is getting harder to just speed up a chip
  - One issue: the wires on the chip get very thin
  - But more serious: heat becomes a problem
- Heat dissipated by a chip rises roughly as the square of the clock speed!
  - So: if the clock is four times as fast…
  - … the chip produces sixteen times as much heat!
- Chips are at risk of *burning up*

Multicores are the answer

- Multicores are the answer to keeping up with increasing CPU performance despite:
  - The **memory wall** (gap between CPU and memory speeds)
  - The **ILP wall** (not enough instruction-level parallelism to keep the CPU busy)
  - The **power wall** (higher clock speeds require more power and create thermal problems)
- Consequence:
  - Single-thread performance doesn't improve…

…but we can put more cores on a chip

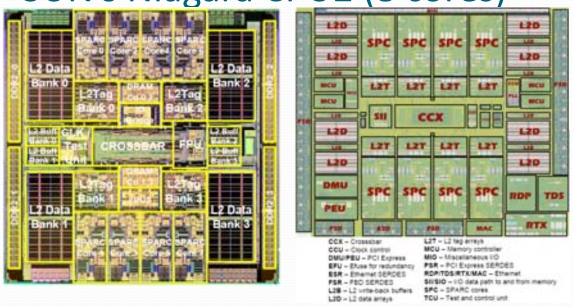
Why?

- Suppose you have four cores and each of them runs at 1/4th the speed of some hypothetical fast chip
  - Your power consumption turns out to be *less* than four times the power consumption of one *slow* chip
    - Because they share a lot of hardware
  - And a *fast* chip would generate easily ten times as much heat as four slow chips, to do the same amount of work
- But there's one "issue"
  - Programs need to use all those processors!

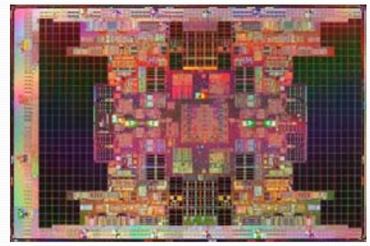
Multicores are everywhere

- **Dual-core** commonplace in laptops
- **Quad-core** in desktops
- **Dual quad-core** in servers
- All major chip manufacturers produce multicore CPUs
  - **SUN Niagara** (8 cores, 32 concurrent threads)
  - **Intel Xeon** (4 cores)
  - **AMD Opteron** (4 cores)
  - …

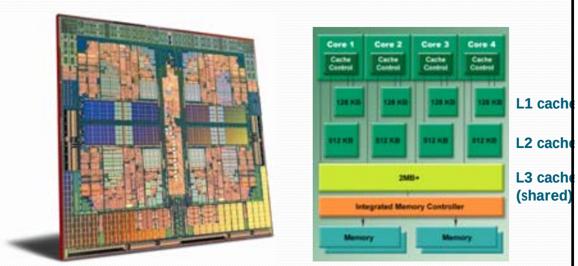
SUN's Niagara CPU2 (8 cores)



Intel Tukwila Itanium Chip (4 cores)



AMD Opteron (4 cores)



Threads

- On Tuesday we discussed processes that run in separate address spaces
  - Two processes can execute the same program, but they will have distinct registers, stacks, data regions. Only the code is shared.
- A *thread* (sometimes called a *lightweight process*) runs within a single process. Each thread
  - Has its own registers and its own stack
  - Threads share *data* with other threads in same process

Threads

- With threads we can write programs that
  - Stay busy when doing input/output (I/O) operations
  - Exploit multicore parallelism
- But exploiting threads isn't trivial

Applications that use threads

- Web browsers
  - Each frame (mini-window) might have its own thread
- Web servers
  - Each request could be done with a separate thread.
- Graphics or gaming application
  - Each object could be rendered by its own thread

Case for Parallelism

```
main()
  read_data()
  for(all data)
    compute();
    write_data();
  endfor
```

```
main()
  read_data()
  for(all data)
    compute();
    CreateProcess(write_data());
  endfor
```

Case for Parallelism

Consider the following code fragment

```
for(k = 0; k < n; k++)
  a[k] = b[k] * c[k] + d[k] * e[k];

CreateProcess(fn, 0, n/2);
CreateProcess(fn, n/2, n);
fn(l, m)
  for(k = l; k < m; k++)
    a[k] = b[k] * c[k] + d[k] * e[k];
```

Case for Parallelism

Consider a Web server

create a number of process, and for each process do

- get network message from client
- get URL data from disk
- compose response
- send response

Threads and Processes

- Most operating systems adopt the view that processes are *containers* in which threads execute
- Initially, a process is created with one thread
  - It runs the main() procedure
- But you can create additional threads
  - They start by running any procedure you like
  - Then they can create additional threads, or terminate
- A process is expensive to create (fork/exec)
- A thread is very cheap to create

Thread Creation: Java syntax

Create a new thread, start it. It will be initialized via a call to the constructor and then later, but we can't predict precisely when, run() will be invoked.

```
public class Main {
    Thread A = new ThreadDemo("my name is A"), B = new ThreadDemo("my name is B");
     ....
     A.start();  B.start();
     ....
}
public class ThreadDemo extends Thread {
      private final String name;                // Cannot later be changed
      public void ThreadDemo(String s) {        // Constructor initializes stuff
            this.name = s;
      }
      public void run()
      {
            system.out.println("Thread <"+this.name+"> is running");
      }
}
```

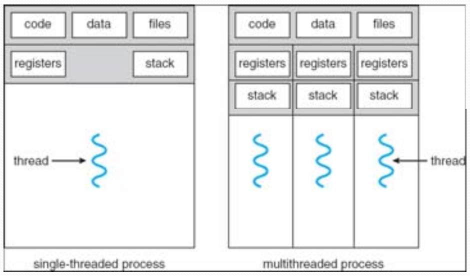
Prints "Thread <my name is A> is running", "Thread <my name is B> is running" (in some order), then exits.

Thread Creation in Java

- Once "started", the "run" procedure will be invoked

```
public void run()
{
      system.out.println("Thread <"+this.name+"> is running");
}
```

Multithreaded Processes



Cooperative (non-premptive) threads

Each thread runs until *it* decides to give up the CPU

```
main()
{
    tid t1 = CreateThread(fn, arg);
    …
    Yield(t1);
}
fn(int arg)
{
    …
    Yield(any);
}
```

Cooperative Threads

- Cooperative threads use non-preemptive scheduling
  - Definition: one thread runs at a time, until it pauses by “yielding” the CPU
- Advantages:
  - Simple
    - Scientific apps
- Disadvantages:
  - For badly written code
- Scheduler gets invoked only when Yield is called
- A thread should yield the processor when it blocks for I/O, e.g. to read from a file or the keyboard

Non-Cooperative Threads

- No explicit control passing among threads
- Rely on a scheduler to decide which thread to run
- A thread can be pre-empted at any point
- Often called pre-emptive threads
- Most modern thread packages use this approach

User-Level Threads

- For speed, implement threads at the user level
- A user-level thread is managed by the run-time system
  - user-level code that is linked with your program
- Each thread is represented simply by:
  - PC
  - Registers
  - Stack
  - Small control block
- All thread operations are at the user-level:
  - Creating a new thread
  - switching between threads
  - synchronizing between threads

User-Level Threads

- User-level threads
  - the thread scheduler is part of a library, outside the kernel
  - thread context switching and scheduling is done by the library
  - Can either use cooperative or pre-emptive threads
    - cooperative threads are implemented by:
      - CreateThread(), DestroyThread(), Yield(), Suspend(), etc.
    - pre-emptive threads are implemented with a timer (signal)
      - where the timer handler decides which thread to run next

Example User Thread Interface

t = thread_fork(initial context)
  create a new thread of control

thread_stop()
  stop the calling thread, sometimes called thread_block

thread_start(t)
  start the named thread

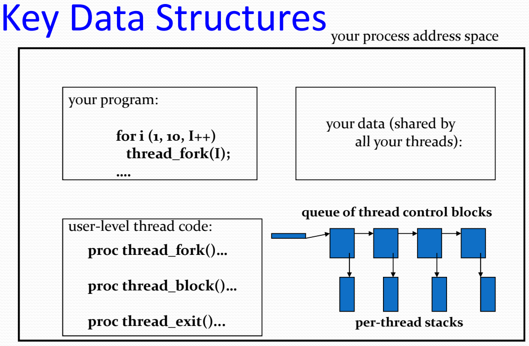
thread_yield()
  voluntarily give up the processor

thread_exit()
  terminate the calling thread, sometimes called thread_destroy

Kernel threads

- User threads can run with even a single CPU
  - We simply use *context switching* when one blocks and another starts, just as the kernel does to run a process
- But modern machines often have multiple cores
  - This leads to the idea of a *kernel thread*
  - Basically, a thread with its own core (CPU)
- User sees the same thread API, but each kernel thread can switch among some set of user threads

Key Data Structures



User-Level vs. Kernel Threads

User-Level
- Managed by application
- Kernel not aware of thread
- Context switching cheap
- Create as many as needed
- Must be used with care

Kernel-Level
- Managed by kernel
- Consumes kernel resources
- Context switching expensive
- Number limited by kernel resources
- Simpler to use

Key issue: kernel threads provide virtual processors to user-level threads, but if all of kthreads block, then all user-level threads will block *even* if the program logic allows them to proceed

Common tradeoffs

- Threads do have costs
  - Especially, the stack space required
  - With many threads, this becomes a serious burden
- Modern machines have multiple cores… but rarely more than 2. At most 8 or 16.
  - So we can't just have one kernel thread per user thread
  - Context switching costs can become a big overhead

Thread Hazards: Much like our race condition from last week!

```
int a = 1, b = 2, w = 2;
main() {
   CreateThread(fn, 4);
   CreateThread(fn, 4);
   while(w) ;
}
fn() {
  int v = a + b;
  w--;
}
```

Concurrency Problems

A statement like w-- in C (or C++) is implemented by several machine instructions:

```
mov    w,r4
sub    $1,r4
mov    r4,w
```

Now, imagine the following sequence, what is the value of w?

mov w,r4	mov r4,-(sp)
________	mov w,r4
________	sub $-1,w
________	mov r4,w
sub $1,r1	mov (sp)+,r4
mov r4, w	

Threads share global memory

- When a process contains multiple threads, they have
  - Private registers and stack memory (the *context switching* mechanism needs to save and restore registers when switching from thread to thread)
  - Shared access to the remainder of the process "state"
- This can result in *race conditions*

Two threads, one counter

Popular web server

- Uses multiple threads to speed things up.
- Simple shared state error:
  - each thread increments a shared counter to track number of hits

```
...
hits = hits + 1;
...
```

- What happens when two threads execute concurrently?

some slides taken from Mendel Rosenblum's lecture at Stanford

Shared counters

- Possible result: lost update!

hits = 0

time

T1	T2
read hits (0)	
	read hits (0)
hits = 0 + 1	
	hits = 0 + 1

hits = 1

- One other possible result: everything works.
  ⇒ Difficult to debug
- Called a "race condition"

Race conditions: Improved defn

- *A timing dependent error involving shared state*
  - Whether it happens depends on how threads scheduled
  - In effect, once thread A starts doing something, it needs to "race" to finish it because if thread B looks at the shared memory region before A is done, it may see something inconsistent
- Hard to detect:
  - All possible schedules have to be safe
    - Number of possible schedule permutations is huge
    - Some bad schedules? Some that will work sometimes?
  - they are intermittent
    - Timing dependent = small changes can hide bug

Scheduler assumptions

Process a:
```
while(i < 10)
    i = i +1;
print "A won!";
```

Process b:
```
while(i > -10)
    i = i - 1;
print "B won!";
```

If i is shared, and initialized to 0

- Who wins?
- Is it guaranteed that someone wins?
- What if both threads run on identical speed CPU
  - executing in parallel

Scheduler Assumptions

- Normally we assume that
  - A scheduler always gives every executable thread opportunities to run
    - In effect, each thread makes *finite progress*
  - But schedulers aren't always fair
    - Some threads may get more chances than others
  - To reason about worst case behavior we sometimes think of the scheduler as an adversary trying to "mess up" the algorithm

Critical Section Goals

- Threads do some stuff but eventually *might* try to access shared data

time

T1

CSEnter();
Critical section
CSExit();

T1

T2

CSEnter();
Critical section
CSExit();

T2

Critical Section Goals

- Perhaps they loop (perhaps not!)

T1

CSEnter();
Critical section
CSExit();

T1

T2

CSEnter();
Critical section
CSExit();

T2

Critical Section Goals

- We would like
  - Safety: No more than one thread can be in a critical section at any time.
  - Liveness: A thread that is seeking to enter the critical section will eventually succeed
  - Fairness: If two threads are both trying to enter a critical section, they have equal chances of success
- ... in practice, fairness is rarely guaranteed

Solving the problem

- A first idea:
  - Have a boolean flag, *inside*. Initially false.

```
CSEnter()
{
    while(inside) continue;
    inside = true;
}
```

Code is unsafe: thread 0 could finish the while test when inside is false, but then 1 might call CSEnter() before 0 can set inside to true!

```
    inside = false;
}
```

- Now ask:
  - Is this Safe? Live? Fair?

Solving the problem: Take 2

- A different idea (assumes just two threads):
  - Have a boolean flag, *inside[i]*. Initially false.

```
CSEnter(int i)
{
    inside[i] = true;
    while(inside[i^1]) continue;
}
```

Code isn't live: with bad luck, both threads could be looping, with 0 looking at 1, and 1 looking at 0

```
    Inside[i] = false;
}
```

- Now ask:
  - Is this Safe? Live? Fair?

Solving the problem: Take 3

- Another broken solution, for two threads
  - Have a turn variabl

Code isn't live: thread 1 can't enter unless thread 0 did first, and vice-versa. But perhaps one thread needs to enter many times and the other fewer times, or not at all

```
CSEnter(int i)
{
   while(turn != i) continue;
}
```

```
   turn = i ^ 1;
}
```

- Now ask:
  - Is this Safe? Live? Fair?

A solution that works

- Deker's Algorithm (book: Section 7.2.1.3)

```
CSEnter(int i)
{
   int J = i^1;
   inside[i] = true;
   turn = J;
   while(inside[J] && turn == J)
       continue;
}
```

```
CSExit(int i)
{
   Inside[i] = false;
}
```

Why does it work?

- Safety: Suppose thread 0 is in the CS.
  - Then inside[0] is true.
  - If thread 1 was simultaneously trying to enter, then turn must equal 0 and thread 1 waits
  - If thread 1 tries to enter "now", it *sets* turn to 0 and waits
- Liveness: Suppose thread 1 wants to enter and can't (stuck in while loop)
  - Thread 0 will eventually exit the CS
  - When inside[0] becomes false, thread 1 can enter
  - If thread 0 tries to reenter immediately, it sets turn=1 and hence will wait politely for thread 1 to go first!