### From Processes to Threads

Don Porter Portions courtesy Emmett Witchel

### Processes, Threads and Processors

- Hardware can execute N instruction streams at once
   > Uniprocessor, N==1
  - > Dual-core, N==2
- Sun's Niagara T2 (2007) N == 64, but 8 groups of 8
- An OS can run 1 process on each processor at the same time
  - $\succ$  Concurrent execution increases performance
- An OS can run 1 thread on each processor at the same time

### Processes and Threads

- Process abstraction combines two concepts
   Concurrency
   Each process is a sequential execution stream of instructions
  - Protection
  - FIDIECIION
  - Each process defines an address space
     Address space identifies all addresses that can be touched by the program

### Threads

- > Key idea: separate the concepts of concurrency from protection
- > A thread is a sequential execution stream of instructions
- A process defines the address space that may be shared by multiple threads
- Threads can execute on different cores on a multicore CPU (parallelism for performance) and can communicate with other threads by updating memory

# Example Redux Virtual Address Space hello heap stk1 stk2 libc.so Linux 0 0xffffffff • 2 threads requires 2 stacks in the process • No problem! • Kernel can schedule each thread separately > Possibly on 2 CPUs > Requires some extra bookkeeping

### The Case for Threads

Consider the following code fragment for(k = 0; k < n; k++) a[k] = b[k] \* c[k] + d[k] \* e[k];

Is there a missed opportunity here? On a Uni-processor? On a Multi-processor?

### The Case for Threads

Consider a Web server get network message (URL) from client get URL data from disk compose response send response

How well does this web server perform?

### Programmer's View

void fn1(int arg0, int arg1, ...) {...}

main() {

tid = CreateThread(fn1, arg0, arg1, ...);

}

At the point CreateThread is called, execution continues in parent thread in main function, and execution starts at fn1 in the child thread, both in parallel (concurrently)

### Introducing Threads

- A thread represents an abstract entity that executes a sequence of instructions > It has its own set of CPU registers
  - > It has its own stack > There is no thread-specific heap or data segment (unlike process)
- Threads are lightweight
   Creating a thread more efficient than creating a process. Communication between threads easier than btw. processes.
  - > Context switching between threads requires fewer CPU cycles and
- memory references than switching processes.
- Threads only track a subset of process state (share list of open files, pid, ...)
- Examples:
  - > OS-supported: Windows' threads, Sun's LWP, POSIX threads > Language-supported: Modula-3, Java These are possibly going the way of the Dodo

## Context switch time for which entity is greater?

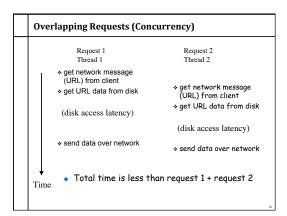
- 1. Process
- 2. Thread

### How Can it Help? • How can this code take advantage of 2 threads? for(k = 0; k < n; k++) a[k] = b[k] \* c[k] + d[k] \* e[k]; Rewrite this code fragment as: do\_mult(I, m) { for(k = I; k < m; k++) a[k] = b[k] \* c[k] + d[k] \* e[k]; 3 main() { CreateThread(do\_mult, 0, n/2); CreateThread(do\_mult, n/2, n);

What did we gain?

### How Can it Help?

- Consider a Web server Create a number of threads, and for each thread do \* get network message from client \* get URL data from disk \* send data over network
- What did we gain?



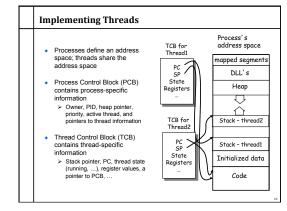
### Why threads? (summary)

- Computation that can be divided into concurrent chunks
  - > Same Instruction (or operation), Multiple Data (SIMD easy) > Harder to identify parallelism in more complex cases
- Overlapping blocking I/O with computation
  - > If my web server blocks on I/O for one client, why not work on another client's request in a separate thread?
  - > Other abstractions we won't cover (e.g., events)

### Threads have their own...?

- CPU
- Address space
- 3. PCB
- Stack 😳
- Registers 🙄

### Threads vs. Processes Processes Threads A thread has no data segment A process has code/data/heap & or heap other segments A thread cannot live on its own, There must be at least one it must live within a process thread in a process There can be more than one Threads within a process share thread in a process, the first code/data/heap, share I/O, but thread calls main & has the each has its own stack & process's stack reaisters If a thread dies, its stack is If a process dies, its resources reclaimed Inter-thread communication via are reclaimed & all threads die memory. Inter-process communication via Each thread can run on a OS and data copying. different physical processor Each process can run on a Inexpensive creation and different physical processor context switch Expensive creation and context switch

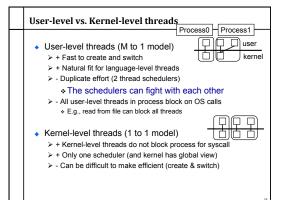


### Threads' Life Cycle

• Threads (just like processes) go through a sequence of start, ready, running, waiting, and done states



# Threads have the same scheduling states as processes 1. True 😳 2. False • In fact, OSes generally schedule threads to CPUs, not processes



### Languages vs. Systems

- Kernel-level threads have won for systems
   Linux, Solaris 10, Windows
- pthreads tend to be kernel-level threads
   User-level threads still used in some Java runtimes
- User tells JVM how many underlying system threads
   Default: 1 system thread
- Java runtime intercepts blocking calls, makes them nonblocking
- JNI code that makes blocking syscalls can block JVM
- JVMs are phasing this out because kernel threads are efficient enough and intercepting system calls is complicated
- Kernel-level thread vs. process
  - Each process requires its own page table & hardware state (significant on the x86)

### Editorial on User vs. Kernel threads

- There is a 25+ year history of debating user vs. kernel threads
- These discussions are couched in grand principles
- The real issue is simple: Performance!!
   > If the kernel implementation of thread context switching is
- If the kerner implementation of thread context switching is slow, everyone starts writing user-level thread packages
   Java did this for a while
- $\succ$  If the kernel implementation gets faster, everyone just uses kernel threads, since they are easier
- Java does this now, Linux 2.6 overhauled its thread implementation

### Latency and Throughput

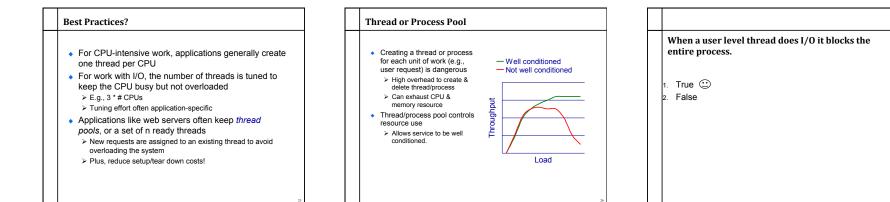
- Latency: time to complete an operation
- Throughput: work completed per unit time
- Multiplying vector example: reduced latency
- Web server example: increased throughput
- Consider plumbing
  - > Low latency: turn on faucet and water comes out
  - High bandwidth: lots of water (e.g., to fill a pool)
- What is "High speed Internet?"
  - > Low latency: needed to interactive gaming
  - $\succ$  High bandwidth: needed for downloading large files
  - Marketing departments like to conflate latency and bandwidth...

### Relationship between Latency and Throughput

- Latency and bandwidth only loosely coupled
   Henry Ford: assembly lines increase bandwidth without reducing latency
- My factory takes 1 day to make a Model-T ford.
   > But I can start building a new car every 10 minutes
- > At 24 hrs/day, I can make 24 \* 6 = 144 cars per day
- $\succ$  A special order for 1 green car, still takes 1 day
- $\succ$  Throughput is increased, but latency is not.
- Latency reduction is difficult
- Often, one can buy bandwidth
- E.g., more memory chips, more disks, more computers
   Big server farms (e.g., google) are high bandwidth

### Latency, Throughput, and Threads

- Can threads improve throughput?
   Yes, as long as there are parallel tasks and CPUs available
- Can threads improve latency?
   Yes, especially when one task might block on another task's
- Can threads harm throughput?
   Yes, each thread gets a time slice.
  - If # threads >> # CPUs, the %of CPU time each thread gets approaches 0
- Can threads harm latency?
   Yes, especially when requests are short and there is little I/O



# Lecture Summary

- Understand the distinction between a process and thread
- Understand the motivation for threads
- Kernel vs. User threads
- Concepts of Throughput vs. Latency
- Thread pools