

From Processes to Threads

Don Porter

Portions courtesy Emmett Witchel

Processes, Threads and Processors

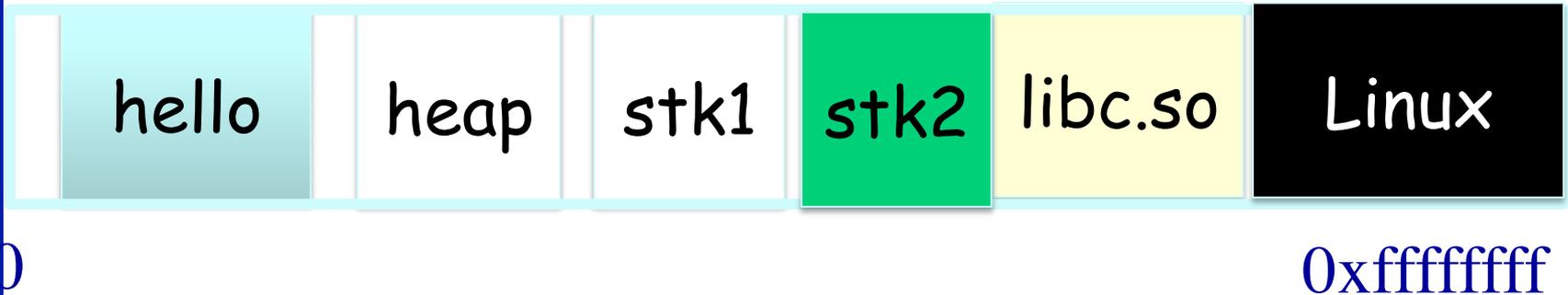
- ◆ Hardware can execute N instruction streams at once
 - Uniprocessor, $N=1$
 - Dual-core, $N=2$
 - Sun's Niagara T2 (2007) $N = 64$, but 8 groups of 8
- ◆ An OS can run 1 process on each processor at the same time
 - Concurrent execution increases performance
- ◆ An OS can run 1 thread on each processor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

Processes and Threads

- ◆ Process abstraction combines two concepts
 - Concurrency
 - ❖ Each process is a sequential execution stream of instructions
 - Protection
 - ❖ 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



- ◆ 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(l, m) {  
    for(k = l; k < m; k++)  
        a[k] = b[k] * c[k] + d[k] * e[k];  
}  
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?

Overlapping Requests (Concurrency)

Request 1
Thread 1

- ❖ get network message (URL) from client
- ❖ get URL data from disk

(disk access latency)

- ❖ send data over network

Request 2
Thread 2

- ❖ get network message (URL) from client
- ❖ get URL data from disk

(disk access latency)

- ❖ send data over network

Time

- ◆ Total time is less than request 1 + request 2

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...?

1. CPU
2. Address space
3. PCB
4. Stack 😊
5. Registers 😊

Threads vs. Processes

Threads

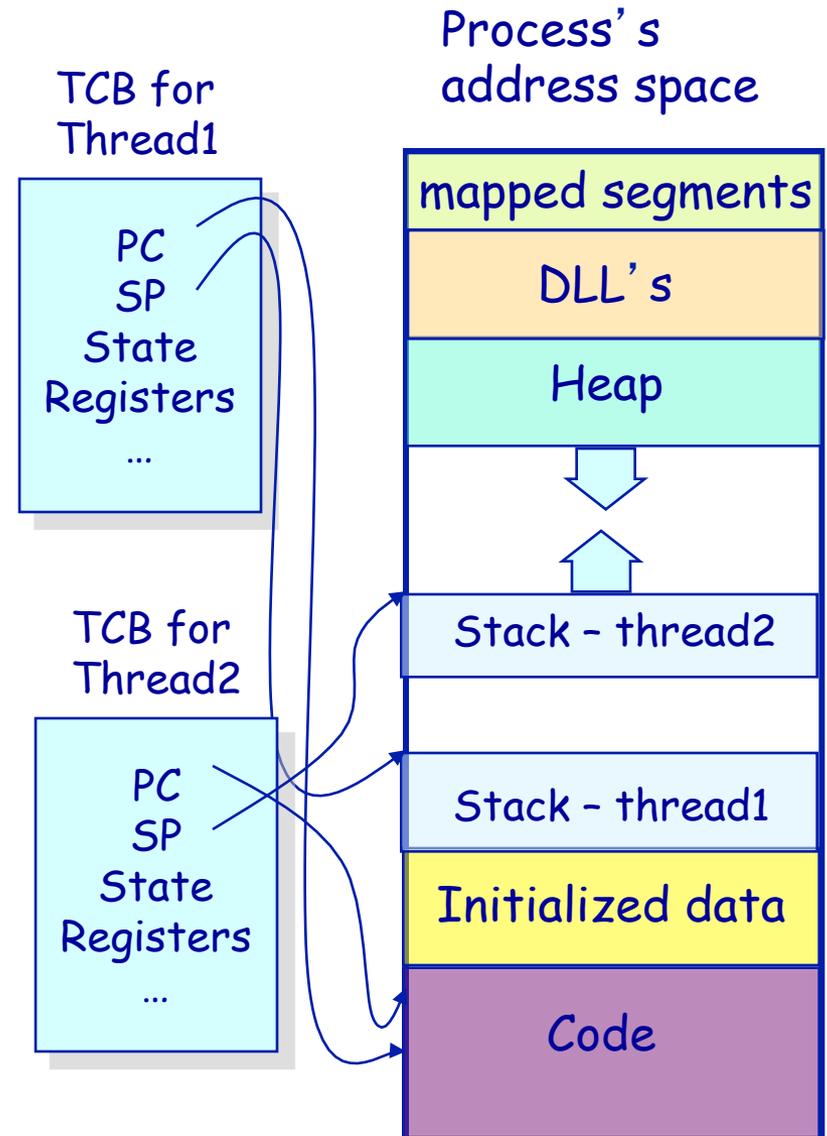
- ◆ A thread has no data segment or heap
- ◆ A thread cannot live on its own, it must live within a process
- ◆ There can be more than one thread in a process, the first thread calls main & has the process's stack
- ◆ If a thread dies, its stack is reclaimed
- ◆ Inter-thread communication via memory.
- ◆ Each thread can run on a different physical processor
- ◆ Inexpensive creation and context switch

Processes

- ◆ A process has code/data/heap & other segments
- ◆ There must be at least one thread in a process
- ◆ Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- ◆ If a process dies, its resources are reclaimed & all threads die
- ◆ Inter-process communication via OS and data copying.
- ◆ Each process can run on a different physical processor
- ◆ Expensive creation and context switch

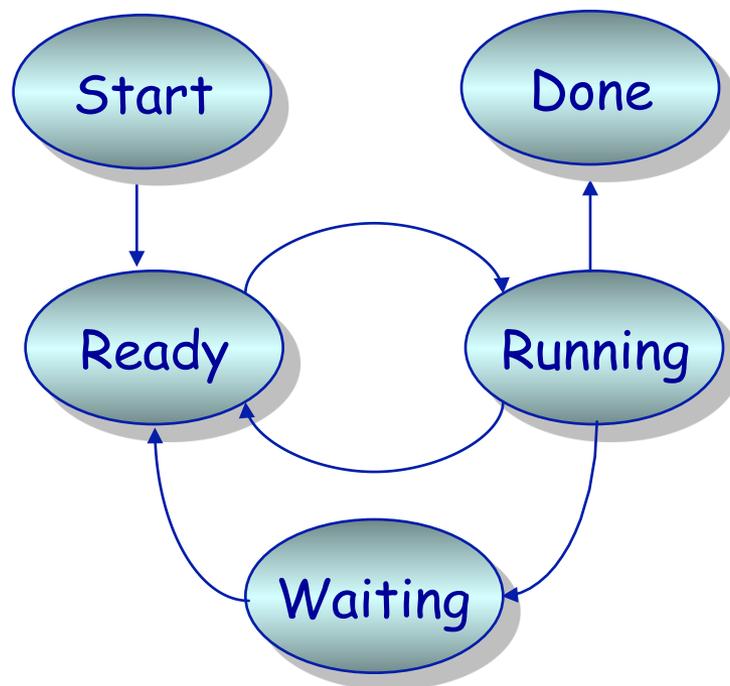
Implementing Threads

- ◆ Processes define an address space; threads share the address space
- ◆ Process Control Block (PCB) contains process-specific information
 - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- ◆ Thread Control Block (TCB) contains thread-specific information
 - Stack pointer, PC, thread state (running, ...), register values, a pointer to PCB, ...



Threads' Life Cycle

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

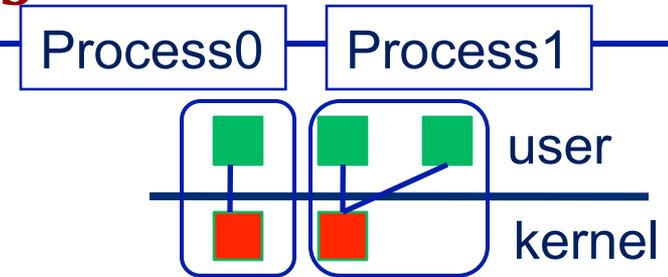


Threads have the same scheduling states as processes

1. True 😊
2. False

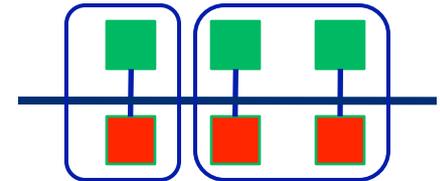
◆ In fact, OSes generally schedule *threads* to CPUs, not processes

User-level vs. Kernel-level threads



◆ User-level threads (M to 1 model)

- + Fast to create and switch
- + Natural fit for language-level threads
- - Duplicate effort (2 thread schedulers)
 - ❖ The schedulers can fight with each other
- - All user-level threads in process block on OS calls
 - ❖ E.g., read from file can block all threads



◆ Kernel-level threads (1 to 1 model)

- + Kernel-level threads do not block process for syscall
- + Only one scheduler (and kernel has global view)
- - Can be difficult to make efficient (create & switch)

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 non-blocking
 - 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 slow, everyone starts writing user-level thread packages
 - ❖ Java did this for a while
 - 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
 - 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
 - A special order for 1 green car, still takes 1 day
 - 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

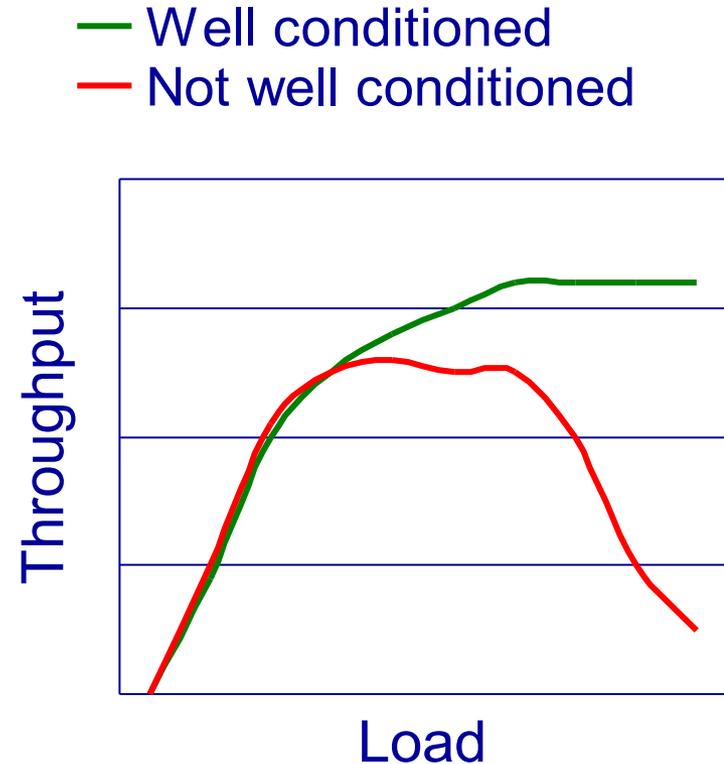
- ◆ Can threads improve throughput?
 - Yes, as long as there are parallel tasks and CPUs available
- ◆ Can threads improve latency?
 - Yes, especially when one task might block on another task's IO
- ◆ Can threads harm throughput?
 - Yes, each thread gets a time slice.
 - If # threads \gg # CPUs, the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

Best Practices?

- ◆ For CPU-intensive work, applications generally create one thread per CPU
- ◆ For work with I/O, the number of threads is tuned to keep the CPU busy but not overloaded
 - E.g., $3 * \# \text{ CPUs}$
 - Tuning effort often application-specific
- ◆ Applications like web servers often keep *thread pools*, or a set of n ready threads
 - New requests are assigned to an existing thread to avoid overloading the system
 - Plus, reduce setup/tear down costs!

Thread or Process Pool

- ◆ Creating a thread or process for each unit of work (e.g., user request) is dangerous
 - High overhead to create & delete thread/process
 - Can exhaust CPU & memory resource
- ◆ Thread/process pool controls resource use
 - Allows service to be well conditioned.



When a user level thread does I/O it blocks the entire process.

1. True 😊
2. False

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