CSE 451: Operating Systems
Spring 2011

Module 4 - Threads

John Zahorjan
zahorjan@cs.washington.edu
Allen Center 534
Module Overview

1) Big Picture: Achieving Concurrency/Parallelism

2) Kernel Threads

3) User-Level Threads

4) Task/Work Queues

5) Event-driven Programming

6) Event-driven Programming Discussion
1. The Big Picture

- Threads are about **concurrency** and **parallelism**
  - Reminder:
    - **Parallelism**: physically simultaneous operations for performance
    - **Concurrency**: logically (and possibly physically) simultaneous operations for convenience

- One way to get concurrency and parallelism is using multiple processes
  - The programs (code) of distinct processes are isolated from each other, at run time and at coding time

- Threads are another way to get concurrency and parallelism
  - Threads “share a process”
  - Threads directly interact, at coding time and at run time
Process Parallelism

• Multiprogramming was developed to maximize CPU utilization
  – While one process is doing I/O, another one (or ten) are eligible to run on the CPU

• Example 1:
  – While I'm working in a bash terminal on attu, so are 10 other people
  – While I'm running gcc on attu, someone else is running emacs, some else a.out, …
Example 2: Process Parallelism For A Single User

• Imagine that I execute:
  
  – $ egrep '[0-9]{3}-[0-9]{2}-[0-9]{4}' foo.txt

• If I were doing this a lot, and I cared about speed, and foo.txt is big enough (and the pattern slow enough), then I might instead execute:
  
  – $ cat foo.txt | egrep '[0-9]{3}-[0-9]{2}-[0-9]{4}'

• Why?

• Hey, it's parallelism!
Example 2 (cont.)

- $ cat foo.txt | egrep '^[0-9]{3}-[0-9]{2}-[0-9]{4}$'

- You can easily imagine it's a lot easier to write `cat` and a limited version of `egrep` separately than to write a performant version of `egrep`
  - Hey, it's concurrency!

- If you wanted `egrep` alone to be able to overlap reading the file with doing the pattern match, you should implement it like the process solution!
  - One “thread” reads the file, and puts successive lines in a buffer
  - Another thread takes lines out of the buffer, and pattern matches

- This will be quite a bit faster than the process solution
  - Why?
    - You have to get around the isolation barriers of the processes
    - Threads (within a single process) are not isolated from each other, so coordinating them is much cheaper
Example 2 Discussion

• The process parallelism worked because the communication required between `cat` and `egrep` was simple enough that a pipe suffices
  – Requirements:
    • Communication is one-way
    • Communication is a stream

• If the two processes needed more complicated communication, processes and pipes wouldn't be handy
  – E.g., both actors need to update a common data structure
Example 3: A Very Strained Analogy - Browsers

• Imagine you want to look at both the latest international news (http://global.nytimes.com/?iht) and the latest hockey scores (http://espn.go.com/nhl)

• You can:
  – Start two browser instances, and look at one page in each, or
  – Start one browser instance, and bring up each page in its own tab

• Tabs come up quicker... (and consume less screen real-estate, and generally seem lighter weight, and ...)
2. Kernel Threads

• Up to now, a process is:
  – An address space (code + data)
  – OS resources (open files, etc.)
  – A stack (procedure call trace + local variables)
  – A PC + general purpose register values

• Let's separate the concepts in that:
  – An address space
  – OS resources
  – A (kernel) thread

• Threads are concurrent executions sharing an address space (and some OS resources)
Kernel Threads vs. Processes

- Address spaces provide isolation
  - If you can't name it, you can't read or write it

- Hence, communicating between processes is difficult
  - Have to go through the OS to move data out of one address space and into another

- Because threads are in the same address space, communication is simple/cheap:
  - Just update a (non-local) variable!
Example Opportunities for Threads

• Imagine a web server, which might like to handle multiple requests concurrently
  – While waiting for the credit card server to approve a purchase for one client, it could be retrieving the data requested by another client from disk, and assembling the response for a third client from cached information

• Imagine a web client (browser), which might like to initiate multiple requests concurrently
  – The CSE home page has 46 “src= ...” html commands, each of which is going to involve a lot of sitting around! Wouldn’t it be nice to be able to launch these requests concurrently?

• Imagine a single parallel program running on a multiprocessor, which might like to employ “physical concurrency”
  – For example, multiplying two large matrices – split the output matrix into k regions and compute the entries in each region concurrently, using k processors
Implementing Threads

• Given the process abstraction as we know it, we could:
  – fork several processes
  – cause each to map to the same physical memory to share data
    • see the `shmget()` system call for one way to do this (kind of)

• This is like making a pig fly – it’s really inefficient
  – space: PCB, page tables, etc.
  – time: creating OS structures, fork/copy address space, etc.

• Some equally bad alternatives for some of the examples:
  – Entirely separate web servers
  – Manually programmed asynchronous programming (non-blocking I/O) in the web client (browser)
Can we do better?

• Key idea:
  – separate the concept of a process (address space, OS resources)
  – ... from that of a minimal “thread of control” (execution state: stack, stack pointer, program counter, registers)

• This execution state is usually called a thread
Threads and processes

• Most modern OS’s (Mach (Mac OS), Chorus, Windows, UNIX) therefore support two entities:
  – the process, which defines the address space and general process attributes (such as open files, etc.)
  – the thread, which defines a sequential execution stream within a process

• A thread is bound to a single process / address space
  – address spaces, however, can have multiple threads executing within them
  – sharing data between threads is cheap: all see the same address space
  – creating threads is cheap too!

• Threads become the unit of scheduling
  – processes / address spaces are just containers in which threads execute
Kernel threads

address space

thread

Mach, NT, Chorus, Linux, ...

os kernel

CPU

(thread create, destroy, signal, wait, etc.)
(old) Process address space

- **address space**
  - 0x00000000
  - 0xFFFFFFFF

- **Stack**
  - (dynamic allocated mem)

- **Heap**
  - (dynamic allocated mem)

- **Static data**
  - (data segment)

- **Code**
  - (text segment)

- **PC**
- **SP**
(new) Address space with threads

0x00000000

address space

0xFFFFFFFF

thread 1 stack

thread 2 stack

thread 3 stack

heap
(dynamic allocated mem)

static data
(data segment)

code
(text segment)

PC (T2)

SP (T1)

SP (T2)

SP (T3)

PC (T1)

PC (T3)
Terminology

• Just a note that there’s the potential for some confusion ...
  – Old world: “process” == “address space + OS resources + single thread”
  – New world: “process” typically refers to an address space + system resources + all of its threads ...
    • When we mean the “address space” we need to be explicit
      “thread” refers to a single thread of control within a process / address space

• A bit like “kernel” and “operating system” ...
  – Old world: “kernel” == “operating system” and runs in “kernel mode”
  – New world: “kernel” typically refers to the microkernel; lots of the operating system runs in user mode
The design space

Key

- address space
- thread

**MS/DOS**
- one thread per process
  - one process

**older UNIXes**
- one thread per process
  - many processes

**Java**
- many threads per process
  - one process

**Mach, NT, Chorus, Linux, …**
- many threads per process
  - many processes
Where do (kernel) threads come from?

• **Natural answer:** the kernel is responsible for creating/managing threads
  – for example, the kernel call to create a new thread would
    • allocate an execution stack within the process address space
    • create and initialize a Thread Control Block (TCB)
      – stack pointer, program counter, register values
    • stick it on the ready queue

– There is a “thread name space”
  • Thread id's (tid's)
  • tid's are integers (surprise!)
Kernel thread summary

- OS now manages threads and processes / address spaces
  - all thread operations are implemented in the kernel
    - e.g., thread creation
  - OS schedules all of the threads in a system
    - if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
    - possible to overlap I/O and computation inside a process

- Kernel threads are cheaper than processes
  - less state to allocate and initialize

- But, they’re still pretty expensive for fine-grained use
  - orders of magnitude more expensive than a procedure call
  - thread operations are all system calls
    - context switch
    - argument checks
  - must maintain kernel state for each thread
3. User-Level Threads

• There is an alternative to kernel threads

• Threads can also be created and managed at the user level (that is, entirely from within the process)
  – a library linked into the program manages the threads
    • because threads share the same address space, the thread manager doesn’t need to manipulate address spaces (which only the kernel can do)
    • threads differ (roughly) only in hardware contexts (PC, SP, registers), which can be manipulated by user-level code
    • the thread package multiplexes user-level threads on top of kernel thread(s)
    • each kernel thread is treated as a “virtual processor”

• We call these user-level threads
User-level threads example

User-level thread library
(thread create, destroy, signal, wait, etc.)

address space

thread

os kernel

CPU
User-level threads: what the kernel sees
User-level threads: the full story

kernel threads

Mach, NT, Chorus, Linux, ...

user-level thread library

(thread create, destroy, signal, wait, etc.)

CPU

(os kernel)

(kernel thread create, destroy, signal, wait, etc.)

thread

address space

04/10/11
User-level threads

• User-level threads are small and fast
  – managed entirely by user-level library
    • e.g., pthreads (*libpthreads.a*)
  – each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
  – creating a thread, switching between threads, and synchronizing threads are done via procedure calls
    • no kernel involvement is necessary!
  – user-level thread operations can be 10-100x faster than kernel threads as a result

• Still need kernel threads...
Performance example

• On a 700MHz Pentium running Linux 2.2.16 (only the relative numbers matter; ignore the ancient CPU!):

  – Processes
    • `fork/exit`: 251 µs

  – Kernel threads
    • `pthread_create()/pthread_join()`: 94 µs (**2.5x faster**)

  – User-level threads
    • `pthread_create()/pthread_join`: 4.5 µs (**another 21x faster**)
User-level thread implementation

• The OS dispatches the kernel thread

• This kernel thread executes user code, including the thread support library and its associated thread scheduler

• The thread scheduler determines when a user-level thread runs
  – it uses queues to keep track of what threads are doing: run, ready, wait
    • just like the OS and processes
    • but, implemented at user-level as a library
Thread interface

- This is taken from the POSIX pthreads API:
  - `rcode = pthread_create(&t, attributes, start_procedure)`
    - creates a new thread of control
    - new thread begins executing at start_procedure
  - `pthread_cond_wait(condition_variable, mutex)`
    - the calling thread blocks, sometimes called thread_block()
  - `pthread_signal(condition_variable)`
    - starts a blocked thread (one waiting on the condition variable)
  - `pthread_exit()`
    - terminates the calling thread
  - `pthread_wait(t)`
    - waits for the named thread to terminate
Thread context switch

• Very simple for user-level threads:
  – save context of currently running thread
    • push CPU state onto thread stack
  – restore context of the next thread
    • pop CPU state from next thread’s stack
  – return as the new thread
    • execution resumes at PC of next thread
  – Note: no changes to memory mapping required...

• This is all done by assembly language
  – it works at the level of the procedure calling convention
    • thus, it cannot be implemented using procedure calls
What if a thread tries to do I/O?

- The kernel thread “powering” it is lost for the duration of the (synchronous) I/O operation!
  - The kernel thread blocks in the OS, as always
  - It maroons with it the state of the user-level thread
    - Where was it executing? What were the register values?

- Could have one kernel thread “powering” each user-level thread
  - “common case” operations (e.g., synchronization) would be quick

- Could have a limited-size “pool” of kernel threads “powering” all the user-level threads in the address space
  - the kernel will be scheduling these threads, obliviously to what’s going on at user-level
Multiple kernel threads “powering” each address space

address space

thread

CPU

os kernel

kernel threads

user-level thread library

(thread create, destroy, signal, wait, etc.)

(kernel thread create, destroy, signal, wait, etc.)
Addressing This Problem

• Effective coordination of kernel decisions and user-level threads requires OS to user-level communication
  – OS notifies user-level that it is about to suspend or destroy a kernel thread

• User-level thread package is then responsible for multiplexing its threads on the available kernel threads

• This is called scheduler activations
4. Task/Work Queues

- **Work queues** (aka task queues) are yet another approach to concurrency/parallelism.

- A “task” is a method pointer and arguments
  - Note: I didn't mention “a stack”

- A task represents work to be done, starting at some particular procedure, called with a particular set of arguments.
Work queue picture

Program

Worker threads

Work queue

head

args

fn

Function (code)

Function (code)

Function (code)

Function (code)
Why do this?

• One way to think of it is as an application of caching to improve performance
  – We create the worker threads once
  – That's caching the work of creating their stacks, initializing TCBs, etc.

• Work queues are most appropriate when the tasks are of known, finite duration
  – Open ended tasks, like “read network packets as they come in and put them in a queue”, are probably not tasks

• Tasks support fine-grained parallelism
  – Not much work in each “unit of parallelism”
  – cf. “coarse-grained parallelism”
An ideal application: row sums

- for (i=0; i<n; i++) {
  b[i] = 0;
  for (j=0; j<m; j++) {
    b[i] += a[i][j];
  }
}

- Turn inner loop into a procedure
- Put n tasks on the queue, all pointing to that procedure, but with args i=0,1,2,...,n-1

Notes:
- work is well defined/finite
- The tasks never block
  - No worries about marooning a thread
There are a lot of unanswered questions

• How many threads execute the tasks?
  – User-level or kernel threads?

• Should the work queue guarantee any kind of ordering of task execution? Priorities?

• Can tasks synchronize? How?

• How small should a task be?
  – Chunking: Increasing granularity by combining logically distinct tasks into a single one
  • E.g., executing 10 row sums instead of one

• Is it a “task” if it might block?
5. Event-driven Programming

• Events are asynchronous software notifications
  – Asynchronous: they happen any time
  – Software: they are raised by some running code
  – Notification: they are not an explicit control flow change
    • Not a procedure call

• Note the similarities to work queues

• They're an extremely common programming paradigm
  – Especially for GUI programming
  – Also for other domains

• Basic control flow:
  – Software registers a handler (function) for a particular event type
  – When the event occurs, the handler is (eventually) invoked
Example 1: System Generated Events

* In windows, each window has an event queue (called a message queue)

* windows events (aka messages) have semantics close to the hardware level

  * Examples:
    * key down(key)
    * key up(key)
    * mouse move(deltax, deltay)
    * left mouse button down(x,y)
    * double-click(x,y)

  (x,y) are window coordinates (e.g., (100,300))

* A "raw windows program" has a thread-per-window sitting in "the message loop"

  * It tries to read a message from the message queue
  * when it gets one, it looks to see if there is a "handler". If so, it invokes the handler (function)
Semantically Richer Events

• Note that the windows events aren't exactly convenient
  – “left button down(100,300)”
  • The user clicked on something, but what?

• The software application must convert this to something meaningful
  – Figure out that (100,300) is in “File” in the menu bar

• A distinct event system layer can be built at this layer, on top of raw events
Example 2: Javascript

```html
<html>
<body>

Field1: <input type="text" id="field1" value="Hello World!" />
<br />
Field2: <input type="text" id="field2" />
<br />
Click the button to copy the content of Field1 to Field2.
<br />
<button onclick="document.getElementById('field2').value=document.getElementById('field1').value">Copy Text</button>

</body>
</html>
```
6. Event-driven Programming

Discussion

• The event mechanism **decouples the caller from the callee**
  – The caller never heard of the callee code
    • It just knows the name of an event
  – There may be 0, 1, or many handlers registered for an event
    • The caller doesn't know or care

• Programmers need to know the event namespace at coding time
  – Not the names of pieces of code (e.g., methods)
Binding Time

• **Binding** is the process of translating one name to another name
  – E.g., a function name to a memory address

• There are choices...
  – Early binding:
    • At code time (you embed a procedure name in your code)
    • At link time (static libraries)
  – Late binding
    • At run time (dynamic link libraries)

• Events introduce a level of indirection to binding
  – A famous aphorism of David Wheeler goes: “All problems in computer science can be solved by another level of indirection” *(Wikipedia: indirection)*
More Uses

• Plug-in architectures
  – Eclipse, Firefox, ...
  – These are frameworks, designed to support multiple as-yet-unwritten applications “running inside them”
  – Plug-in can get event notifications (e.g., button click) and do whatever it does

• “Publish-subscribe” systems
  – Events with guards
    • Publish a data record
    • Subscribe to data records meeting the guard (criteria)
    • Get a notification when a record is published that meets your guard
    • Especially handy in distributed systems
Summary

- Sometimes (often) multiple threads of control, sharing an address space, is the easiest way to program functionality
  - Threads are handy!

- Kernel threads are much more efficient than processes, but they’re still not cheap
  - All operations require a kernel call and argument validation

- User-level threads are:
  - Fast/cheap
  - Great for common-case operations
    - Creation, synchronization, destruction
  - Can suffer in uncommon cases due to kernel obliviousness
    - I/O
    - Preemption of a lock-holder

- Work queues are even faster/cheaper
  - Most appropriate for limited executions

- Event-driven programming is an important specialization / extension