CS 537
Lecture 16
Threads

Michael Swift

Questions answered in this lecture:

- Why are threads useful?
- How does one use POSIX pthreads?

What's in a process?

- A process consists of (at least):
  - User ID
  - state flags
  - an address space
  - the code for the running program
  - the data for the running program
  - an execution stack and stack pointer (SP)
    - traces state of procedure calls made
  - the program counter (PC), indicating the next instruction
  - a set of general-purpose processor registers and their values
  - a set of OS resources
    - open files, network connections, sound channels, …
- That's a lot of concepts bundled together!

Organizing a Process

- Scheduling / execution
  - state flags
  - an execution stack and stack pointer (SP)
  - the program counter (PC), indicating the next instruction
  - a set of general-purpose processor registers and their values
- Resource ownership / naming
  - user ID
  - an address space
  - the code for the running program
  - the data for the running program
  - a set of OS resources
    - open files, network connections, sound channels, …

Concurrency

- 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 CS home page has 66 “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 parallel program running on a multiprocessor, which might like to concurrently employ multiple processors
  - For example, multiplying a large matrix – split the output matrix into k regions and compute the entries in each region concurrently using k processors
- Image a program with two independent tasks: saving (or printing) data and editing text

Common Programming Models

- Concurrent programs tend to be structured in one of three common models:
  - Manager/worker
    Single manager handles input and assigns work to the worker threads
  - Producer/consumer
    Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
  - Pipeline
    Task is divided into series of subtasks, each of which is handled in series by a different thread

What’s needed?

- In each of these examples of concurrency (web server, web client, parallel program):
  - Everybody wants to run the same code
  - Everybody wants to access the same data
  - Everybody has the same privileges
  - Everybody uses the same resources (open files, network connections, etc.)
- But you’d like to have multiple hardware execution states:
  - an execution stack and stack pointer (SP)
    - traces state of procedure calls made
  - the program counter (PC), indicating the next instruction
  - a set of general-purpose processor registers and their values

How could we achieve this?

- Given the process abstraction as we know it:
  - fork several processes
  - cause each to map to the *same* address space 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 and copy addr space, etc.
- Some equally bad alternatives for some of the cases:
  - Entirely separate web servers
  - Asynchronous programming in the web client (browser)

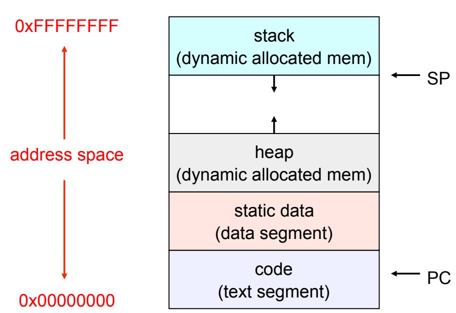
Can we do better?

- Key idea:
  - separate the concept of a process (address space, etc.)
  - from that of a minimal "thread of control" (execution state: PC, etc.)
- This execution state is usually called a thread, or sometimes, a lightweight process

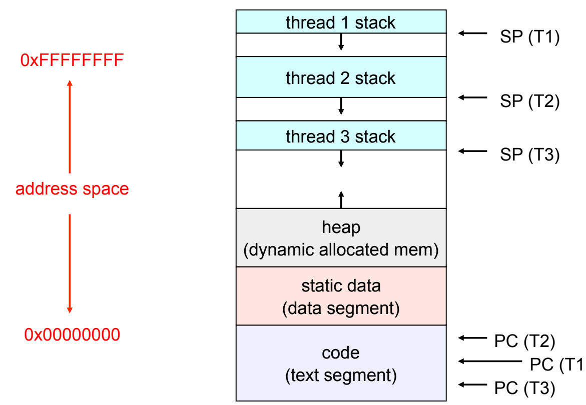
Threads and processes

- Most modern OS's (Mach, Chorus, Windows XP, modern Unix (not Linux)) 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
  - processes, however, can have multiple threads executing within them
  - sharing data between threads is cheap: all see same address space
- Threads become the unit of scheduling
  - processes are just containers in which threads execute

(old) Process address space



(new) Address space with threads

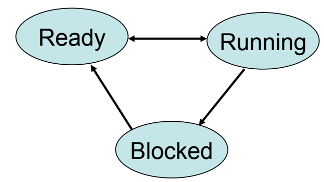


Process/thread separation

- Concurrency (multithreading) is useful for:
  - handling concurrent events (e.g., web servers and clients)
  - building parallel programs (e.g., matrix multiply, ray tracing)
  - improving program structure (the Java argument)
- Multithreading is useful even on a uniprocessor
  - even though only one thread can run at a time
  - QUESTION: When?
- Supporting multithreading – that is, separating the concept of a process (address space, files, etc.) from that of a minimal thread of control (execution state), is a big win
  - creating concurrency does not require creating new processes
  - "faster better cheaper"

Thread states

- Threads have states like processes



- Example: a web server

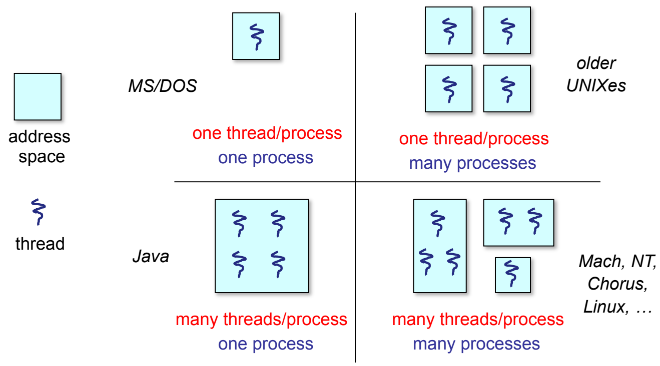
"Where do threads come from, Mommy?"

- 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
      - stack pointer, program counter, register values
    - stick it on the ready queue
  - we call these kernel threads

Kernel threads

- OS now manages threads *and* processes
  - all thread operations are implemented in the kernel
  - 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 (e.g., 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

The design space



- Threads can also be 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
    - Thread package multiplexes user-level threads on top of kernel thread(s), which it treats as "virtual processors"
  - we call these user-level threads

User-level threads

- To make threads cheap and fast, they need to be implemented at the user level
  - managed entirely by user-level library, e.g. `libpthreads.a`
- User-level threads are small and fast
  - 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

Thread context switch

- Like process context switch
  - trap to kernel
  - save context of currently running thread
    - push machine state onto thread stack
  - restore context of the next thread
    - pop machine state from next thread's stack
  - return as the new thread
    - execution resumes at PC of next thread
- What's not done:
  - change address space

Performance example

- On a 3GHz Pentium running Linux 2.6.9:
  - Processes
    - `fork/exit/waitpid`: 120 μs
  - Kernel threads
    - `clone/waitpid`: 13 μs
  - User-level threads
    - `pthread_create()/pthread_join`: < 1 μs

User-level thread implementation

- The kernel thread (the kernel-controlled executable entity associated with the address space) executes the code in the address space
- This code includes the thread support library and its associated thread scheduler
- The thread scheduler determines when a 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

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 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

Thread interface

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

How to keep a thread from hogging the CPU?

- Strategy 1: force everyone to cooperate
  - a thread willingly gives up the CPU by calling `yield()`
  - `yield()` calls into the scheduler, which context switches to another ready thread
  - what happens if a thread never calls `yield()`?
- Strategy 2: use preemption
  - scheduler requests that a timer interrupt be delivered by the OS periodically
    - usually delivered as a UNIX signal (man signal)
    - signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
  - at each timer interrupt, scheduler gains control and context switches as appropriate

Cooperative Threads

A ***cooperative*** 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 pre-emptive scheduling
- Advantages:
  - Simple
    - Scientific apps
- Disadvantages:
  - For badly written code
- Scheduler gets invoked only when Yield is called
- A thread could yield the processor when it blocks for I/O

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!
- 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 its threads obliviously to what's going on at user-level

What if the kernel preempts a thread holding a lock?

- Other threads will be unable to enter the critical section and will block (stall)
  - tradeoff, as with everything else
- Solving this requires coordination between the kernel and the user-level thread manager
  - "scheduler activations"
    - a research paper from UW with huge effect on industry
    - each process can request one or more kernel threads
      - process is given responsibility for mapping user-level threads onto kernel threads
      - kernel promises to notify user-level before it suspends or destroys a kernel thread
    - *ACM TOCS 10*,1

Summary

- You really want multiple threads per address space
- Kernel threads are much more efficient than processes, but they're still not cheap
  - all operations require a kernel call and parameter verification
- User-level threads are:
  - fast as blazes
  - great for common-case operations
    - creation, synchronization, destruction
  - can suffer in uncommon cases due to kernel obliviousness
    - I/O
    - preemption of a lock-holder
- Scheduler activations are the answer
  - pretty subtle though

Multithreading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
  - Asynchronous vs. Deferred Cancellation
- Signal handling
  - Which thread to deliver it to?
- Thread pools
  - Creating new threads, unlimited number of threads
- Thread specific data
- Scheduler activations
  - Maintaining the correct number of scheduler threads