CS 537
Lecture 19
Threads and Cooperation

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

Why support Threads?

- Divide large task across several cooperative threads
- Multi-threaded task has many performance benefits
  - Adapt to slow devices
    One thread waits for device while other threads computes
  - Defer work
    One thread performs non-critical work in the background, when idle
  - Parallelism
    Each thread runs simultaneously on a multiprocessor
  - Modularity
    Independent tasks can be untangled

Common Programming Models

- Multi-threaded 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)

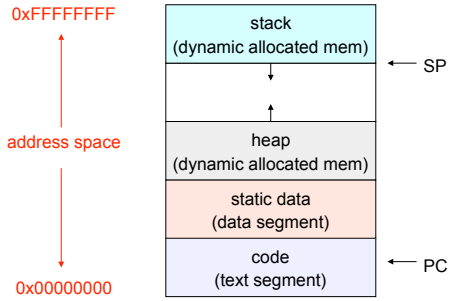
Can we do better?

- 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

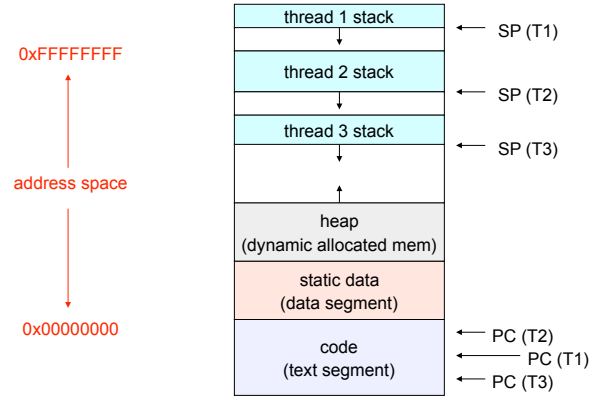
Threads and processes

- 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

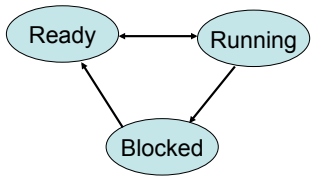


Process/thread separation

- 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
- 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

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

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

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