What's in a process?

- A process consists of (at least):
  - an address space
  - the code for the running program
  - the static and dynamic 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!

- Today: decompose ...
  - address space
  - thread of control (stack, stack pointer, program counter, registers)
  - OS resources

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

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 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 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

How could we achieve this?

- Given the process abstraction as we know it:
  - fork several processes
  - cause each to map to the same physical memory to share data (see the shmem() 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, or sometimes, a lightweight process
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

The design space

Key

- address space
- thread

MS/OS

- 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

(old) Process address space

0xFFFFFFFF

stack (dynamic allocated mem)

address space

0x00000000

heap (dynamic allocated mem)

static data (data segment)

code (text segment)

SP

PC

(new) Address space with threads

0xFFFFFFFF

thread 1 stack

address space

0x00000000

thread 2 stack

heap (dynamic allocated mem)

static data (data segment)

code (text segment)

PC (T1)

PC (T2)

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

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”
“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 / address spaces
  – 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
    – 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

“Where do threads come from, Mommy?” (2)

• There is an alternative to kernel threads
• 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
    • 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: the full story

- User-level threads are small and fast
  - managed entirely by user-level library, e.g., _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

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 20x faster)

Performance example (2)

- On a 700MHz Pentium running Linux 2.2.16:
  - On a DEC SRC Firefly running Ultrix, 1989
    - Processes
      - fork/exit: 251 μs / 11,300 μs
    - Kernel threads
      - pthread_create()/pthread_join(): 94 μs / 948 μs (12x faster)
    - User-level threads
      - pthread_create()/pthread_join: 4.5 μs / 34 μs (another 28x faster)

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

Thread interface

- This is taken from the POSIX pthread 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 user-level 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

Thread context switch

• Very simple for user-level threads:
  – 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

How to keep a user-level thread from hogging the CPU?

Thread context switch

• Very simple for user-level threads:
  – 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

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!
• 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

What if the kernel preempt a thread holding a lock?

• Other threads will be unable to enter the critical section and will block (stall)
  – tradeoff, as with everything else
  – process is given responsibility for mapping user-level threads onto kernel threads
  – kernel promise to notify user-level before it suspends or destroys a kernel thread

User-level threads are:

• creation, synchronization, destruction
• I/O
• preemption of a lock-holder

Scheduler activations are an answer

• pretty subtle though

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 validation
• User-level threads are:
  – really 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
• Scheduler activations are an answer
  • pretty subtle though
The design space

- **Address space**
  - MS/DOS
  - one thread/process
  - one process

- **Thread**
  - Java
  - many threads/process
  - one process
  - many processes

- **Operating systems**
  - Mach, NT, Chorus, Linux, ...
  - older UNIXes
  - many threads/process
  - many processes