What’s “in” a traditional process?

• A process consists of (at least):  
  – An address space, containing  
    • the code (instructions) for the running program  
    • the data for the running program  
  – Execution state, consisting of  
    • The program counter (PC), indicating the next instruction  
    • The stack pointer register (implying the stack it points to)  
    • Other general purpose register 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
Concurrency/Parallelism

• 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 dozens of “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
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
Approach

• 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
• Threads are concurrent executions sharing an address space (and some OS resources)
• Address spaces provide isolation
  – If you can’t name it, you can’t read or write it
• Hence, communicating between processes is expensive
  – Must go through the OS to move data from one address space to another
• Because threads are in the same address space, communication is simple/cheap
  – Just update a shared variable!
The design space

**Key**
- address space
- thread

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

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

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

**Mach, NT, Chorus, Linux, ...**
- many threads per process
  - many processes
(old) Process address space

- **Code (text segment):** 0x00000000 to 0xFFFFFFFF
- **Static data (data segment):** Address space
- **Heap (dynamic allocated mem):** Address space
- **Stack (dynamic allocated mem):** Address space

PC: Pointer to Code
SP: Pointer to Stack
(new) Address space with threads

0xFFFFFFFF

address space

0x00000000

thread 1 stack

thread 2 stack

thread 3 stack

heap (dynamic allocated mem)

static data (data segment)

code (text segment)

SP (T1)

SP (T2)

SP (T3)

PC (T2)

PC (T1)

PC (T3)

© 2010 Gribble, Lazowska, Levy, Zahorjan
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”
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
“Where do threads come from, Mommy?”

• Natural answer: the OS 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
  – There is a “thread name space”
    • Thread id’s (TID’s)
    • TID’s are integers (surprise!)
Kernel threads

Mach, NT, Chorus, Linux, …

Address space

Thread

CPU

OS kernel

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

-user-level thread library

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

OS kernel

CPU

(address space)

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

- Address space
- Thread
- User-level thread library
  - (thread create, destroy, signal, wait, etc.)
- Mach, NT, Chorus, Linux, …
- Kernel threads
  - (kernel thread create, destroy, signal, wait, etc.)
- OS kernel
- CPU
User-level threads

• User-level threads are small and fast
  – managed entirely by user-level library
    • E.g., pthreads (libpthread.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)
User-level thread implementation

• The OS schedules the kernel thread
• The 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 thread 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
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
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

• 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

user-level thread library

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

kernel threads

CPU

(os kernel)

(thread create, destroy, signal, wait, etc.)
What if the kernel preempts a thread holding a lock?

• Other threads will be unable to enter the critical section and will block (stall)
Addressing these problems

• 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 a kernel thread
• This is called “scheduler activations”
  • a research paper from UW with huge effect on practice
  • 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 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

© 2010 Gribble, Lazowska, Levy, Zahorjan
The design space

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

- **Java**
  - many threads/process
    - one process
  - older UNIXes

- **Mach, NT, Chorus, Linux, …**
  - many threads/process
    - many processes

© 2007 Gribble, Lazowska, Levy, Zahorjan