What’s in a process?

- According to the book… a process consists of (at least):
  - an address space
  - the code for the running program
  - the data for the running program
  - at least one thread
    - Registers, IP
    - Floating point state
    - Stack and stack pointer
  - a set of OS resources
    - open files, network connections, sound channels, …
- That’s a lot of concepts bundled together!
- Today: decompose …
  - threads of control
  - (other resources…)
States of a thread

- **running**
- **ready**
- **blocked**
- **dispatch**
- **interrupt**
- **exception**
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 a large matrix – 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
How could we achieve this?

• Given the process abstraction as we know it:
  – create several processes
  – cause each to map to the same physical memory to share data
    • see the `MapViewOfFile()` system call for one way to do this (kind of);
      use `mmap()` on LINUX/UNIX

• This is like making a pig fly – it’s really inefficient
  – space: `_KPROCESS, page tables, etc.
  – time: creating OS structures, initializing addr 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, 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 (VMS, Mach, Chorus, Windows, modern 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/DOS

- one thread/process
- one process

- many threads/process
- many processes

older UNIXes

Java

- one thread/process
- one process

- many threads/process
- many processes

Mach, WINDOWS, UNIX, …
(old) Process address space

- 0x00000000
- 0x7FFFFFFFF

- stack
  - (dynamic allocated mem)

- heap
  - (dynamic allocated mem)

- static data
  - (data segment)

- code
  - (text segment)

PC  SP
(new) Process address space with threads

- Address space: 0x00000000 - 0x7FFFFFFF
- Code (text segment)
- Static data (data segment)
- Heap (dynamic allocated mem)
- Thread stacks:
  - Thread 1 stack
  - Thread 2 stack
  - Thread 3 stack

Memory pointers:
- PC (T1)
- SP (T1)
- SP (T2)
- SP (T3)
- PC (T2)
- PC (T3)
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?”

- 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
    - See CreateThread()
  - we call these **kernel threads**
“Where do threads come from?” (2)

- 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 LINUX thread package multiplexes user-level threads on top of kernel thread(s), which it treats as “virtual processors”
  - we call these user-level 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
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 (user-space _KTHREAD)
  – 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
The design space

- MS/DOS
  - Address space
  - One thread/process
    - One process
  - Many threads/process
    - One process

- Older UNIXes
  - Many threads/process
    - Many processes
  - One thread/process
    - Many processes

- Java
  - Many threads/process
    - One process
  - Many threads/process
    - Many processes

- Mach, NT, Chorus, Linux, …
Kernel threads

address space

thread

VMS, Mach, NT, Chorus, LINUX, …

OS kernel

CPU

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

- VMS, Mach, NT, Chorus, LINUX, ...
- User-level thread library
- (thread create, destroy, signal, wait, etc.)
- Address space
- Thread
- Os kernel
- Cpu
User-level threads, really

- Address space
- Thread

VMS, Mach, NT, Chorus, LINUX, ...

User-level thread library

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

(os kernel)

CPU
Multiple kernel threads "powering" each address space

address space

thread

VMS, Mach, NT, Chorus, LINUX, ...

user-level thread library

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

• The kernel believes the user-level process is just a normal process running code
  – But, 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 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 user-level thread from hogging the CPU?

• Strategy 1: force everyone to cooperate
  – a thread willingly gives up the CPU by calling \texttt{yield()}
  – \texttt{yield()} calls into the scheduler, which context switches to another ready thread
  – what happens if a thread never calls \texttt{yield()}?

• Strategy 2: use preemption
  – scheduler requests that a timer interrupt be delivered by the OS periodically
    • usually delivered as a UNIX signal (\texttt{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

• 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
    • e.g., a thread might be preempted (and then resumed) in the middle of a procedure call
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
  – no real difference from kernel threads – “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 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 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 verification
• User-level threads are:
  – fast
  – 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