Today's agenda

- Administrivia
  - ...
- Final thoughts on processes
- Threads
  - lightweight execution contexts
  - kernel vs. user-level threads
Processes

• A process includes many things:
  – an address space (all the code and data pages)
    • protection boundary
  – OS resources (e.g., open files) and accounting info
  – hardware execution state (PC, SP, regs)
• Creating a new process is costly, because of all of
  the data structures that must be allocated/initialized
  – Linux: over 95 fields in task_struct
    • on a 700 MHz pentium, fork+exit = 251 microseconds,
      fork+exec = 1024 microseconds
• Interprocess communication is costly, since it must
  usually go through the OS
  – overhead of system calls
    • 0.46 microseconds on 700 MHz pentium

Parallel Programs

• Recall the web server example, which forks off
  copies of itself to handle multiple simultaneous tasks
  – or, imagine we have any parallel program on a
    multiprocessor
• To execute these, we need to:
  – create several processes that execute in parallel
  – 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)
  – have the OS schedule them in parallel
    • multiprogramming or true parallel processing on an SMP
• This is really inefficient
  – space: PCB, page tables, etc.
  – time: creating OS structures, fork and copy addr space, etc.
Can we do better?

• What’s similar in these processes?
  – they all share the same code and data (address space)
  – they all share the same privileges
  – they all share the same resources (files, sockets, etc.)

• What’s different?
  – each has its own hardware execution state
    • PC, registers, stack pointer, and stack

• 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, NT, 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
  – 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

0xFFFFFFFF

address space

0x00000000

stack
(dynarmic allocated mem)

heap
(dynarmic allocated mem)

static data
(data segment)

code
(text segment)

PC

SP

(new) Address space with threads

0xFFFFFFFF

address space

0x00000000

thread 1 stack

thread 2 stack

thread 3 stack

heap
(dynarmic allocated mem)

static data
(data segment)

code
(text segment)

PC

SP (T1)

SP (T2)

SP (T3)

PC (T1)

PC (T2)

PC (T3)
Thread Design Space

<table>
<thead>
<tr>
<th>MS/DOS</th>
<th>older UNIXes</th>
</tr>
</thead>
<tbody>
<tr>
<td>one thread/process</td>
<td>many processes</td>
</tr>
<tr>
<td>one process</td>
<td>many processes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java</th>
<th>Mach, NT, Chorus, Linux, …</th>
</tr>
</thead>
<tbody>
<tr>
<td>many threads/process</td>
<td>many threads/process</td>
</tr>
<tr>
<td>one process</td>
<td>many processes</td>
</tr>
</tbody>
</table>

Process/Thread Separation

- Separating threads and processes makes it easier to support multi-threaded applications
  - creating concurrency does not require creating new processes
- Concurrency (multithreading) is useful for:
  - improving program structure (the Java argument)
  - handling concurrent events (e.g., web servers)
  - building parallel programs (e.g., raytracer)
- So, multithreading is useful even on a uniprocessor
  - even though only one thread can run at a time
Kernel thread and user-level threads

- Who is responsible for creating/managing threads?
- Two answers, in general:
  - the OS (kernel threads)
    - thread creation and management requires system calls
  - the user-level process (user-level threads)
    - a library linked into the program manages the threads
- Why is user-level thread management possible?
  - threads share the same address space
    - therefore the thread manager doesn’t need to manipulate address spaces
  - threads only differ in hardware contexts (roughly)
    - PC, SP, registers
    - these can be manipulated by the user-level process itself!

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 can still be too expensive
  - thread operations are all system calls
    - OS must perform all of the usual argument checks
    - but want them to be as fast as a procedure call!
  - must maintain kernel state for each thread
    - can place limit on # of simultaneous threads, typically ~1000
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 (TBC)
  – 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:
  – Processes
    • fork/exit: 251 µs
  – Kernel threads
    • pthread_create()/pthread_join(): 94 µs
  – User-level threads
    • pthread_create()/pthread_join: 4.5 µs
User-level Thread Limitations

• But, user-level threads aren’t perfect
  – tradeoff, as with everything else
• User-level threads are invisible to the OS
  – there is no integration with the OS
• As a result, the OS can make poor decisions
  – scheduling a process with only idle threads
  – blocking a process whose thread initiated I/O, even though the process has other threads that are ready to run
  – unscheduling a process with a thread holding a lock
• Solving this requires coordination between the kernel and the user-level thread manager

Coordinating K/L and U/L Threads

• Another possibility:
  – use both K/L and U/L threads in a single system
  – can associate a user-level thread with a kernel-level thread
  – or, can multiplex user-level threads on top of kernel threads
• “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
• pop question:
  – why would a process have more user-level threads than kernel threads?
Thread Interface

- This is taken from the POSIX pthreads API:
  - $t = \text{pthread\_create}(\text{attributes, start\_procedure})$
    - creates a new thread of control
    - new thread begins executing at start\_procedure
  - $\text{pthread\_cond\_wait}(\text{condition\_variable})$
    - the calling thread blocks, sometimes called thread\_block()
  - $\text{pthread\_signal}(\text{condition\_variable})$
    - starts the thread waiting on the condition variable
  - $\text{pthread\_exit}()$
    - terminates the calling thread
  - $\text{pthread\_wait}(t)$
    - waits for the named thread to terminate

User-level thread implementation

- a thread scheduler determines when a thread runs
  - it uses queues to keep track of what threads are doing
    - just like the OS and processes
    - but, implemented at user-level as a library
  - run queue: threads currently running
  - ready queue: threads ready to run
  - wait queue: threads blocked for some reason
    - maybe blocked on I/O, maybe blocked on a lock
- how can you prevent a thread from hogging the CPU?
  - how did the OS handle this?
Preemptive vs. non-preemptive

• Strategy 1: force everybody 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 to caller 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