Threads

• Support concurrency/parallelism within an application e.g. a web server that

• 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)

• Threads are more lightweight, so much faster to create and switch between than processes
The design space

Key

- address space
- thread

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

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

Two approaches to implementing threads:
• Kernel threads
• User-level threads

Today:
• quick review of kernel threads
• more about user-level threads
• scheduler activations:
  adding kernel support for better user-level 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, just like processes
- Kernel threads are cheaper than processes
  - less state to manage: just the processor context (PC, SP, registers)
- Switching between kernel threads
  - trap into kernel
  - kernel saves running thread’s processor context in TCB
  - kernel picks new thread to run
  - kernel loads new thread’s registers, jumps to its saved PC

- Call this 1:1 scheduling
  - 1 app thread per 1 kernel scheduled entity
Kernel threads

All thread operations (creating, destroying, waiting) go through the kernel

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

- Can implement threading entirely in user space
  - run many user-level threads in one kernel thread
  - call this **N:1 threading**
- Keep separate stack & processor context for each thread, in user space
- User-level thread lib schedules and switches threads
- Switching between threads entails:
  - library saves running thread’s processor context
  - library picks a new thread to run
  - library restores new thread’s context, jumps to saved PC
- Pretty much same as before, but kernel not involved!
User-level threads

All thread operations (creating, destroying, waiting) are handled by the thread library (not the kernel)
User-level threads: what the kernel sees

Kernel is oblivious to user-level threads!
User-level vs kernel threads

• User level threads are faster
  – Faster to switch between threads
    • Round-trip to kernel: about 500 ns
    • Switching in user space: closer to 5 ns (like a function call)
  – Faster to create and destroy threads

• Some problems with user-level threads
  – Can we take advantage of more than one processor?
  – What if one of the threads does I/O, and blocks?

• Basic problem: lack of information in each scheduler
  – Kernel doesn’t know about user-level threads
  – User-level scheduler doesn’t know about other processes
User-level scheduling, multiprocessor style

- If all user-level threads run in one kernel thread, only one can run at a time!
- Most machines have more than 1 CPU core now…

- Solution: use more than one kernel thread!
  1 kernel thread per processor (N:M threading)
- User-level scheduler in each kernel thread chooses which user-level thread to run
- Kernel schedules the kernel-level threads, but is still oblivious to what's going on at user level
Multiple kernel threads “powering” each address space

(address space)

(thread)

 millennials

[thread create, destroy, signal, wait, etc.]

user-level thread library

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

(os kernel)

(CPU)
What if a thread tries to do I/O?

- The kernel thread “powering” it is lost for the duration of the I/O operation!
- Even if other user-level threads are ready, can't run them!
- Kernel doesn’t know there’s anything else ready to run
- Same problem with other blocking ops (e.g. page faults)
Scheduler Activations

• Support for user-level threads without these problems

• Basic idea:
  – let the kernel scheduler and the user-level scheduler coordinate with each other
  – involves communication from user-level to OS and back

• From UW: [Anderson, Bershad, ‘92]
• Lots of impact on practical systems (more info later)
Scheduler Activations: 2-way communication

- OS and user-level schedulers give each other hints

- User-level scheduler tells the kernel what it needs
  - request more CPUs (might not get them!) or release them

- Kernel calls user-level scheduler to notify it of events
  - more/fewer CPUs available to process
  - thread blocked on I/O, or unblocked when I/O finished

- Kernel to user-space communication: *upcall*
  - A bit unusual: usually user-space makes syscalls to kernel!
  - But this is also how signals work, and like an interrupt
Scheduler Activations

• “Scheduler activations” replace kernel threads

• A scheduler activation is like a kernel thread
  – has a separate stack and processor context
  – can be scheduled on a CPU

• …but different:
  – If the kernel interrupts an activation, it doesn’t restart it where it left off (like a thread)
  – Instead, it restarts execution in the user-level scheduler
  – User-level scheduler can then decide which thread it wants to run on that CPU
Starting a new process

- New thread starts executing in thread lib
- User-level sched picks thread to run, starts it
- Can reschedule a different user-level thread later
Blocking I/O

- Thread blocked on I/O
- Kernel creates *new* activation – starts in the thread lib, and picks a new thread to run
- When I/O finishes, old thread doesn’t resume
  - Kernel interrupts an activation, lets the scheduler pick what to run
Performance

• Is all that really faster than kernel-level threads?
  – Not really – lots of upcalls, not especially cheap

• But what we just saw were the uncommon cases!

• When threads aren’t blocking on I/O, it’s just user-level thread management!
  – orders of magnitude faster than kernel-level threads
  – and now we have an answer for the blocking I/O problem
The state of threading today

• Scheduler activations pretty widely used:
  – Various Unixes: FreeBSD, NetBSD, Solaris, Digital UNIX (some now defunct)
  – Windows 7 User-Mode Scheduling
  – Recent research on multicore Oses

• Trend back to kernel-scheduled threads
  – Linux, FreeBSD
  – performance getting better, and less complex

• User-level threading still popular in massively-parallel applications
• 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 and other blocking operations
• Scheduler activations are an answer
What if a thread tries to do I/O?

• Remember: I/O operations are blocking
• The kernel thread “powering” it is lost for the duration of the I/O operation!
  – The kernel thread blocks in the OS, as always
  – Can’t run a different user-level thread
• Same problem w/ other blocking ops (e.g. page faults)
• Again: kernel doesn’t know there are user threads, so doesn’t know there’s something else it could run