# CS 3204
# Operating Systems

Lecture 25

Godmar Back

---

## Announcements

- Project 4 due Dec 10 11:59pm
  - Don't postpone to after Thanksgiving
  - Should get done before break: buffer cache (so all regression tests pass), and have designed and partially implemented on-disk data structures (inode + index trees)
- Project 4 Help Session Slides online
- Check exam time and let me know of possible conflicts as per University policy
- Reading Chapter 10-12



---

## Address Spaces vs Protection Domains

---

## Address Spaces & Protection Domains

- Normal case: each user process has its own address space & own protection domain
- Sharing an address space means to put the same meaning to a particular virtual address
- Sharing a protection domain means to have the same access rights to a particular piece of memory
- The two are not always identical: Single address space OS
  - all processes share one address space – ideally 64bit
  - advantage: can use pointers as names for objects
  - disadvantage: loading/linking slightly more complex



---

## Address Space & Threads

- Real-world combinations

| # of address spaces | 1 | many |
|---|---|---|
| 1 thread/space | MS-DOS<br>MacOS-9 | Traditional Unix (BSD 4.3, 4.4, SVR3); Pintos |
| many threads/space<br>multi-threading | Embedded Systems;<br>Pilot (1978) | VMS, Mach, Win/NT, Solaris, Linux |

NB: threads listed here are "kernel-level" threads (KLT) – threads of which the kernel is aware

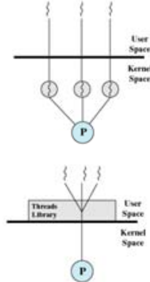


---

## Kernel-level vs User-level Threads

# Kernel-level vs User-level Threads

- Kernel-level threads: (KLT)
  - Aka 1:1 model
  - Kernel knows about them:
    - Have kernel-assigned thread id + TCB
    - Have their own kernel stack
- Alternative: it is also possible to build "user-level" threads (ULT)
  - Kernel is unaware of them
  - Aka 1:N model
- Combinations of these models are possible as well






# User-level Threads

- Usually implemented using library
  - (recall: core of context switching code in Pintos did not require any privileged instructions – so can do it in a user program also)
  - Possible to implement via setjmp/longjmp
- Similar to "co-routine" concept
- Advantages
  - can be lightweight
  - context switches can be fast (don't have to enter kernel, and since shared address space no TLB flush required)
  - can be done (almost) portably for any OS



# User-level Threads - Issues

- How can traditional RUNNING/READY/BLOCKED state model be implemented?
  - Problem: RUNNING->BLOCKED transitions should cause another READY thread to be scheduled
  - Q.: what happens if user-level thread calls the "read()" system call and blocks in kernel?
- Must use elaborate mechanisms that avoid blocking in the kernel
  - Option: Redirect all system calls that might block entire process and replace them with non-blocking versions
  - Then poll kernel later when I/O has completed (or have kernel notify you)
  - Overhead: may require additional system call for starting and completing I/O
- Since kernel sees only one thread, can use at most 1 CPU – not truly SMP-capable
  - But: where application-specific scheduling is desirable, it can be more easily implemented in a ULT system



# Preemption vs Nonpreemption

- Implementing preemption in user-level threads requires timer- or I/O-interrupt like notification facility (SIGALRM + SIGIO in Unix)
  - But then overhead of saving all state returns
- Truly lightweight user-level threads are non-preemptive
  - Highly scalable in number of threads
  - Makes implementing locks really easy – no need for atomic instructions!
  - But then: cannot preempt uncooperative threads, lose ability to round-robin schedule CPU bound threads



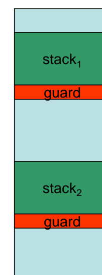
# Aside: UNIX/POSIX Signals

- General notification interface that is used for many things in POSIX-like systems
- Examples (read kill(2), signal(2), signal(7)):
  - Job control (Ctrl-C, Ctrl-Z) send SIGINT/SIGSTOP to process
  - Processes can send each other (or themselves) signals
  - Signals are used for error conditions: SIGSEGV, SIGILL
  - Also used for timers, I/O conditions, profiling
- Default handling depends on signal: ignore, terminate, stop, core-dump
  - processes can override handling
  - kernel may invoke signal handlers if so instructed: like interrupt handlers – same issues apply wrt safety
- POSIX signals are per-process, complex rules describe which thread within process may handle a signal

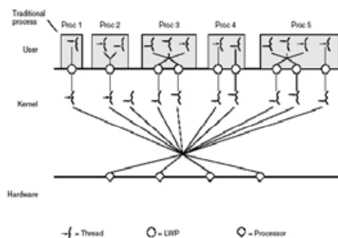


# Managing Stack Space

- Stacks require continuous part of virtual address space
  - On 32-bit systems: virtual address space fragmentation can result
  - only have 3GB total in user space for code, data, shared libs – limits the number of threads
- What size should stack have?
- This is an issue for both ULT & KLT
- How to detect stack overflow (or grow stack)?
  - Detect in software or in hardware (or ignore)
  - Stack growth usually only available in KLT implementations
  - Compiler support can create discontiguous stacks
- Related Issues: how to implement
  - Get local thread id "pthread_self()"
  - Thread-local storage (TLS)



# M:N Model




- Solaris Lightweight Processes



# M:N Model (cont'd)

- Invented for use in Solaris OS early 90s
- Championed for a while
  - Idea was to get the best of both worlds
  - Fast context switches if between user-level threads
  - Yet enough concurrency to exploit multiple CPUs
- Since abandoned in favor of kernel-level threads only approach
  - Too complex – what's the "right" number of LWP?
  - 2-level scheduling/resource management was hard: both user/kernel operated half-blind
  - Interesting history can be found here



# Multi-Threading in Linux

- Went through different revisions
  - Started as 1:1 with minimal kernel support (only clone(2)), and high overhead ("linux-threads")
- Today (Linux 2.6): NPTL – Next-Generation POSIX Thread Library
  - 1:1 model
- optimizes synchronization via "futexes"
  - avoids mode switch for common case of uncontended locks by performing atomic operation
  - constant-time scheduling operation allow for scaling in number of threads



# Summary

- Address Spaces vs Protection Domains
- Kernel vs User-Level Threads
  - Don't confuse "kernel-level threads" with "kernel threads"
    - kernel threads: never execute user code, execute kernel code, in kernel mode, only; are not associated with user address space
    - kernel-level threads: are threads that execute user code, can call into the kernel via syscalls/page faults; are associated with a user address space
    - both are viewed as schedulable entities by the kernel