CS 3204 Operating Systems

Lecture 29

Godmar Back

Announcements

- Project 3 page table design document
  - Should have received feedback – ask us questions if you're still unclear
- Project 3 due April 13
- Th Apr 6, 7pm, 655 McBryde: attend town-hall meeting regarding planned restructuring of 6th floor undergrad space

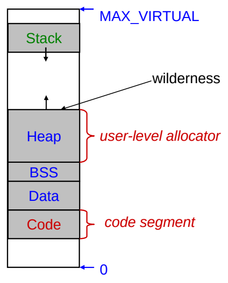
Memory Management Wrap-Up, Address Spaces & User-level Threads

Mem Mgmt Without Virtual Memory

- Book spends a great deal of time on it
  - Historically important, and still important for VM-less devices (embedded devices, etc.)
- Imagine if we didn't have VM, it would be hard or impossible to
  - Retain the ability to load a program anywhere in memory
  - Accommodate programs that grow or shrink in size
  - Use idle memory for other programs
  - Move/relocate a running program in memory
- VM *drastically* simplifies systems design

User-level Memory Management

- Goals:
  - Minimize fragmentation
  - Speed
  - Maximize locality
  - Provide for some error detection
- Typical algorithms:
  - First-fit, best-fit
  - No universally best algorithm: can always construct worst case sequence
- Conservative heap growth
  - "wilderness preservation"



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

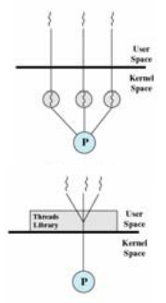
Address Space & Threads

- In Pintos: one thread per address space
- More combinations in real world:

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

Kernel-level vs User-level Threads

- Threads on previous slide were "kernel-level" threads
  - 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
  - Kernel is unaware of them
- Combinations of these models are possible as well



User-level Threads

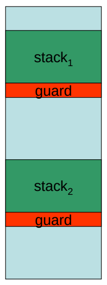
- Usually implemented using library
  - (recall: core of context switching code in Pintos did not require any privileged instructions – so can do it in a user program also)
- Similar to "co-routine" concept
- Advantages
  - can be lightweight
  - context switches can be fast (don't have to enter kernel, and since shared address space no TLB switch 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 "read()" system call and blocks in kernel?
- Must use elaborate mechanisms that avoid blocking in the kernel
  - Redirect all system calls that might block entire process and replace them with non-blocking versions
  - Overhead: may require additional system call
- Since kernel sees only one thread, can use at most 1 CPU – not truly SMP-capable

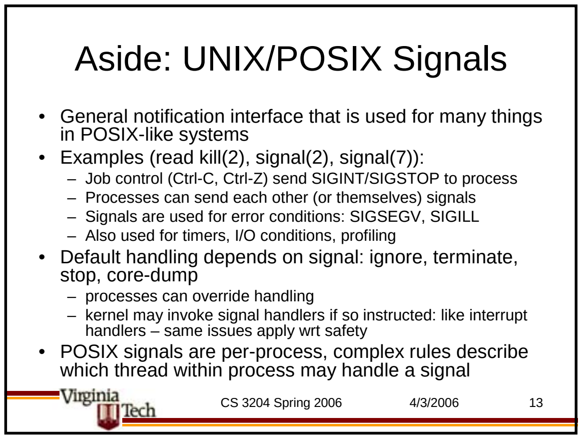
Managing Stack Space

- Stacks require continuous 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)



Preemption vs Nonpreemption

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



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