Cs5250

Tan Yee Jian

May 5, 2021

Contents

1 Week 6:

1.1 System calls

Two ways to enter kernel: `SYSCALL/SYSENTER` or an interrupt

1.1.1 How to do system calls?

- Model Specific Registers (MSR) in x86 allows configuration of OS-specific things like `sysenter/exit`
  - `rdmsr, wrmsr` instructions used to move data from(to) `MSR[ECX]` to(from) `EDX:EAX`
  - Existence can be confirmed through the `cpuid` instruction

1. SYSCALL/SYSENTER instruction and security
   - was implemented as an interrupt `int 0x80`, now using `SYSENTER/SYSEXIT/SYSCALL` instruction
   - having an instruction prevents attacker from calling it before their code (can only be called by kernel)
2. Difference between SYSENTER and interrupt Interrupts can happen even at ring 0, however SYSENTER is used to transit from other rings to 0

1.1.2 Stacks

1. Kernel stack
   - Every thread has one

    - Stored in `task_struct`
    - Grows toward the `thread_info` structure

2. Intel's hardware support for stacks: Task State Segment `https://wiki.osdev.org/Task_State_Segment`

1.1.3 Page Table Isolation

- Full set of kernel page tables used to coexist with user page tables, presents a security loophole
- Now each process's virtual memory space only save partial kernel PT

1.1.4 What happens when we do a syscall?

More info: `arch/x86/entry/entry_64.s:entry_SYSCALL_64`

1. save the current stack location and load the full kernel stack (pointed to by `CR3`)
2. Uses assembly to construct the `pt_regs` struct, which stores registers needed by `syscall`
3. Calls the service dispatcher (call the correct syscall given the sycall number), in `arch/x86/entry/common.c:do_syscall_64()`

1.1.5 How to speed up syscalls?

1. Old idea: call them in userspace
    - some system calls (eg. read, write) are called very often. How do we speed up?
    - Answer: execute **read-only** syscalls in userspace, eg. `gettimeofday()`, `time()`, `getcpu()` etc
2. New idea: virtual dynamic shared object (vDSO) + Address Space Randomization (ASR)
    - Still allows system calls (whose object file is mapped in userspace) in user space
    - However there is Address Space Randomization (ASR) to change the address of this vDSO, so attacker will not know where it is.

1.2 Interrupts

Motivating question: how to get input from devices?

1. Poll from each device regularly Waste resources
2. Use interrupts - devices take the lead

1.2.1 When are interrupts checked?

- At EVERY pipeline round, Fetch - Decode - Execute - Check for Interrupts, Fetch etc
- If there is an interrupt:
    1. Save context
    2. Get interrupt number
    3. Execute and clean up Instruction Service Routine (ISR)

1.2.2 Two types of interrupts

1. Classification 1
    (a) Async Not related to current instruction - from external sources such as mouse click.
    (b) Synchronous (aka exceptions)
        i. Faults (eg page faults) It is retried. Intel since Pentium 4 has a `replay queue` to retry instructions in case of fault.
        ii. Traps, Not retried.
            - Eg. debuggers like gdb stop at a certain breakpoint. This is done by injecting the `INT 3 (0xCC)` one-byte instruction at the instruction location where you want to stop. CPU will hit this one-byte instruction and the parent debugger process will gain control.
        iii. Abort Hardware failure
        iv. Programmed, eg syscalls
2. Classification 2
    (a) I/O interrupts
        i. Critical

      ii. Non-Critical
      iii. Non-Critical Deferrable
   (b) Timer
   (c) Interprocessor

1.2.3 Hardware to cause interrupts

1. History: Intel 8259 Programmable Interrupt Controller (PIC)
   - A hardware chip that has multiple physical pins.
   - When the pins receive signals (by ethernet, whatever), it sends interrupts to the CPU
   - Every interrupt has a mask (an interrupt is masked if CPU does not want to service this interrupt) and a priority.
2. Extension: Advanced PIC (APIC)
   - PIC can only service interrupts per-CPU. For multithreaded systems, APIC can service inter-CPU interrupts.
3. Interrupt vectors (Intel)
   - Each interrupt is mapped to a number (called vectors) in software.
   - Interrupt vector `32` and above are masked. Vector `2` is non-maskable interrupt (NMI), which is the **hardware reset button**.
   - Which pin maps to which vectore is entirely programmable (by the hardware designer)
4. Assigning IRQs to Devices
   - PCI assigns at boot
   - Usually vector = IRQ + 32
   - The mapping is saved at the Interrupt Descriptor Table. Location is indicated by the IDTR (IDT Register).

1.2.4 How are interrupts handled by x86?

1. Interrupt Descriptor Table Registor (IDTR)

   - Points to a table (IDT) consisting of either
     (a) Task-gate descriptor
     (b) Interrupt-gate descriptor No more execute
     (c) Trap-gate descriptor Can still execute

2. Interrupts and stacks

   - Interrupts are handled in the kernel, thus there is a switch of stack from higher to lower ring.
   - If a ring change happens, the stack switches to that pointed to the current task's Task State Segment (TSS), storing info for all registers.

3. Things that can happen during interrupt

   (a) Double Fault The exception-handling code induces another exception. Panics and dies.

   (b) Nested interrupt Not to be confused with above - While handling eg. mouse interrupt, a mic interrupt comes, and while handling mic interrupt, another interrupt comes etc...
   - This happens on a CPU. We can mask all other interrupts while handling one, but it is not preferrable.
   - We can utilize other cores.
   - To handle an interrupt, we check if it is nested by checking whether the privilege level is changed. If it is nested, ring 0 -> ring 0 then no change.

1.2.5 How does OS handle interrupts?

1. Overall strategy Splits the handling into **top** and **bottom** half.

   (a) Top half bare minimum needed, save register, unmask etc. Very fast to return

2. Bottom half

   (a) Software IRQ (ksoftirq)
   - Interrupt Queue as a software

  - Each processor runs a daemon, `ksoftirqd` which infinitely polls for softIRQs to service.
  - softIRQ is re-entrant, means can be re-executed as needed

(b) Tasklet
  - Can be statically/dynamically allocated, whereas softIRQ is fixed
  - Is non-re-entrant.

(c) Work queue

(d) Kernel Thread

1.3 Signals

1.3.1 POSIX Signals

- 1990 and 2001, like `SIGINT, SIGKILL, SIGTERM` etc
- Sent using syscall (to others) `kill`, or to self using `raise`

1.3.2 Data structures for signal handling

- `task_struct` is the process descriptor, saves signal handlers

1.3.3 Handling flow

- Check slides 78 of Chapter 6
- Kernel points the instruction pointer (RIP) to the signal handler, as well as back up the stack

2 Lecture 8: Memory

2.1 The memory hierachy

	L1 Cache	L2 Cache	Main Memory	Secondary Memory
clock cycle	a few	tens	hundreds	millions
size	word (4-8B)	block(8-32B)	1-4blocks	very huge

2.2 Principle of Caching: Locality

- Temporal locality: repeatedly use the same data
- Spatial locality: use nearby data

2.3 Virtual vs Physical addressing

- Every process has illusion that it owns all 2^{64} bits of memory in its virtual memory
  - What if runs out of a certain limit? Out-Of-Memory (OOM) error
- Supports “Modern OS Features”:
  - Protection: don’t use other process’s memory
  - Translation: “use disk swap”
  - Sharing: Map multiple virtual pages to a sams physical page

2.4 Address Translation

- Given page size = 4k, we have offset = 12 bits (2^{12}=4k)
- Address = virtual page # + offset
  - Virtual page # leads to an entry in page table, with
    1. A valid bit
    2. A next level virtual page #
- Page fault: page is not in main memory

2.4.1 Caching address traslation

- Problem with caching:
  - Cache that contains memory (physical page) is only relevant only when address is **translated** which is expensive
  - Enter **TLB**, which caches virtual address translation
    - Smaller, 128-256 entries max

2.5 Paging in Linux x86

- 4KB a page
- `PG` bit in register `CR0` toggles paging
- Root page table given by (40bits) “page directory base register” in `CR3`

2.5.1 x86-64 paging

- only **48** bits are used. 48 = 9 + 9 + 9 + 9 + 12
Page Level	Name
4	PLM4
3	Directory Ptr
2	Directory
1	Table
- Each process has its own CR3 hence root page table value.
- Run through:
    1. Get CR3 hence RPT address.
    2. RPT + first 9 bits = addr of Level 2 page table.
    3. Addr from 2 + next 9 bits = addr of Level 3 Page table.

1. The CR3 Register

63:M	M-12	11:0
reserved	RPT addr	PCID

2. Reserve bits must be sign-extended IE if the most significant (non-reserved) bit is 1, then the reserved is 1 and vv.
3. Process Context ID = Address Space ID
    - ASID is different across processes (even though RPT might b the same)

2.5.2 Linux memory map

- If the 48th bit is 0 -> user space
- Otherwise -> kernel space (half the Virtual Address Space)
- Note that all are sign-extended

2.5.3 Master Kernel Page Table

- Every process has own page table
- Of which contains the same MKPT
- MKPT maps the whole physical memory, unlike user process

2.5.4 TLB

- Q: When is it flushed? A: Always by kernel. For eg when CR3 is changed

2.5.5 Page Table Isolation (previously KAISER)

Only kernel process has the full kernel page table. User process has only enough kernel page table to enter kernel mode (to prevent attacks).

2.5.6 Virtual Memory Area (`vm_area_struct`)

Stores the stack, heap, data, bss segments

1. Lazy expansion
   (a) `brk()` requests for more pages when you run out of memory (eg. via repeatedly calling `malloc` until oom)
   (b) New **virtual** pages are allocated. But not linked to physical pages.
   (c) UNTIL new virtual pages are used by eg. malloc.

2.5.7 Physical Memory Zone

1. Direct Memory Access Write directly into memory of drivers, disk etc
   (a) Modes

2.6 Memory Allocator

2.6.1 Physical memory allocator (buddy allocator)

Eg. buddy system.

1. Buddy System Ask for k page frames. Upgrade to the nearest 2^{n} power. If there is such a frame, give it. Otherwise, go up one level to $2^{(n+1)}$, give it half, and put half below.

2. API
   - `kmalloc()` for physical
     - Either `GFP_ATOMIC` (not allowed to sleep, crucial)
     - or `GFP_KERNEL`
   - `vmalloc` for virtual. kernel's `malloc()`

2.6.2 Virtual memory allocator (object allocator)

1. Simple List of Blocks (SLOB)
    - A kernel object is like the project we did in 2106, contains some bookeeping data (pointing to the next free frame), some payload etc
    - SLOB uses first-fit, optimized (?) by best-fit
2. SLAB
    - A slab is a few contiguous pages.
    - Objects are aligned in to the size of cache lines.

2.7 Page Fault

2.7.1 Page Fault or SIGSEGV?

- If the address trying to access is in the process address space:

Have rights to access?	Result
Yes	Access, or Page fault
No	SIGSEGV

- Otherwise (wrong address space)

Mode?	Result
User	SIGSEGV
Kernel	Kernel bug, kill the process

2.7.2 The Linux Kernel is not pageable

This is to prevent page fault in page fault handling etc.

2.7.3 Process

1. Entry point: `arch/x86/mm/fault.c:do_page_fault()`
2. Captures the linear address causing the fault in `CR2` contro register.
3. Block the process while setting if the fault is big.
4. Run the flowchart on slide page 99

2.8 Reclamation of Page Frame (deallocation of frame)

2.8.1 Page Frame Reclamation Algorithm (PFRA)

- essentially removes/evict page frames to the swap to keep RAM free
- must run before all frames are used up (otherwise no more process can run and will crash)
- 4 kinds of frame:

Type	Reclaim action
Unreclaimable	-
Swappable (anonymous)	evict to swap
Syncable (mapped to file)	sync with hard disk
Discardable	just discard

2.8.2 Reverse mapping

Motivation: given a frame, which page table entries (PTE) point to it?

1. Object-based Reverse Mapping
   - Done by two fields:
   - `_mapcount` counts the number of PTEs pointing to it. 0-indexed (ie. -1 when no one points to it)
   - `mapping`:

state	meaning
null	belongs to the swap cache
non-null, lsb=1	anonymous, points to `anon_vma`
non-null, lsb=0	mapped, points to the `address_space` obj in page cache

   (a) `anon_vma`
      - A doubly linked list collecting all the pointers to the same frame

   (b) `address_space`
      - Similar as `anon_vma`

2.8.3 Page Reclamation Process

1. For mapped page frames:

   - Two clock lists (Least Recently Used), fixed size, one is the `active` list, the other is `inactive` list
   - Original algorithm:
     (a) If freshly faulted (allocated) you start at the inactive list
     (b) If used for the second time, promoted to active list
     (c) In active list, membership cannot be refreshed. Once sufficient active pages enter the active list, it is demoted to inactive
     (d) In inactive long enough then reclaimed, else step (2)
   - Some observations
     – eviction: only happens when a new fault is introduced. kicks the head of `inactive` list away.
     – activation: second time it is read, promoted from `inactive` to `active`.
     – min # of inactive page access = eviction + activation.
     – refault distance: R - E where E is the reading of the sum above when it is evicted, R is the reading above when it is refaulted back into memory.
       ∗ Intuition: If the list was made R-E longer, it wont be evicted.
     – Minimum access distance: length of inactive + (R-E)

2.8.4 Page Cache (for opened files)

- Write-back caching: changes are not written immediately but only marked dirty on cache. Periodically, it is written into the disk (when evicted by algorithm)

1. Important fields in `address_space`

   (a) `page_tree` a radix tree of all pages
   - Radix trees represent a map from keys (strings) to values.
   - All pages have their memory addresses as keys, pointed to the frames. Every 6 bits as a unit for branching.

2.9 Swap

- A disk partition (or a file), maximally `MAX_SWAPFILES=32`
- Swap area consists of swap slots, each slot is 4KB.
- Pages can be **swapped in** to swap or **swapped out** into RAM
- Race conditions are resolved using the swap cache

2.9.1 Swap Cache

1. Suppose A, B are processes using a same page P.
2. Page P needs to be swapped.
3. Swap cache caches P. It provides a reference in case when swapping in P, either process needs to read P.
4. Once done, both A and B point to the swap slot.

3 Lecture 9: Synchronization

3.1 Concurrency in the Kernel

- Motivation: SMP OS Kernels are pre-emptible
- True concurrency: due to multiple processors
- Pseudo concurrency: due to interleaving of instructions
  - Software based: voluntary/involuntary preemption (avoidable by disabling preemption)
  - Hardware based: interrupt/trap/fault/exception handlers (can be disabled)

3.2 Synchronization techniques in Linux

3.2.1 Per-CPU Variables

- Removes the need to synchronize between cores

1. How is this implemented?

- Code, Data, Extra, Stack Segments (CS,DS,ES,SS) have segment base = 0, creating a flat address space.
- FS, GS are exceptions.
- `SWAPGS` instruction swaps in correct value of the GS register (kept to 0 in user mode) from the MSR
- Which points to an array of pointers, maintaining per-cpu variables

3.2.2 Atomic Operations

- Atomic instructions, implemented by the CPU

3.2.3 Memory Barriers

- Optimization barrier (user library): `barrier()` to prevent reordering of instructions by the compiler
- Memory barrier (instructions): to prevent read-write reordering

3.2.4 Spin Locks (Active waiting)

1. API `spin_lock()` and `spin_unlock()`
2. Ticket Locks Assumption: there is an atomic `fetch_and_inc(var)` function.
    - entering cs: take a ticket number with fetch and inc. Active waiting when is not my number
    - exiting cs: increment the $\text{now}_{\text{serving}}$ with `fetch_and_inc(var)`.
3. Problem with interrupts
    - What if an interrupt happens when a process holds the spin lock?
    - What if the interrupt handler needs to acess data held by the spin lock holder?
4. Solution: disable interrupts while holding spin lock

3.2.5 Semaphore

1. Etymology Flags used on ships to communicate
2. Difference compared to spin locks
   - Contention for the lock causes **blocking** not **spinning**.
   - Can allow concurrency of more than 1 process if necessary
3. API
   - `down()` or `P()`: can be blocked, although can either down or try_{down}
   - `up()` or `V()`

3.2.6 Reader/writer locks

Allows multiple reader or a single writer. Increase efficiency of semaphore

1. Big reader locks (`br_lock`) Specialized form, very fast to acquire for reading but slow for writing. Think possible starvation for writer?

3.2.7 Seqlock

A reader-writer lock with high priority on writers

1. How it works
   (a) On top of a spin lock, there is a sequence field
   (b) The sequence value is incremented when writer uses
   (c) Readers might need to re-read

3.2.8 Read-copy Update (RCU)

- Lock-free data structure
- Another R-W lock. Writes are written to a new copy, and writer changes the pointer to the new copy atomically.
- Before reading, some time is given (swap to user space, idle loop) to ensure the new copy is pointed to, hence read.

3.2.9 Completion

3.3 Lock-free data structures

3.3.1 Problem with locks

1. Deadlock
2. Priority inversion
3. Convoying - long thread get the lock, stalling short ones
4. Signal safety
5. Kill-tolerance - what if the holder is killed?
6. Pre-emption tolerance - what if the holder is preempted?
7. Performance - forced sequential operations

3.3.2 The heart of lock-free data structures - CAS

... is the `compare_and_swap(*ptr, old_val, new_val)` atomic operation.

3.3.3 An example: lock-free stack

1. Structure A (singly) linked list, where head is the top of the stack
2. API
   (a) push
      i. create a linked list node with the new value
      ii. while `compare_and_swap(head, curr_node->next, curr_node)` is false, make the new node point to the (new) linked list head

   (b) pop
3. The ABA Problem
   (a) Say the stack is A -> B -> X
   (b) P1 reads A from the stack
   (c) P2 pops A
   (d) P2 pops B
   (e) P2 pushes A

(f) P1 comes back and compares with the head, which is still A, but the value is changed (they use the same pointer?)

(a) Solution
   - Keep an update count for each element. That way the new generation will not be confused as the previous.
   - Uses the doubleword compare and swap (CAS)

3.3.4 CAS is Insufficient

Imagine a linked list, P1 tries to delete the 2nd element, P2 tries to insert between the 2nd and 3rd element. The new list does not the 1st element point to P2 in one execution (verify this).

4 Lecture 10: Filesystem

A bookeeping attempt for the array of bits on hard drive.

4.1 Virtual Filesystem (VFS)

- Known though its APIs: `open, close, read, write` syscalls
- Abstracts through different fs: `ext2, ms-dos` etc

4.1.1 Common File Model

- The data structure to represent files

1. Superblock object
   - Stores info about a mounted system
   - All superblocks are stored in a doubly linked list
   - corr. to (filesystem control block)
2. Inode object
   - metadata for a file, directory etc
   - `ls -i`
   - corr. to (file control block)
3. File object

- Only exists in kernel memory, when it is opened.
- Transactonal data between a process (reading/writing it) and the file
- hierachy `task_struct->files_struct->struct file`

4. Dentry object
    - Directory entry: links dirs to files
    - Is cached since takes long to construct
    - Exists because a directory may not exist in the disk, need a common API

4.1.2 Everything is a file

filesystems include `usbfs` etc

4.1.3 API

- 0th layer: system calls
- 1st layer: `read, write` etc in libc
- 2nd layer: `fread, fwrite` etc work with file objects and have buffering

4.2 Disk filesystems

4.2.1 Purpose

- provides structure to data
- provides a logical namespace
- abstract user interface
- security

4.2.2 Journaling

- Keeps a "journal" for not-yet-committed changes to files, especially for complex filesystems like deleting a file (remove dir entry, release inode, move blocks into free pool, bookeeping. . .)
- Provides points for recovery in case the chain above breaks due to a crash

4.2.3 Examples

1. File Allocation Table (FAT)
   (a) Core Idea the FAT (the table) points to sectors, which are then linked lists of sectors forming the file.
2. New Technology File System (NTFS)

4.3 Sysfs - in-memory filesystem