Advanced Operating Systems

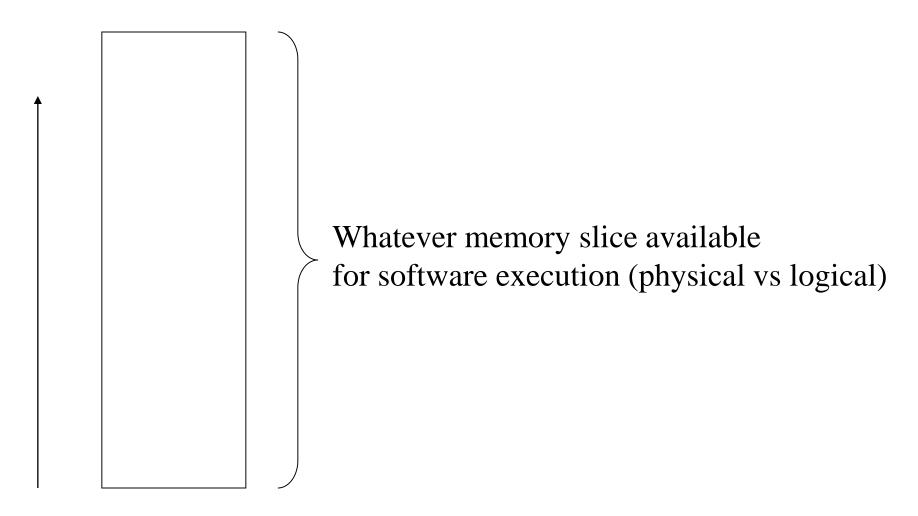
MS degree in Computer Engineering
University of Rome Tor Vergata

Lecturer: Francesco Quaglia

Kernel programming basics

- Addressing schemes and software protection models
- Hardware/software protection support
- Kernel access GATEs
- Per-CPU/per-thread memory
- System call dispatching
- Case study: the LINUX kernel

Linear addressing



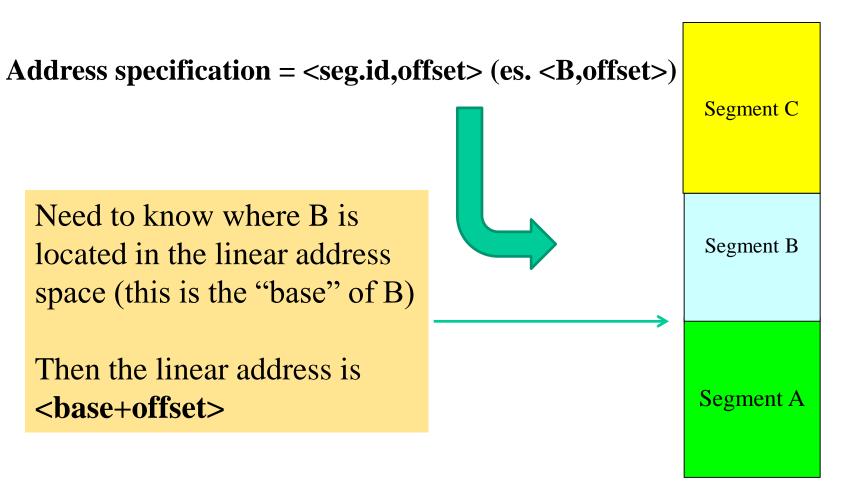
Linear address (<offset>)

Segmentation

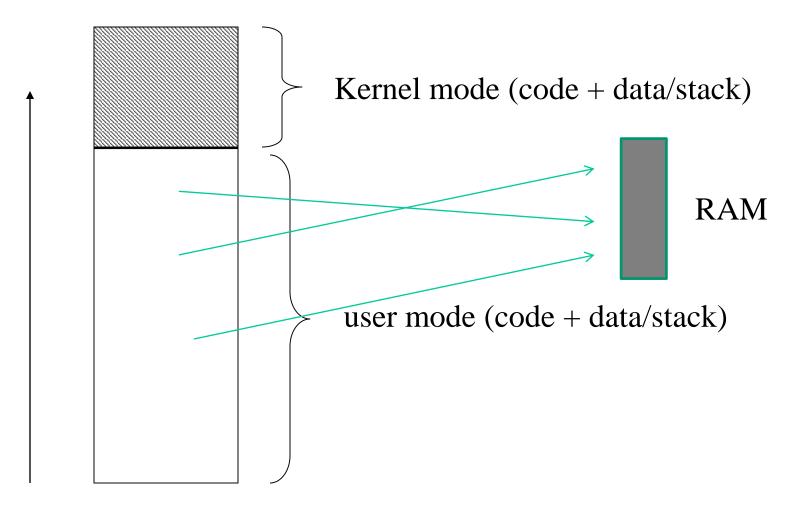
Address space (a linear one) Segment B Segment C Segment A

 $address = \langle seg.id, offset \rangle (es. \langle A, 0x10)$

Combining segments in a linear address space



Virtual memory



Linear addressing + mapping to actual storage (if existing)

Segmentation based addresses

- Code relies on addresses formed by <segment number,
 offset>
- If segment numbers are not specified by the machine instruction, some <u>default segment</u> is used for each target datum (<u>instruction or operand</u>)
- Modern processors (system processors) are equipped such in a way to support segmentation efficiently, in combination with linear addressing and virtual memory (say paging)
- The whole architecture is therefore requested to handle a complex address mapping scheme such as

segmented addr \Rightarrow linear addr \Rightarrow paged addr \Rightarrow physical addr

A very base x86 example

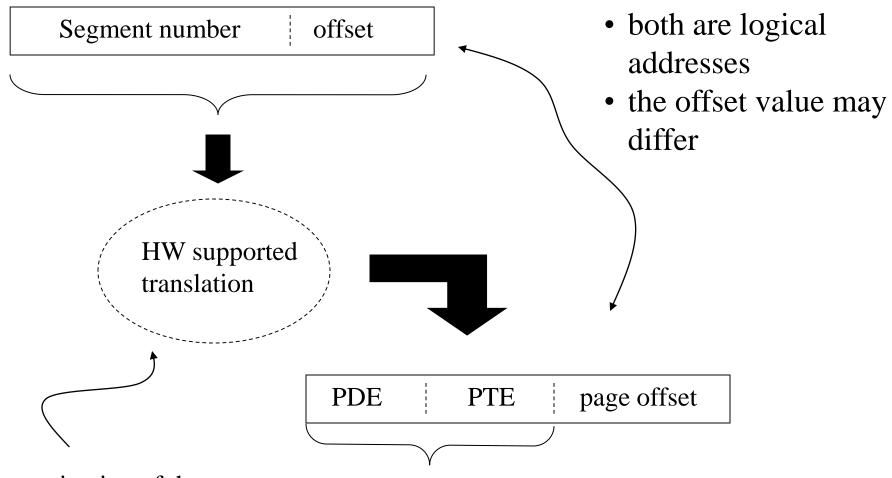
We might not imagine it but, when running this piece of code our x86 processor is implicitly using 3 different segments of memory!!

To have an exact idea of what is going on along program flow (in terms of reflection on the hardware usage) we need to know such segmentation related details

"System" processors vs segmentation

- "system" processors (those oriented to host operating system software) rely on hardware components that allow **fast and transparent access to segmentation information** (e.g. segment specific information)
- These are
 - >CPU registers
 - ➤ Main memory tables (directly pointed by registers)

Segmentation with paging



Determination of the linear address relying on <a href="https://doi.org/10.2016/j.jup/

2-level paging example

The concept of segment selector

- In general, when a memory address is expressed, the target segment is identified via a <u>segment selector register</u> (or <u>simply segment register</u>)
- Hence the access is based on segment-selector identifiers
- Through the content of the segment selector we get information on what segment ID needs to be involved in the access
- This also means that using a same selector may lead to access to different segment IDs (hence to different bases)

Segment selector register



Target segment ID

Address = <segment-selector ID, offset>

x86 memory access modes

• Real mode

- ✓ Offers backward compatibility towards 286!!
- ✓ a 16-bit segment register (there where four!) keeps the target segment ID
- ✓ 16-bit (general) registers keep the segment offset
- ✓ <u>Targeted addresses</u> are physical, and are computed as

 PhysicalAddress = Segment * 16 + Offset
- \checkmark Around 1MB (2 $^{\circ}$ 20B) of memory is allowed
- ✓ Minimal support for separating chunks of memory in the addressing scheme
- ✓ No segment specific protection information!!
- ✓ Not suited for modern software systems!!!

x86 memory access modes

- 80386 protected mode
 - ✓ a 16-bit segment register keeps the target segment ID (using 13 bits)
 - ✓ 32-bit (general) registers keep the segment offset
 - ✓ The base of the segment in linear addressing is kept into a table in memory
 - ✓ <u>Targeted addresses</u> are linear and are computed as

 address = TABLE[segment].base + offset
 - ✓ Up to 4GB of linear (either physical or logical) memory is allowed
 - ✓ 3-bit for control (protection) are kept in the segment register much better for OS software!!!

x86 memory access modes

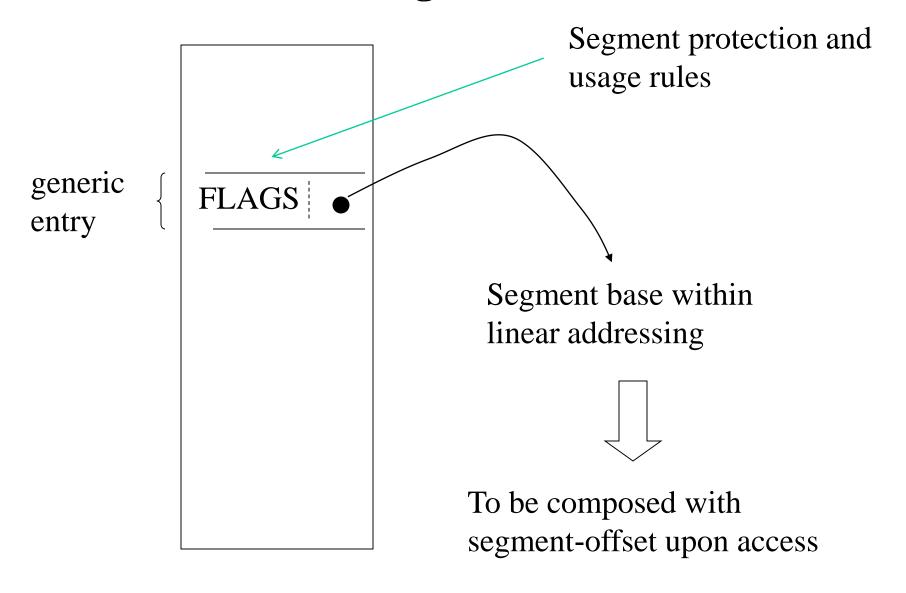
- Long mode (x86-64)
 - ✓ a 16-bit segment register keeps the target segment ID (using 13 bits)
 - ✓ 64-bit (general) registers keep the segment offset (<u>limited to 48-bit global addressing in canonical form</u>)
 - ✓ The base of the segment in linear addressing is kept into a table in memory
 - ✓ <u>Targeted addresses</u> are linear and are computed as

 address = TABLE[segment].base + offset
 - ✓ Up to 2^48 B (256 TB) of linear memory is allowed
 - ✓ 3-bit for control (protection) are kept in the segment register

x86 segment tables

- The are two table types keeping segments information:
 Global Descriptor Table (GDT) and Local Descriptor
 Table (LDT)
- Typically GDT and LDT are kept in main memory, and are directly accessible via pointers maintained by CPU registers
- GDT determines the mapping of linear addresses at least for kernel mode (namely kernel level segments) ... nowadays it is the unique used segment table in most operating systems
- LDT determines the mapping of linear addresses for user mode (namely user level segments), if not done via GDT
- These addresses are then used to access physical memory via page tables (if paging is activated)

GDT organization



Segmentation vs paging

- Segmentation and paging typically have different targets
- Segmentation is a classical means for protecting code and data
- This protection mechanism is generally based on <u>coarse grain</u> schemes (in fact, segments may have very large sizes, covering up to the whole address space of the application)
- Paging (possibly coupled with virtual memory techniques) is generally employed as a means for **improving physical-memory management efficiency**
- Such "efficiency oriented" mechanism is based on a <u>fine-grain</u> <u>approach</u>, namely it relies on the size of the page frame for the specific hardware architecture (e.g. 4KB or 2/4MB for x86 architectures)

Segmentation vs multi-cores/multi-threading

- ... we know that paging schemes are still able to enforce protection of memory (via control bits in page-table entries)
- So we may think that segmentation is somehow useless in modern software systems
- This is a wrong concept, since as we will show <u>segmentation</u> still plays a central role in multi-core architectures
- It also plays a central role in multi-thread programming
- in 1985 paging was already there in the hardware but Intel further extended the segmentation support (e.g. in the 80386 processor)
- although the segmentation logic has been significantly revised in x86-64 processors

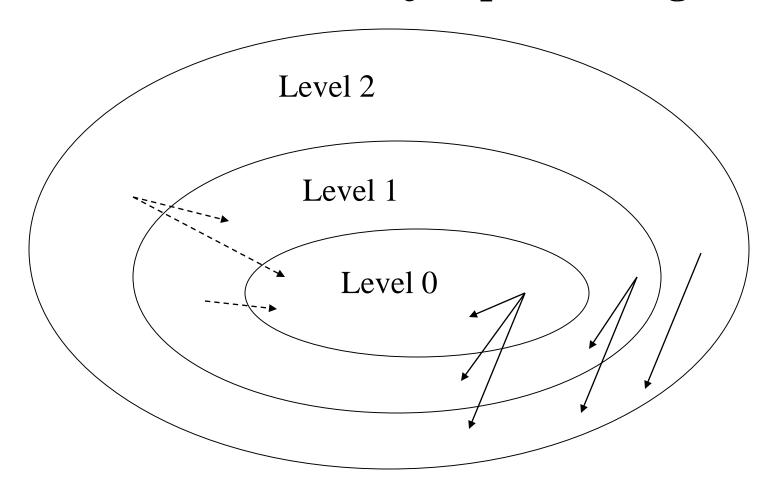
Segmentation based protection model (i)

- Each segment is associated with a given <u>protection level (or privilege level)</u>
- Each routine having protection level *h* can invoke any other routine having protection level *h*, within any segment (**this can be achieved via intra-segment and cross-segment jumps**)
- Routines having protection level h can invoke routines having protection level different from h via **cross-segment jumps**
- Cross-segment jumps always allow jumping from protection level h to protection level h+i
- Each segment having protection level *h* is associated with a set of access points, called GATEs, each one identified as *<seg.id*, *offset>*
- Any GATE is associated with a maximum level max=h+j starting from which the GATE can be passed through

Segmentation based protection model (ii)

- If level(S)=h and max(GATE(S))=h+i then segment S entails a GATE for accessing level h for modules associated with protection level up to h+i
- Cross-segment jumps <u>deny the access</u> to the destination if the source operates at protection level greater than the maximum one associated with the gate
- Overall, cross-segment jumps deny the access to the destination anytime we do not use a GATE as the destination *entry* for the jump

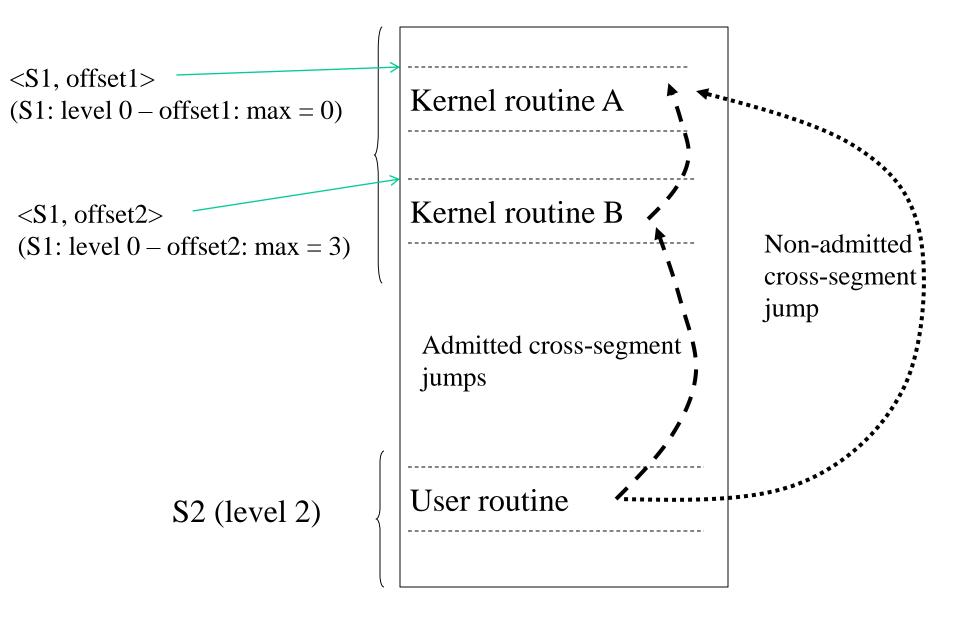
Protection levels and jumps: the ring model



→ Always admitted

Admitted depending on the *max* origin level associated with the target GATE

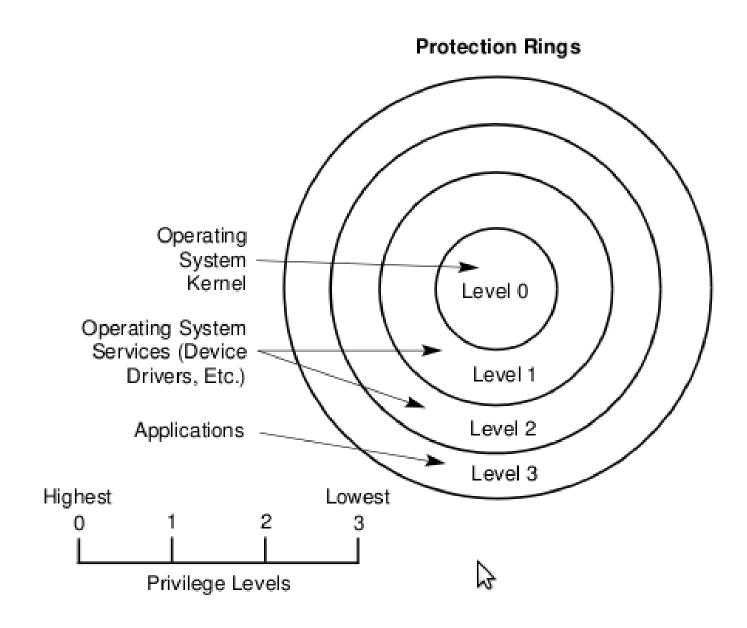
An example



Objectives of protection levels

- Denial of uncontrolled access to kernel level modules
- Kernel level access is controlled via specific "entry points" (the GATEs), which are explicitly used as destinations for jumps (more generally control flow variations) originated while running at worse protection levels
- In conventional operating systems, the entry points are typically associated with:
 - > interrupt handlers (asynchronous invocations)
 - > software traps (synchronous invocations)

Ring scheme for x86 machines

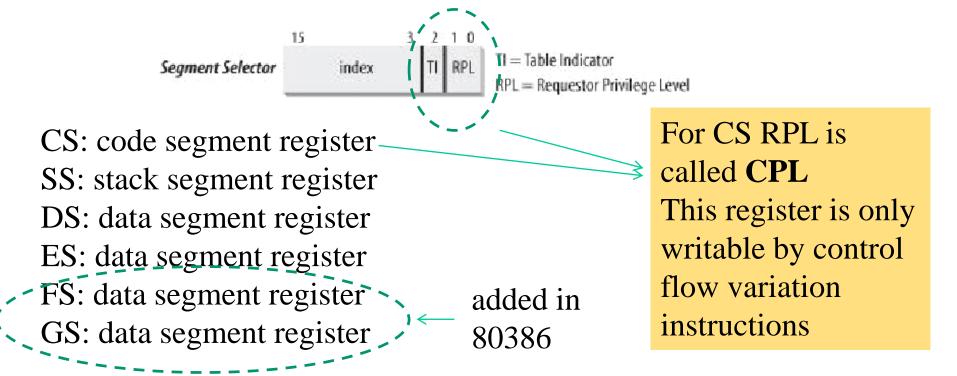


x86 address composition with segmentation

- An address does not specify the segment ID directly
- It can specify a segment-selector register
- This register keeps information on the actual segment to which we are accessing
- An example:

<segment-selector-register, displacement>

x86 details on the segmentation support



- CS (Code Segment Register) points to the current segment. The 2 lsb identify the CPL (Current Privilege Level) for the CPU (from 0 to 3).
- SS (Stack Segment Register) points to the segment for the current stack.
- DS (Data Segment Register) points to the segment containing static and global data.

Back to the very early x86 example

Here we are seamlessly (say implicitly) using <u>CS</u>, and <u>DS</u> for the first instruction and <u>CS</u> and <u>SS</u> for the second instruction

ES is an additional (to DS) implicit segment for specific classes of machine instructions, e.g. string-targeted ones like stos and movs

x86 GDT entries (segment descriptors)

31			16	15		0		
Base 0:15				Limit 0:15				This directly supports protected mode
63 56	55 52	51	48	47 40	39	32	İ	protected mode
Base 24:31	Flags	Lin 16:	nit 19	Access Byte	Base	16:23		

Access byte content:

Pr - Present bit. This must be 1 for all valid selectors.

Privl - Privilege, 2 bits. Contains the ring level (0 to 3)

Ex - Executable bit (1 if code in this segment can be executed)

.

Flags:

Gr - Granularity bit. If **0** the limit is in 1 B blocks (byte granularity), if **1** the limit is in 4 KB blocks (page granularity)

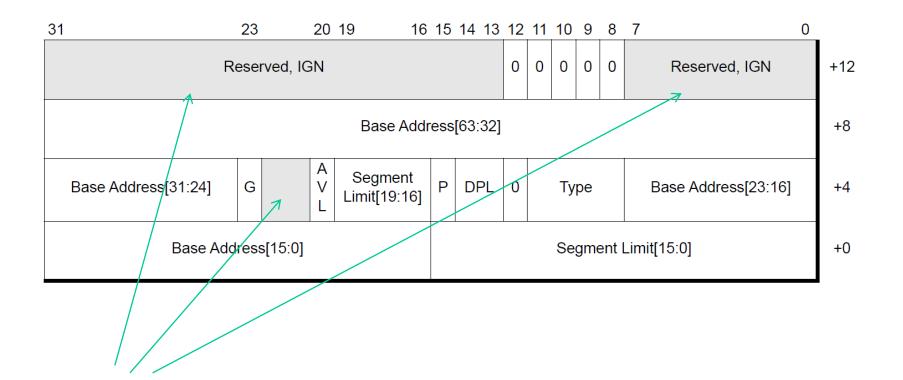
. . . .

Accessing GDT entries

- Given that a *segment descriptor* is 8 bytes in size, its relative address within GDT is computed by multiplying the 13 bits of the *index* field of *segment selector* by 8
- E.g, in case GDT is located at address 0x00020000 (value that is kept by the **gdtr register**) and the *index* value within *segment selector* is set to the value 2, the address associated with the *segment descriptor* is 0x00020000 + (2*8), namely 0x00020010

This is not only a pointer but actually a packed struct describing positioning and size of the GDT

Long mode descriptors



ignored bits

Store Global Descriptor Table Register

	Opcode	Mnemonic	Description
6	OF 01 /0	SGDT m	Store GDTR to m.

Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a 6-byte memory location. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s.

SGDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated.

See "LGDT/LIDT-Load Global/Interrupt Descriptor Table Register" in Chapter 3 for information on loading the GDTR and IDTR.

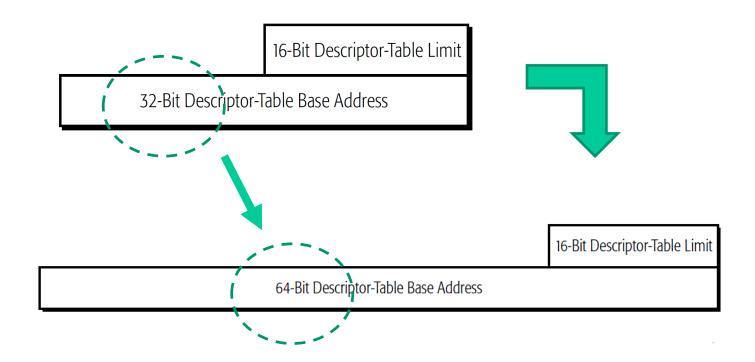
```
Operation
if(OperandSize == 16) {
    Destination[0..15] = GDTR.Limit;
    Destination[16..39] = GDTR.Base; //24 bits of base address loaded
    Destination[40..47] = 0;
}
else { //32-bit Operand Size
    Destination[0..15] = GDTR.Limit;
    Destination[16..47] = GDTR.Base; //full 32-bit base address loaded
}
```

IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 family, Pentium, Intel 486, and Intel 386 processors fill these bits with 0s.

x86 long mode provides 2 (the table size) + 8 (the table address) bytes

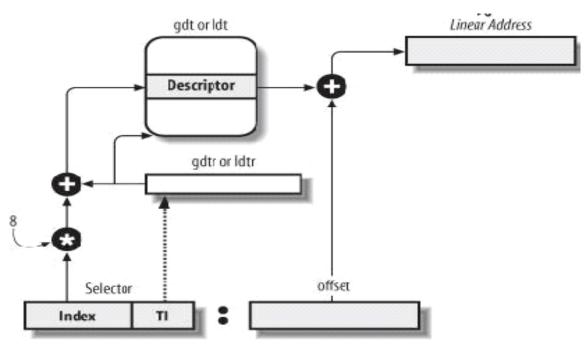
Long mode GDTR extensions



Example code

```
#include <stdio.h>
struct desc ptr {
        unsigned short size;
        unsigned long address;
} attribute ((packed));
#define store gdt(ptr) asm volatile("sgdt %0":"=m"(*ptr))
int main (int argc, char**argv) {
 struct desc ptr gdtptr;
char v[10]; //another way to see 10 bytes packed in memory
 store gdt(&gdtptr);
 store gdt(v);
printf("comparison is %d\n", memcmp(v, &qdtptr, 10));
printf("GDTR is at %x - size is %d\n", gdtptr.address, gdtptr.size);
printf("GDTR is at %x - size is %d\n",((struct desc_ptr*)v)->address,
           ((struct desc ptr*)v)->size);
```

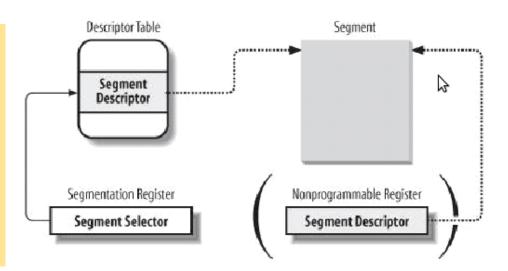
Access scheme



Logical Address

Caching of descriptors
(1 cache register per segment selector – non-programmable)

Cache line filled upon selector update



Making explicit usage of segments while coding

```
#include <stdio.h>
#define load(ptr,var) asm volatile("mov, %%ds:(%0), %%rax":"=a" (var):"a" (ptr))
#define store(val,ptr) asm volatile(" mov %0, %%ds:(%1)"\
                                       int main (int argc, char**argv) {
       unsigned long x = 16;
       unsigned long y;
                                                        explicit reference
                                                        to the data segment
       load(&x,y);
       printf("variable y has value %u\n",y);
                                                        register (DS)
       store (y+1, &x);
       printf("variable x has value u\n",x);
```

Code/data segments for LINUX

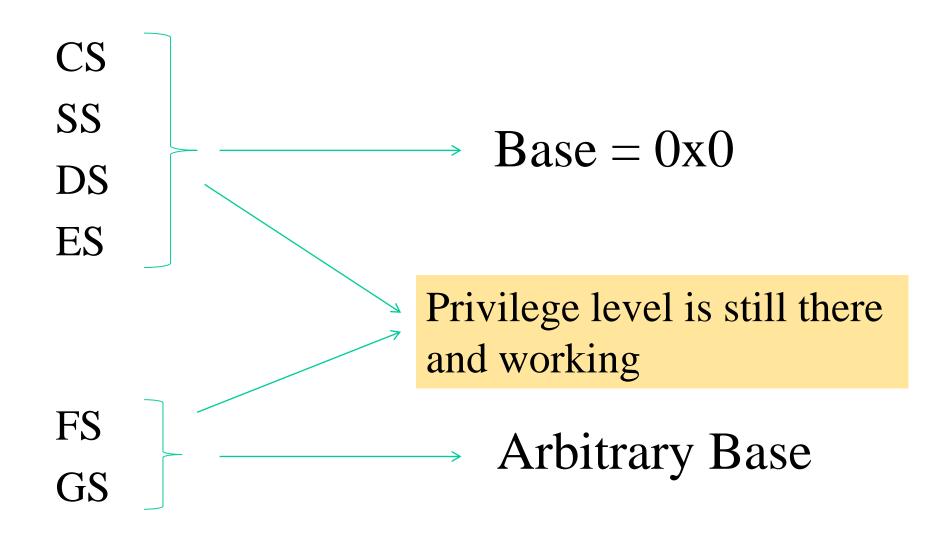
Segment	Base	G	Ļimit	s (Туре	DPL	D/B	P
user code	0:0000000	1	O:###	1/	10	3	1	1
user data	0:10000000	1	0x###	1	2	3	1	1
kernel code	0:0000000	1	0xffff	1	10	0	1	1
kernel data	0:0000000	1/	0xfffff	1	2	0	1	1
Ca	n we read/writ	Is 1	the seo	ment r	oresent?			

x86-64 directly forces base to 0x0 for the corresponding segment registers

The x86-64 revision

- Registers keeping track of segment IDs (also known as selectors) are not all managed the same way by firmware on board of the processor
- For some registers keeping segment IDs (hence for the corresponding segments in the GDT table) a fixed base of 0x0 is enforced for the segments
- Protection bits in the segment table entries associated with those segments IDs still work
- For a few registers keeping segment IDs the classical rule relying on arbitrary base values for the segments is adopted

x86-64 selector management details



Segment selectors update rules

- CS plays a central role, since it keeps the CPL (Current Privilege level)
- CS is only updated via control flow variations
- All the other segment registers can be updated if the segment descriptor they would point to after the update has DPL ≥ CPL
- Clearly, with CPL = 0 we can update everything (ring 0 has no limit)

LINUX GDT on x86

	Linux's GDT	Segment Selectors		Linux's GDT	Segment Selectors
	null	0x0		TSS	0×80
	reserved		\$	LDT	0×88
	reserved			PNPBIOS 32-bit code	0x90
	reserved			PNPBIOS 16-bit code	0×98
	not used			PNPBIOS 16-bit data	0xa0
	not used			PNPBIOS 16-bit data	0xa8
D [TLS #1	0x33	Ī	PNPBIOS 16-bit data	0xb0
Beware	TLS#2	0x3b		APMBIOS 32-bit code	0xb8
these	TL5 #3	0x43	1	APMBIOS 16-bit code	0xc0
	reserved			APMBIOS data	0xc8
	reserved			not used	1
	reserved			not used	
	kernel code	0x60 (KERNEL_CS)		not used	
	kernel data	Ox68 (KERNEL_DS)	Ī	not used	
	user code	0x73 (USER_CS)		not used	
	user data	0x7b (USER_DS)		double fault TSS	0xf8

TSS

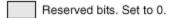
- TSS (Task State Segment): the set of linear addresses associated with TSS is a subset of the linear address space destined to kernel data segment
- each TSS (one per CPU-core) is kept within the **int_tss** array
- the *Base* field within the *n*-th core TSS register points to the *n*-th entry of the **int_tss** array (transparently via the TSS segment)
- Gr=0 while *limit is 104 * 4 bytes*
- *DPL*=0, since the TSS segment cannot be accessed in user mode

x86 TSS structure

31	15	0
I/O Map Base Address		T 100
	LDT Segment Selector	96
	GS	92
	FS	88
	DS	84
	SS	80
	cs	76
	ES	72
EC	DI .	68
ES	SI	64
EB		60
ES	SP.	56
EB		52
EC)X	48
EC	X	44
EA	X	40
EFLA	AGS	36
EI	P	32
CR3 (P	DBR)	28
	SS2	24
ES	P2	20
	SS1	16
ES	P1	12
	SS0	8
ES	P0	4
	Previous Task Link	0

Although it could be ideally used for hardware based context switches, it is not in Linux/x86

It is essentially used for privilege level switches (e.g. access to kernel mode), based on stack differentiation



x86-64 variant

offset	31-16	15-0
0x00	reserved	
0x04	RSP0 (low)	
0x08	RSP0 (high)	
0x0C	RSP1 (low)	
0x10	RSP1 (high)	
0x14	RSP2 (low)	
0x18	RSP2 (high)	

room for 64-bit stack pointers has been created sacrificing general registers snapshots



Loading the TSS register

- x86 ISA (Instruction Set Architecture) offers the instruction LTR
- This is privileged and must be executed at CPL = 0
- The TSS descriptor must be filled with a source operand
- The source can be a general-purpose register or a memory location
- Its value (16 bits) keeps the index of the TSS descriptor into the GDT

LTR — Load Task Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 00 /3	LTR <i>r/m</i> 16	M	Valid	Valid	Load $r/m16$ into task register.

Instruction Operand Encoding ¶

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

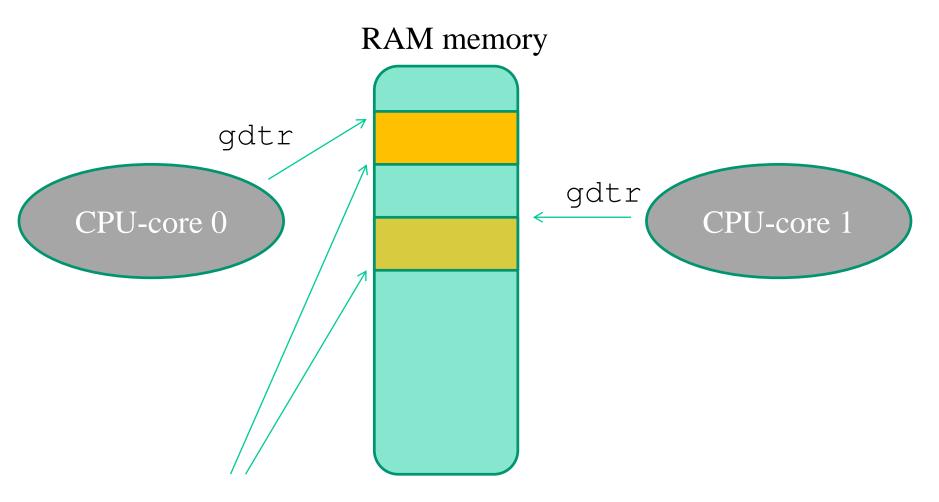
The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

GDT replication

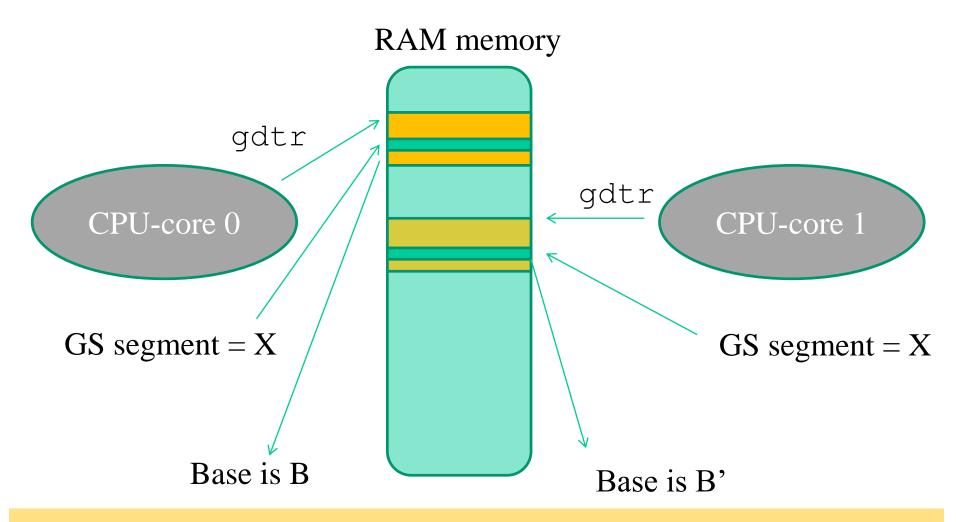
- By the discussion on TSS we might have already observed that different CPU-cores in a multi-core/multi-processor system may need to fill a given entry of the GDT with different values
- To achieve this goal the GDT is actually replicated in common operating systems, with one copy for each CPU-core
- Then each copy slightly diverges in a few entries
- The main (combined) motivations are
 - ✓ performance
 - ✓ transparency of data access separation

Actual architectural scheme



The two tables may differ in a few entries!!

Replication benefits: per-CPU seamless memory accesses



Same displacement within segment X seamlessly leads the two CPUcores to access different linear addresses

Per-CPU memory

- No need for a CPU-core to call, e.g. CPUID (... devastating for the speculative pipeline ...) to determine what memory portion is explicitly dedicated to it
- Fast access via GS segment displacing for per-CPU common operations such as
 - ✓ Statistics update (non need for LOCKED CMPXCHG)
 - ✓ Fast control operations

Per-CPU memory setup in Linux

- Based on some per-CPU reserved zone in the linear addressing scheme
- The reserved zone is displaced by relying on the GS segment register
- Based on macros that select a displacement in the GS segment
- Based on macros that implement memory access relying on the selected displacement

An example

```
DEFINE_PER_CPU(int, x);
int z;
z = this cpu read(x);
```

The above statement results in a single instruction:

To operate with no special define we can also get the actual address of the per-CPU data and work normally:

$$y = this cpu ptr(&x)$$

TLS – Thread Local Storage

- It is based on setting up different segments associated with FS and GS selectors
- Each time a thread is CPU-dispatched, kernel software restores its corresponding segment descriptors into TLS#1, TLS#2 and TLS#3 within the GDT
- We have system calls allowing us to change the segment descriptors to be posted on TLS entries

Segment management system calls (i)

NAME

top

```
arch prctl - set architecture-specific thread state
SYNOPSIS
            top
       #include <asm/prctl.h>
       #include <sys/prctl.h>
       int arch prctl(int code, unsigned long addr);
       int arch prctl(int code, unsigned long *addr);
DESCRIPTION
               top
       arch prctl() sets architecture-specific process or thread state.
       code selects a subfunction and passes argument addr to it; addr is
       interpreted as either an unsigned long for the "set" operations, or
       as an unsigned long *, for the "get" operations.
       Subfunctions for x86-64 are:
```

Segment management system calls (ii)

Subfunctions for x86-64 are:

ARCH SET FS

Set the 64-bit base for the FS register to addr.

ARCH GET FS

Return the 64-bit base value for the FS register of the current thread in the unsigned long pointed to by addr.

ARCH SET GS

Set the 64-bit base for the GS register to addr.

ARCH GET GS

Return the 64-bit base value for the GS register of the current thread in the unsigned long pointed to by addr.

RETURN VALUE top

On success, arch_prctl() returns 0; on error, -1 is returned, and errno is set to indicate the error.

x86-64 control registers

- CR0-CR3 or CR0-CR4 (on more modern x86 CPUs)
- CR0: is the baseline one
- CR1: is reserved
- CR2: keeps the linear address in case of a fault
- CR3: is the page-table pointer

CR0 structure vs long mode

Long mode uses a combination of this and the EFER (Extended Feature Enable Register) MSR (model specific register)

Bit	Name	Full Name	Description
0	PE	Protected Mode Enable	If 1, system is in protected mode, else system is in real mode
1	MP	Monitor co-processor	Controls interaction of WAIT/FWAIT instructions with TS flag in CR0
2	EM	Emulation	If set, no x87 FPU is present, if clear, x87 FPU is present
3	TS	Task switched	Allows saving x87 task context upon a task switch only after x87 instruction used
4	ET	Extension type	On the 386, it allowed to specify whether the external math coprocessor was an 80287 or 80387
5	NE	Numeric error	Enable internal x87 floating point error reporting when set, else enables PC style x87 error detection
16	WP	Write protect	When set, the CPU can't write to read-only pages when privilege level is 0
18	AM	Alignment mask	Alignment check enabled if AM set, AC flag (in EFLAGS register) set, and privilege level is 3
29	NW	Not-write through	Globally enables/disable write-through caching
30	CD	Cache disable	Globally enables/disable the memory cache
31	PG	Paging	If 1, enable paging and use the CR3 register, else disable paging

Interrupts/traps vs kernel access

- Interrupts are <u>asynchronous events</u> that are not correlated with the current CPU-core execution flow
- Interrupts are generated by external devices, and can be masked (vs non-masked)
- Traps, also known as **exceptions**, are **synchronous events**, strictly coupled with the current CPU-core execution (e.g. division by zero)
- Multiple executions of the same program, under the same input, may (but not necessarily do) give rise to the same exceptions
- Traps are (<u>actually have been historically</u>) used as the mechanism for on demand access to kernel mode (via system calls)

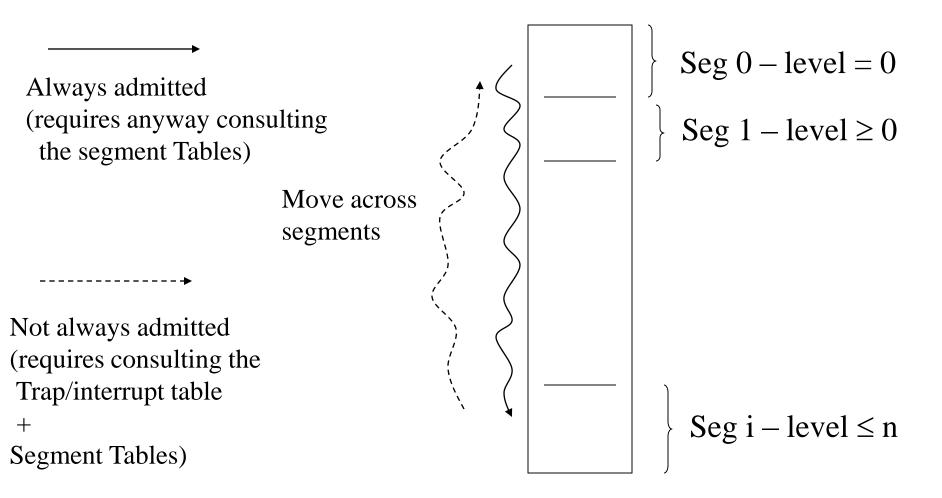
Management of trap/interrupt events

- The kernel keeps a **trap/interrupt table**
- Each table entry keeps a **GATE descriptor**, which provides information on the address associated with the GATE (e.g. <seg.id,offset>) and the GATE protection level
- The content of the trap/interrupt table is exploited to determine whether the access to the GATE can be enabled
- The check relies on the current content of CPU registers, the segment registers, which specify the current privilege level (CPL)
- In principle, it may occur that a given GATE <u>is described</u> within multiple entries of the trap/interrupt table (aliasing), possibly with different protection specifications

Summary on x86 control flow variations

- <u>intra-segment</u>: standard jump instruction (e.g. JMP < displacement > on x86 architectures)
 - Firmware only verifies whether the displacement is within the current segment boundary
- **cross-segment**: long jump instructions (e.g. LJMP <seg.id>, <displacement> on x86 architectures)
 - Firmware verifies whether jump is enabled on the basis of privilege levels (no CPL improvement is admitted)
 - Then, firmware checks whether the displacement is within the segment boundaries
- <u>cross-segment via GATEs</u>: trap instructions (e.g. INT on x86 architectures)
 - Firmware checks whether jumping is admitted depending on the privilege level associated with the target GATE as specified within the **trap/interrupt table**

An overview



GATE details for the x86 architecture (i)

- The trap/interrupt table is called Interrupt Descriptor
 Table (IDT)
- Any entry keeps
 - ➤ The ID of the target segment and the segment displacement
 - ➤ the *max* level starting from which the access to the GATE is granted
- IDT is accessible via the idtr register which is a packed structure keeping the linear address of the IDT and the size (number of entries, each made up by 8 or 16 bytes, depending on whether extended 64-bit mode is active)
- The register is loadable via the LIDT machine instruction

GATE details for the x86 architecture (ii)

- We know the **current privilege level** is kept within CS
- If protection information enables jumping, the segment ID within IDT is used to access GDT in order to check whether jumping is within the segment boundaries
- If check succeeds the current privilege level gets updated
- The new value is taken from the <u>corresponding entry</u> of GDT (this value corresponds to the privilege level of the target segment)
- The GATE description also tells whether the activated code is interruptible or not

Conventional operating systems

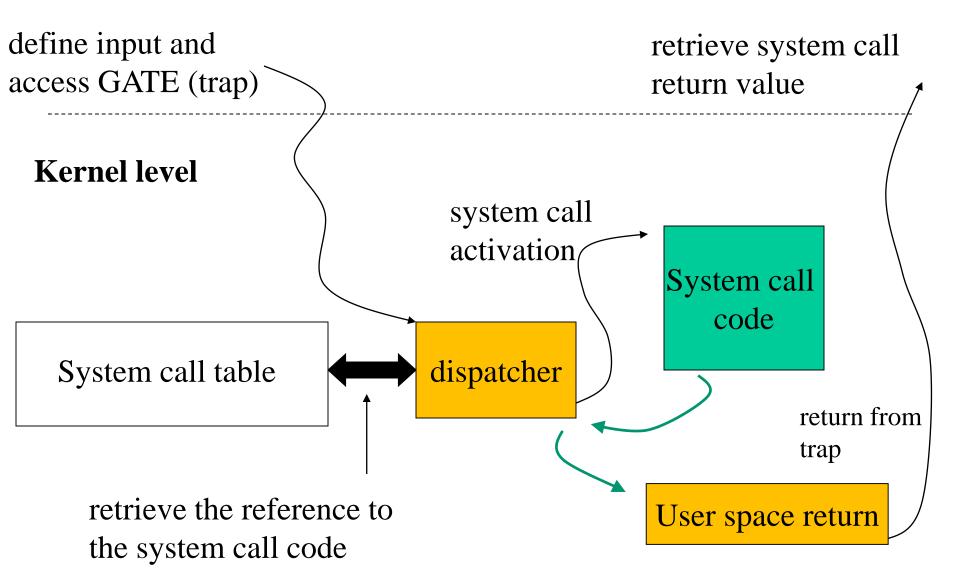
- For LINUX/Windows systems, the GATE for on-demand access (via software traps) to the kernel **is unique**
- For i386 machines the corresponding software traps are
 - > INT 0x80 for LINUX (with backward compatibility in x86-64)
 - > INT 0x2E for Windows
- Any other GATE is reserved for the management of run-time errors (e.g. divide by zero exceptions) and interrupts
- They are not usable for on-demand access via software (clearly except if you hack the kernel)
- The software module associated with the on-demand access GATE implements a dispatcher that is able to trigger the activation of the specific system call targeted by the application

Data structures for system call dispatching

- There exists a "system call table" that keeps, in any entry, the address of a specific system call
- Such an address becomes the target for a subroutine activation by the dispatcher
- To access the correct entry, the dispatcher gets in input the number (the numerical code the index) of the target system call (typically this input is provided within a CPU register)
- The code is used to identify the target entry within the system call table
- Then the dispatcher invokes the system call routine (as a "jump sub-routine" CALL instruction on x86)
- The actual system call, once executed, provides its output (return) value within a CPU register

The trap-based dispatching scheme

User level



Trap vs interruptible execution

- Differently from interrupts, **trap** management is typically configured so as not to entail/enable automatically resetting the interruptible-state for the CPU-core
- Any critical code portion associated with the management of the trap within the kernel requires explicit set of the interruptible-state bit, and the reset after job is complete (e.g. via CLI e STI instructions in x86 processors)
- For SMP/multi-core machines this <u>may not suffice</u> for guaranteeing correctness (e.g. atomicity) while handling the trap
- To address this issue, spinlock mechanisms are adopted, which are base on atomic **test-end-set code portions** (e.g., generated via the x86 LOCK prefix on standard compilation tool chains)

Test-and-set support

- Modern instruction sets offer a single instruction to atomically test-and-set memory, this is the CAS (Compare And Swap) intruction
- On x86 machines the actual CAS is called CMPXCHG (Compare And Exchange)
- ... but we already discussed of this while dealing with memory consistency!!

System call software components

- User side: software module (a) providing the input parameters to the GATE (and to the actual system call) (b) activating the GATE and (c) recovering the system call return value
- kernel side:
 - > dispatcher
 - >system call table
 - > actual system call code
- Addition of a new system call means working on both sides
- Typically, this happens with no intervention on the dispatcher in all the cases where the system call format is compliant with those predefined for the target operating system

Linux along our path

- Kernel 2.4 : highly oriented to expansibility modifiability
- Kernel 2.6: more scalable
- Kernel 3.x: more structured and secure
- Kernel 4.x, 5.x: even more secure

LINUX system calls support: the whole (long) history up to kernel 5.x

System call indexing

- This has changes along time
- We originally had the so called UNISTD_32 indexing scheme
- This is still supported in modern kernel versions (e.g. 4.x and 5.x)
- Now we have the UNISTD_64 indexing
- Given that the system call indexes are used/needed at user space, we can exploit them for user code programming via the /usr/include/asm directory (or /usr/include/x86_64-linux-gnu/asm)
- The two indexing schemes are stated in
 - unistd 32.h
 - unistd 64.h
- Two indexing schemes imply two different system call tables at kernel level, which coexist with each other (and of course two dispatchers)

UNISTD_32 listing

```
#ifndef ASM X86 UNISTD 32 H
#define ASM X86 UNISTD 32 H 1
#define
          NR restart syscall 0
#define
          NR exit 1
#define
          NR fork 2
#define
          NR read 3
#define
          NR write 4
#define
          NR open 5
#define
          NR close 6
#define
          NR waitpid 7
#define
          NR creat 8
#define
          NR link 9
#define
          NR unlink 10
#define
          NR execve 11
#define
          NR chdir 12
#define
          NR time 13
#define
          NR mknod 14
          NR chmod 15
#define
#define
          NR lchown 16
```

UNISTD_64 listing

```
#ifndef ASM X86 UNISTD 64 H
#define ASM X86 UNISTD 64 H 1
#define
          NR read 0
#define
          NR write 1
#define
          NR open 2
#define
          NR close 3
#define
          NR stat 4
#define
          NR fstat 5
#define
          NR lstat 6
#define
          NR poll 7
#define
          NR lseek 8
#define
          NR mmap 9
#define
          NR mprotect 10
#define
          NR munmap 11
#define
          NR brk 12
          NR rt sigaction 13
#define
#define
          NR rt sigprocmask 14
```

User level tasks for accessing the gate GATE

- 1. Specification of the input parameters via CPU registers (note that these include the actual system call parameters and the dispatcher ones)
- 2. ASM instructions triggering the GATE (e.g. traps)
- 3. Recovery of the return value of the systems call (upon returning from the trap associated with GATE activation)

Predefined system call formats

- These ae specified in header files that enable using GATE access functions in C
- Thes eheadre files define the standard formats for the user level module triggering access to the system GATE (namely the module that activates the system call dispatcher), each for a different value of the number of system call parameters (from 0 to 6)
- Essentially these header files contain **ASM** vs C directives and architecture specific compilation directives
- They represent a meeting point between standard C programming and machine specific ASM language (in relation to the GATE access functionality)

Code block for a standard system call with no parameter (e.g. fork()) – classical define style

```
#define syscall0(type,name) \
type name(void) \
                                     Assembler instructions
long res; \
  asm volatile ("int $0x80"
                                    Tasks to be done after the
       : "=a" ( res)
                                    execution of the assembler
       : "0" (__NR_##name));
                                    code block
  syscall_return(type,__res);
                                     Tasks preceding the assembler
                                     code block
```

Managing the return value and errno

```
/* user-visible error numbers are in the range -1 - -124:
  see <asm-i386/errno.h> */
#define syscall return(type, res) \
do { \
      if ((unsigned long) (res) >= (unsigned long) (-125)) { \
             errno = -(res); \
             res = -1; \
      return (type) (res); \
} while (0)
                                   Case of res within the
                                   interval [-1, -124]
```

Note: why the do/while(0) construct?

It is a C construct that allows to

- #define a multi-statement operation
- put a semicolon after and
- still use within an **if** statement

Code block for a standard system call with one parameter (e.g. close()) – classical define style

```
#define syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
long res; \
 asm volatile ("int $0x80" \
     : "=a" ( res) \
     : "0" ( NR ##name), "b" ((long)(arg1))); \
  syscall return(type, res); \
                              2 registers used for the input
```

Code block for a system call with six parameters – classical define style (tailored to UNISTD_32)

```
#define syscall6(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4,
         type5,arg5,type6,arg6) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4, type5 arg5, type6
arg6) \
long res; \
 asm volatile ("push %%ebp; movl %%eax,%%ebp; movl %1,%%eax; int
$0x80 ; pop %%ebp" \
       : "=a" ( res) \
       : "i" ( NR ##name), "b" ((long)(arg1)), "c" ((long)(arg2)), \
         "d" ((long)(arg3)), "S" ((long)(arg4)), "D" ((long)(arg5)), \
         "0" ((long)(arg6))); \
 _syscall_return(type,__res); \
```

We use 4 general purpose registers (eax,ebx,ecx,edx) plus the additional registers ESI e EDI, and the ebp register (<u>base pointer</u> for the current stack frame, which is saved before overwriting) and a local integer variable "i"

UNISTD_32 calling conventions for system calls

```
/*
                                      The stack layout representation
*
       0(%esp) - %ebx
                             ARGS
                                      complies with the traditional
       4(%esp) - %ecx
                                      stack based passage of
       8(%esp) - %edx
*
                                      parameters
*
       C(%esp) - %esi
      10(%esp) - %edi
      14(%esp) - %ebp
*
                              END ARGS
      18(%esp) - %eax
      1C(%esp) - %ds
                                             Ring and baseline CPU
 *
      20 (%esp) - %es
                                             state information
      24 (%esp) -- orig_eax
                                             (firmware saved onto
     -28(%esp) - %eip
                                             the system stack)
      2C(%esp) - %cs
      30(%esp) - %eflags
      34(%esp) - %oldesp
      38(%esp) - %oldss
```

UNISTD_64 calling conventions for system calls

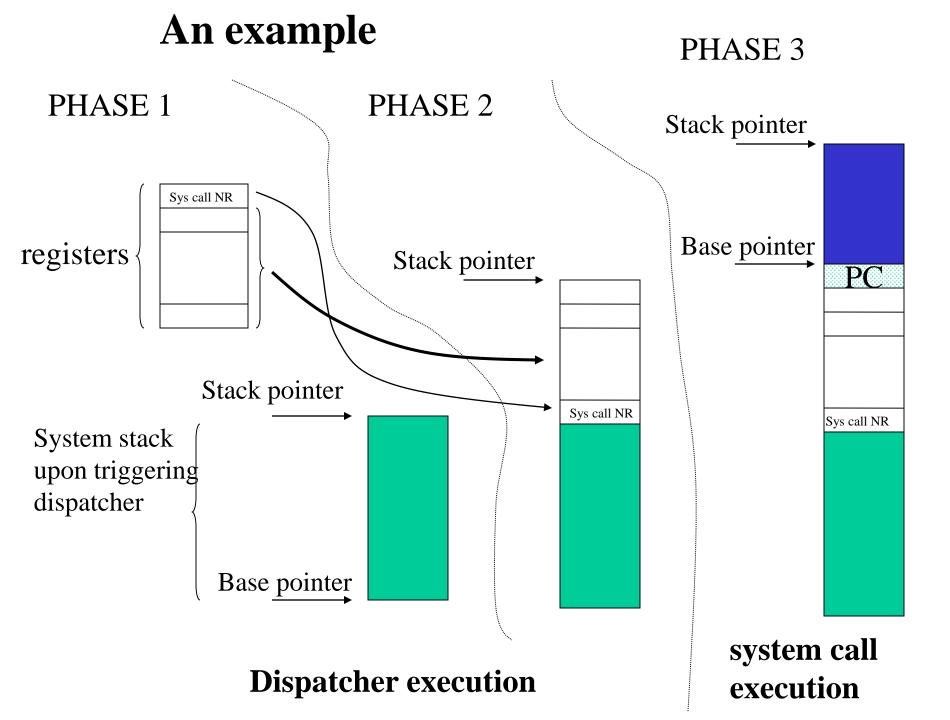
```
* Register setup:
* rax system call number
* rdi arg0
* rcx return address for syscall/sysret, C arg3
* rsi arg1
* rdx arg2
* r10 arg3 (--> moved to rcx for C)
* r8 arg4
* r9 arg5
* r11 eflags for syscall/sysret, temporary for C
* r12-r15, rbp, rbx saved by C code, not touched.
\star
* Interrupts are off on entry.
* Only called from user space.
* /
```

System V AMD64 ABI additional details

- If the callee wishes to use registers RBX, RBP, and R12–R15, it must restore their original values before returning control to the caller
- All other registers must be saved by the caller if it wishes to preserve their values

Details on passing parameters

- Once gained control, the dispatcher will take a complete snapshot of CPU registers
- The snapshot is taken within the **system level stack**
- Then the dispatcher will invoke the system call as a subroutine call (e.g. via a CALL instruction in x86 architectures)
- The actual system call will retrieve the parameters according to the proper ABI
- The taken snapshot can be modified upon the system call return (e.g. for delivering the return value)



UNISTD_32 stack alignment

```
struct pt_regs {
         unsigned long ;
          unsigned long ;
          unsigned long :
          unsigned long ;
          unsigned long :
          unsigned long bp;
          unsigned long ;
                                                       Software saved
          unsigned short di;
                                                       (no distinction between
          unsigned short __dsh;
          unsigned short ;
                                                       caller/callee save)
          unsigned short __esh;
          unsigned short !;
          unsigned short fsh;
          unsigned short :;
          unsigned short __gsh;
         unsigned long orig_ax;
         unsigned long;
          unsigned short ;
         unsigned short __csh;
          unsigned long flags;
                                                           Firmware saved
          unsigned long ;
          unsigned short ::;
          unsigned short __ssh;
```

UNISTD_64 stack alignment

```
struct pt_regs {
           /* * C ABI says these regs are callee-preserved. They aren't saved on
kernel entry * unless syscall needs a complete, fully filled "struct pt_regs". */
          unsigned long to; unsigned long to; unsigned long to;
          unsigned long | ; unsigned long bp; unsigned long | ;
/* These regs are callee-clobbered. Always saved on kernel entry. */
          unsigned long ;
          unsigned long [10];
          unsigned long ;
          unsigned long :;
          unsigned long ;
          unsigned long ;
          unsigned long :
          unsigned long ; unsigned long d;
/* * On syscall entry, this is syscall#. On CPU exception, this is error code. * On
hw interrupt, it's IRQ number: */
          unsigned long orig_ax;
/* Return frame for iretq */
          unsigned long ;
          unsigned long ;
          unsigned long flags;
          unsigned long ;
          unsigned long ;;
          /* top of stack page */
```

Firmware managed

Simple examples for adding system calls to the <u>user API</u>

Provide a C file which:

- contains the definition of the numerical codes for the new system calls
- contains (or includes) the macro-definition for creating the actual standard module associated with the new system calls (e.g. syscall0 () for UNISTD_32)

```
#include <unistd.h>
#define _NR_my_first_sys_call 254
#define _NR_my_second_sys_call 255

_syscall0(int,my_first_sys_call);
  syscall1(int,my_second_sys_call,int,arg);
```

Simple overriding of the fork () UNISTD_32 system call

```
#include <unistd.h>
#define NR my fork 2 //same numerical code as the original
#define new syscall0(name) \
int name(void) \
asm("int $0x80" : : "a" (__NR_##my_fork) ); 
return 0; \
} \
new syscall0(my fork)
int main(int a, char** b) {
       my fork();
       pause(); // there will be two processes pausing !!
```

"int 0x80" system call path performance implications

- One memory access to the IDT
- One memory access to the GDT to retrieve the kernel CS segment
- One memory access to the GDT (namely the TSS) to retrieve the kernel level stack pointer
- A lot of clock cycles waiting for data coming from memory (just to control the execution flow)
- Asymmetric delays in asymmetric hardware (e.g. NUMA)
- Unreliable outcome for time-interval measures using system calls, see gettimeofday() (and *rdtsc*)

The x86 revolution (starting with Pentium3)

- CS value for kernel code cached into an apposite MSR (Model Specific Register)
- Kernel entry point offset (the target EIP/RIP) kept into an apposite MSR
- Kernel level stack/data base kept into an apposite MSR
- Entering kernel code is as easy as flushing the MSRs values onto the corresponding original registers (e.g. CS, DS, SS recall that the corresponding bases are defaulted to 0x0)
- No memory access for activating the system call dispatcher
- This is the fast system call path!!

Fast system call path additional details

SYSENTER instruction for 32 bits - **SYSCALL** instruction for 64 bits

based on (pseudo) register manipulation

- CS register set to the value of (SYSENTER_CS_MSR)
- EIP register set to the value of (SYSENTER_EIP_MSR) for 32 bits
- SS register set to the sum of (8 plus the value in SYSENTER_CS_MSR for 32 bits another bitmask for 64 bits)
- ESP register set to the value of (SYSENTER_ESP_MSR)

Details

SYSCALL — Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 05	SYSCALL	ZO	Valid	Invalid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

Instruction ordering. Instructions following a SYSCALL may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSCALL have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Fast system call path additional details

SYSEXIT instruction for 32 bits - **SYSRET** instruction for 64 bits

based on pseudo register manipulation

- CS register set to the sum of (16 plus the value in SYSENTER_CS_MSR for 32 bits another bitmask for 64 bits)
- EIP register set to the value contained in the EDX register for 32 bits RCX for 64 bits
- SS register set to the sum of (24 plus the value in SYSENTER_CS_MSR for 32 bits another bitmask for 64 bits)
- ESP register set to the value contained in the ECX register for 32 bits

MSR and their setup

/usr/src/linux/include/asm/msr.h:

```
101 #define MSR_IA32_SYSENTER_CS 0x174
 102 #define MSR IA32 SYSENTER ESP 0x175
 103 #define MSR_IA32_SYSENTER_EIP 0x176
/usr/src/linux/arch/i386/kernel/sysenter.c:
36 wrmsr(MSR_IA32_SYSENTER_CS, __KERNEL_CS, 0);
37 wrmsr(MSR_IA32_SYSENTER_ESP, tss->esp1, 0);
38 wrmsr(MSR IA32 SYSENTER EIP,
                         (unsigned long) sysenter_entry, 0);
```

rdmsr and wrmsr are the actual machine instructions for reading/writing the registers

The syscall () construct (since Pentium3/kernel 2.6)

- syscall() is implemented within glibc (in stdlib.h)
- It allows triggering a trap to the kernel for the execution of a generic system call
- The first argument is the system call number
- The other parameters are the input for the system call code
- The actual ASM code implementation of syscall() is targeted and optimized for the specific architecture
- Specifically, the implementation (including the kernel level counterpart) relies on ASM instructions such as sysenter/sysexit or syscall/sysret, which have been made available starting from Pentium3 processors

An example

```
#include <stdlib.h>
#define NR my first sys call 333
#define NR my second sys call 334
int my first sys call() {
       return syscall( NR my first sys call);
int my second sys call(int arg1){
       return syscall ( NR my second sys call, arg1);
int main(){
        int x;
       my first sys call();
       my second sys call(x);
```

The system call table

- It is an array of function pointers
- However, we cannot easily resize the array and recompile the kernel
- This is because that table (like may other kernel level data structures) is positioned at compile time in specific zones of virtual addresses
- Simple enlarging on the table with no other modification of the kernel compilation layout will lead to data structures' overlap
- Such strict compilation rules depend on the fact that hardware setup for running the kernel may require CPU registers to be populated with compile time defined values
- The before described fast system call path is a clear example!!

System call table hacking: entry reusage

- In older versions of the kernel the system call table was oversized, just for kernel modifiability purposes
- Hence the addition of new system calls in the kernel software could be based on the free entries
- In current (or more recent) kernel versions no oversize is put in place
- This is because the less "free" zones of data structures exist, the less the likelihood that they can be exploited against security
- But we are lucky because a few entries, although reserved, are not actually used to point to actual kernel level functions
- In the essence this is the scenario of kernel services that were planned (with given indexing) but not actually implemented
- All these entries point to the so called "sys_ni_syscall" kernel module, which simply returns upon its invocation

x86 system call table details

- For kernel 2.4 and i386 machines the system call table is defined in arch/i386/kernel/entry.S
- For kernel 2.6.xx the table is posted on the file arch/x86/kernel/syscall table32.S
- For kernel 4.15.xx and UNISTD_64 the table pointer is defined in /arch/x86/entry/syscall 64.c
- The .S files contain pre-processor ASM directives
- Any table entry keeps a symbolic reference to the kernel level name of a system call (typically, the kernel level name resembles the one used at application level)
- The above files (or other .S) also contains the code block for the dispatcher associated with the kernel access GATE

Table structure – i386/UNISTD_32 style

```
ENTRY (sys call table)
       .long SYMBOL NAME(sys ni syscall) /* 0 - old "setup()"
system call*/
       .long SYMBOL NAME(sys exit)
       .long SYMBOL NAME(sys fork)
       .long SYMBOL NAME(sys read)
       .long SYMBOL NAME(sys write)
                                                 /* 5 */
       .long SYMBOL NAME(sys open)
       .long SYMBOL NAME(sys close)
       .long SYMBOL NAME (sys sendfile64)
       .long SYMBOL NAME(sys ni syscall) /* 240 reserved for futex
       .long SYMBOL NAME(sys ni syscall) /* 252
sys set tid address */
                              New symbols need to be inserted here
       .rept NR syscalls-(.-sys call table)/4
              .long SYMBOL NAME (sys ni syscall)
```

.endr

Definition of system call symbols

• For the previous example, the actual system call specification will be

```
.long SYMBOL_NAME(sys_my_first_sys_call)
.long SYMBOL NAME(sys my second sys call)
```

- The actual code for the system calls (generally based exclusively on C with compilation directives for the specific architecture) can be included within new modules added to the kernel or within already exiting modules
- The actual code can rely on the kernel global data structures and on functions already available within the kernel, except for the case where they are explicitly masked (e.g. masking with static declarations external to the file containing the system call)

Definition of the system call table – UNISTD_64 style

• The kernel level source file that defines the system call table is arch/x86/entry/syscall_64.c

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
      [0 ... __NR_syscall_max] = &sys_ni_syscall,
      #include <asm/syscalls_64.h>
};

After the "include" expansion
```

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    [0] = sys_read,
    [1] = sys_write,
    [2] = sys_open,
    ...
    ...
    ...
};
```

Compilation directives for kernel side systems calls

- Specific directives are used to make the system call code compliant with the dispatching rules
- Compliance is assessed on the basis of how the input parameters are passed/retrieved
- The input parameters passage by convention historically took place via the kernel stack
- The corresponding compilation directive is asmlinkage
- This directive is now mapped to the current ABI
- Hence for the previous examples we will have the following system call definitions

```
asmlinkage long sys_my_first_sys_call() { return 0;}
asmlinkage long sys_my_second_sys_call(int x) {
    return ((x>0)?x:-x);}
```

The ni_sys_call module

```
asmlinkage long sys_ni_syscall(void) {
    return -ENOSYS;
}
```

The actual dispatcher (trap driven activation – i386/UNISTD_32)

```
Manipulating
ENTRY(system_call)
                                                         the CPU
       pushl %eax
                                    # save orig_eax
                                                         snapshot in
       SAVE_ALL
       GET_CURRENT(%ebx)
                                                         the stack
       testb $0x02,tsk_ptrace(%ebx)
                                    #PT_TRACESYS
       jne tracesys
                                       Beware this!!!
       cmpl $(NR_syscalls),%eax 
       jae badsys
       call *SYMBOL_NAME(sys_call_table)(,%eax,4) _
    movl %eax,EAX(%esp)
                                    # save the return value
ENTRY(ret_from_sys_call) ----
       cli
                                    # need_resched and signals atomic test
       cmpl $0,need_resched(%ebx)
       jne reschedule
       cmpl $0,sigpending(%ebx)
       jne signal_return
restore all:
       RESTORE_ALL
```

The actual dispatcher (syscall driven activation – UNISTD_64/kernel 2.4)

```
ENTRY(system_call)
         swapgs
                                                          #define PDAREF(field) %gs:field
                   %rsp,PDAREF(pda_oldrsp)
         movq
                   PDAREF(pda kernelstack),%rsp
         movq
         sti
         SAVE_ARGS 8,1
         movq %rax,ORIG_RAX-ARGOFFSET(%rsp)
         movq %rcx,RIP-ARGOFFSET(%rsp)
         GET CURRENT(%rcx)
                                                              Part of the stack switch
         testl $PT TRACESYS,tsk ptrace(%rcx)
         jne tracesys
                                                              work originally done
         cmpq $ NR syscall max,%rax
                                                              via firmware is moved
         ja badsys
         movq %r10,%rcx
                                                              to software
         call *sys call table(,%rax,8) # XXX:
                                                 rip relative
         movq %rax,RAX-ARGOFFSET(%rsp)
         .globl ret from sys call
ret from sys call:
sysret with reschedule:
                                                   Beware this!!!
         GET_CURRENT(%rcx)
         cli
         cmpq $0,tsk_need_resched(%rcx)
         jne sysret_reschedule
         cmpl $0,tsk_sigpending(%rcx)
         jne sysret_signal
```

sysret_restore_args:

User vs kernel GS segment

SWAPGS — **Swap GS Base Register**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 01 F8	SWAPGS	ZO	Valid	Invalid	Exchanges the current GS base register value with the value contained in MSR address C0000102H.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (IA32_KERNEL_GS_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32_KERNEL_GS_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The IA32_KERNEL_GS_BASE MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. The WRMSR instruction ensures that the IA32_KERNEL_GS_BASE MSR contains a canonical address.

... moving to kernel 4.xx or later

Snippet taken from

https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry 64.S

```
ENTRY (entry SYSCALL 64)
        UNWIND HINT EMPTY
         * Interrupts are off on entry.
         * We do not frame this tiny irq-off block with TRACE IRQS OFF/ON,
         * it is too small to ever cause noticeable irq latency.
        swapgs
         * This path is only taken when PAGE TABLE ISOLATION is disabled so it
         * is not required to switch CR3.
        movq
                %rsp, PER CPU VAR (rsp scratch)
                PER CPU VAR(cpu current top of stack), %rsp
        movq
        /* Construct struct pt regs on stack */
        pushq
               $ USER DS
                                                 /* pt regs->ss */
               PER CPU VAR (rsp scratch)
        pushq
                                                 /* pt regs->sp */
        pushq
                %r11
                                                 /* pt regs->flags */
                                                 /* pt regs->cs */
        pushq
                $ USER CS
                                                 /* pt regs->ip */
        pushq
                %rcx
GLOBAL (entry SYSCALL 64 after hwframe)
                                                 /* pt regs->orig ax */
        pushq
        PUSH AND CLEAR REGS rax=$-ENOSYS
        TRACE IRQS OFF
        /* IRQs are off. */
                %rax, %rdi
                                         /* returns with IROs disabled */
                                         /* we're about to change IF */
        TRACE IRQS IRETQ
         * Try to use SYSRET instead of IRET if we're returning to
         * a completely clean 64-bit userspace context. If we're not,
         * go to the slow exit path.
         */
```

Here we pass control to a C-stub, not to the actual system call

Snippet taken from

295

#endif

https://github.com/torvalds/linux/blob/master/arch/x86/entry/common.c

```
#ifdef CONFIG X86 64
271
272
       visible void do syscall 64(unsigned long nr, struct pt regs *regs)
273
              struct thread info *ti;
274
275
              enter_from_user_mode();
276
              local_irq_enable();
277
              ti = current_thread_info();
278
              if (READ_ONCE(ti->flags) & _TIF_WORK_SYSCALL_ENTRY)
279
                      nr = syscall trace enter(regs);
280
281
282
               * NB: Native and x32 syscalls are dispatched from the same
283
               * table. The only functional difference is the x32 bit in
284
               * regs->orig ax, which changes the behavior of some syscalls.
285
286
287
                (likely(nr < NR_syscalls)) {
288
                      nr = array_index_nospec(nr, NR_syscalls);
289
                      regs->ax = sys_call_table[nr](regs);
290
291
292
              syscall_return_slowpath(regs);
293
294
```

Wrong-speculation cannot rely on arbitrary sys-call indexes!!!!

Also, from kernel 4.17 the system call table entry no longer points to the actual system call code, rather to another wrapper that masks from the stack non-useful values

Overall

- For more security-oriented implementations we have
 - ✓ More strict checks and manipulation of the user provided information before any action is taken
 - ✓ A more layered architecture for better decoupling user/kernel information flows
- The latter point has reflection on programming aspects since for, e.g., Kernel 4.17 the kernel-side creation of a new system call should be based on kernel level macros for implementing a stubbased execution of the native system-call code
- These macros are SYSCALL_DEFINEO, SYSCALL_DEFINE1, SYSCALL DEFINE2, SYSCALL DEFINE3

Actual usage/effect of kernel-side sys-call macros

• The SYSCALL_DEFINE2 example (still representative of other macros)

SYSCALL_DEFINE2(name, param1type, param1name, param2type, param2name){
 actual body implementing the kernel side system call

The macro creates a function sys_name (aliased by SyS_name) or x86 sys name from kernel 4.17

In 4.17 this function passes only the requested values (i.e. param1name and param2name) to the actual function related to the above specified body - such a function has now name se sys name

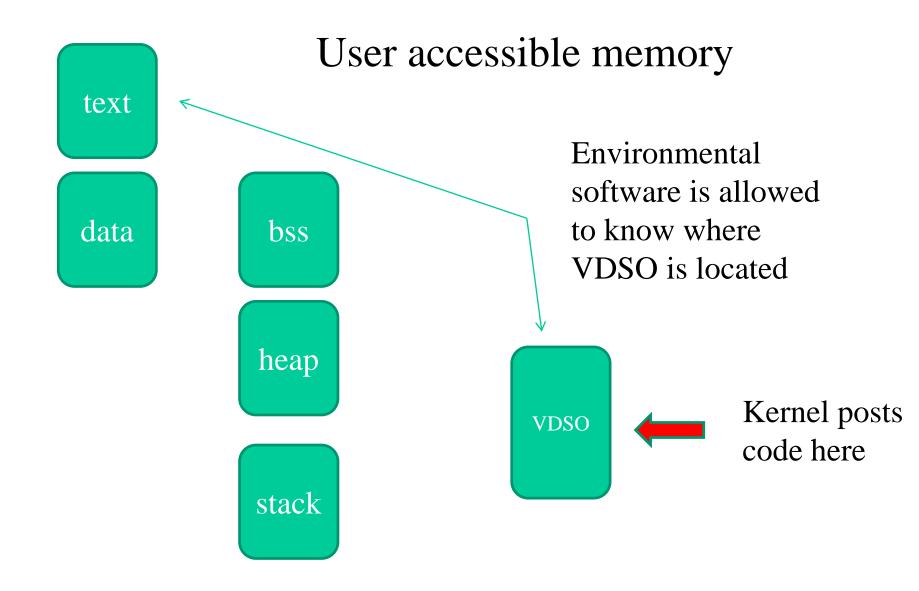
Finally ... PTI (Page Table Isolation)

```
ENTRY(entry_SYSCALL_64)
             UNWIND HINT EMPTY
147
              * Interrupts are off on entry.
              * We do not frame this tiny irq-off block with TRACE_IRQS_OFF/ON,
              * it is too small to ever cause noticeable irg latency.
                                                                                      Switch to the kernel
             swapgs
             /* tss.sp2 is scratch space. */
                          DER COU VAR(cou tss rw + TSS_sp2)
                                                                                      view of memory
             SWITCH TO KERNEL_CR3 scratch_reg=%rsp
                     PER_CPU_VAR(cpu_current_top_of_stack), %rsp
             /* Construct struct pt_regs on stack */
             pushq $_USER_DS
                                                           /* pt_regs->ss */
             pushq
                     PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
                                                           /* pt regs->sp */
             pushq
                     %r11
                                                           /* pt regs->flags */
                     $ USER CS
                                                           /* pt regs->cs */
             pusha
             pushq %rcx
                                                           /* pt_regs->ip */
      GLOBAL(entry_SYSCALL_64_after hwframe)
             pushq %rax
                                                           /* pt regs->orig ax */
             PUSH_AND_CLEAR_REGS rax=$-ENOSYS
             TRACE IROS OFF
             /* IRQs are off. */
                     %rax, %rdi
             mova
174
                     %rsp, %rsi
             movq
             call
                     do_syscall_64
                                            /* returns with IRQs disabled */
                                            /* we're about to change IF */
             TRACE IRQS IRETQ
             /*
              * Try to use SYSRET instead of IRET if we're returning to
              * a completely clean 64-bit userspace context. If we're not,
```

Virtual Dynamic Shared Object (VDSO)

- Kernel also setups system call entry/exit points for user processes
- Kernel creates a single page (or a few) in memory and attaches it to all processes' address space when they are loaded into memory.
- This page contains the actual implementation of the system call entry/exit mechanism
- Kernel calls this page virtual dynamic shared object (VDSO)
- Originally exploited for making the fast system call path available (in relation to a few services)

VDSO and the address space



Application exposed facilities

SYNOPSIS #include <sys/auxv.h>

void *vdso = (uintptr_t) getauxval(AT_SYSINFO_EHDR);

DESCRIPTION

The "vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications. Applications usually do not need to concern themselves with these details as the vDSO is most commonly called by the C library. This way you can code in the normal way using standard functions and the C library will take care of using any functionality that is available via the vDSO.

The actual VDSO (getcpu example)

```
E quaglia@paco: ~
                                                                                           o_getcpu@@LINUX_2.6+0xffffffffffffc18c>
        44 39 2b
                                         %r13d, (%rbx)
 aed:
                                  cmp
 af0:
                                         a9e <__vdso_clock_gettime@@LINUX_2.6+0x7e>
                                  jne
        75 ac
 af2:
        4c 89 d0
                                         %r10,%rax
                                  mov
 af5:
                                         %cl,%rax
        48 d3 e8
                                  shr
        48 3d ff c9 9a 3b
                                         $0x3b9ac9ff,%rax
 af8:
                                  cmp
 afe:
        Of 86 d1 01 00 00
                                         cd5 <__vdso_clock_gettime@@LINUX_2.6+0x2b5>
                                  jbe
 b04:
        31 d2
                                         %edx,%edx
                                  xor
                                         $0x3b9aca00.%rax
 b06:
        48 2d 00 ca 9a 3b
                                  sub
        83 c2 01
                                         $0x1.%edx
 b0c:
                                  add
 b0f:
        48 3d ff c9 9a 3b
                                         $0x3b9ac9ff,%rax
                                  cmp
 b15:
        77 ef
                                         b06 <__vdso_clock_gettime@@LINUX_2.6+0xe6>
                                  ja
 b17:
        48 01 16
                                         %rdx,(%rsi)
                                  add
 b1a:
        48 89 46 08
                                         %rax,0x8(%rsi)
                                  mov
 b1e:
        8b 44 24 0c
                                         0xc(%rsp), %eax
                                  mov
 b22:
        85 c0
                                  test
                                         %eax,%eax
        Of 85 58 ff ff ff
                                         a82 <__vdso_clock_gettime@@LINUX_2.6+0x62>
 b24:
                                  ine
 b2a:
        49 63 fb
                                  movslq %r11d,%rdi
 b2d:
        b8 e4 00 00 00
                                         $0xe4,%eax
                                  mov
                                  syscall
 b32:
        0f 05
        48 8d 65 e0
 b34:
                                         -0x20(%rbp),%rsp
                                  lea
 b38:
        5b
                                         %rbx
                                  pop
 b39:
        41 5c
                                         %r12
                                  pop
 b3b:
        41 5d
                                         %r13
                                  pop
        41 5e
 b3d:
                                         %r14
                                  pop
 b3f:
                                         %rbp
        5d
                                  pop
 b40:
        c3
                                  retq
 b41:
        f3 90
                                  pause
 b43:
                                         (%rbx), %eax
        8b 03
                                  mov
 b45:
                                         $0x1,%al
        a8 01
                                  test
 b47:
        75 f8
                                         b41 <__vdso_clock_gettime@@LINUX_2.6+0x121>
                                  jne
 b49:
        48 8b 15 70 c5 ff ff
                                         -0x3a90(%rip),%rdx
                                                                     # fffffffffffd0c0 <__vds
                                  mov
o_getcpu@@LINUX_2.6+0xfffffffffffffc1b0>
 b50:
                                         %rdx,(%rsi)
        48 89 16
                                  mov
 b53:
        48 8b 15 6e c5 ff ff
                                         -0x3a92(%rip),%rdx
                                                                     # fffffffffffd0c8 <__vds
                                  mov
                                                                                 へ 恒 ワ 切 10/11/2020
```

Performance effects

- The VDSO exploits flat (linear) addressing proper of operating system memory managers in order to bypass segmentation and the related operations
- It therefore reduces the number of accessed to memory in order to support the change to kernel mode
- Studies show that the reduction of clock cycles for system calls can be of the order of 75%
- This is in the end typical for any usage of the fast system call path

The current picture

- VDSO is now used to replace the old facilities supported via the **vsyscall** section, say support for specific system calls (e.g. query system calls such as gettimeofday())
- VDSO is randomized (in terms of positioning into the address space) so security gets increased
- The system call mechanism in the wide, which relies on sysenter/syscall and sysexit/sysret, is in charge of the dynamic linker (ld-linux.so)

Back to the coexistence of slow and fast system call paths

Slow path

- ✓ Still based on int 0x80
- ✓ Still accessing IDT/GDT (which is the reason why the target entry still requires to be populated)
- ✓ The kernel level system call dispatcher accesses the i386/INISTD_32 system call table

Fast path

- ✓ Base on the syscall instruction (no IDT/GDT access)
- ✓ The kernel level dispatcher (different from the previous one) accesses the x86-64/UNISTD_64 system call table

Kernel software organization

- About the 80-90% of the actual code for system calls is embedded within a few main portions of the kernel archive
- These are contained in the following directories
 - kernel (process and used management)
 - > mm (basic memory management)
 - > ipc (interprocess communication management)
 - > fs (virtual file system management)
 - > net (network management)

Kernel compiling

- You can exploit make
- It executes a set of tasks (compilation, assembly and linking tasks) which are specified via a Makefile
- This file can specify differentiated actions to be done (possibly exhibiting dependencies) which are described within a field called **target**
- Each action can be specified by the following syntax:

```
action-name: [ dependency-name] * { new-line }
{tab} action-body
```

• Further, we can define variables via the syntax:

```
variable-name = value
```

Any variable can be accessed via the syntax:

```
$ (variable-name)
```

Standard compilation steps (current tyle)

```
make config (or menuconfig)
make
make modules
make modules_install (ROOT)
make install (ROOT)
mkinitrd (or mkinitramfs) –o initrd.img-<vers> <vers>
```

update-grub

OR

grub-mkconfig -o /boot/grub/grub.cfg (ROOT)

About 'config'

- The possibilities
 - allyesconfig (likelihood of conflicting modules)
 - allnoconfig (likelohood of non-sufficient services in the kernel image)
 - Answer to the individual questions you may be asked for
 - Retrieve a good configuration file (depending on you machine/settings) on the web
 - Reuse the configuration files(s) you find in the
 /boot directory of your root file system (likely works when recompiling the same kernel version you already have)

Role of initrd

- It is a RAM disk
- It can be (temporary) mounted as the root file system and programs can be run from it
- A different root file system can be then mounted from a different device
- The previous root (from initrd) can then be moved to a directory and can be subsequently unmounted
- With initrd system startup can occur in two phases
 - the kernel initially comes up with a minimum set of compiled-in drivers
 - additional modules are loaded from initrd

Step effects

```
make config (or menuconfig)
make
```

make modules

```
make modules_install (ROOT) (writes into /lib/modules)
```

make install (ROOT) (writes into /boot: the kernel image, the system map and the config file)

update-grub

OR

grub-mkconfig -o /boot/grub/grub.cfg (ROOT)

"Extended" Kernel compilation (current style)

- Makefile updates
 - 1. setting of the EXTRAVERSION variable (non-mandatory)
 - 2. use obj-directive to add a file or a directory into the compilation tree
 - 3. the addition is within already available makefiles (or new ones)

Kernel anatomy: the system map

- It contains the symbols and the corresponding virtual memory reference (as determined at compile/link time beware randomization) for:
 - Kernel functions (steady state ones)
 - Kernel data structures
- Each symbol is also associated with a tag that defines the 'storage class' as determined by the compiling process
- As an example, 'T' usually denotes a global (non-static but not necessarily exported) function, 't' a function local to the compilation unit (i.e. static), 'D' global data, 'd' data local to the compilation unit. 'R' and 'r' same as 'D'/'d' but for read-only data

System map applications

- Kernel debugging
- Kernel run-time hacking
- The system map is also (partially) reported by the (pseudo) file /proc/kallsysm
- The latter is exploited for run-time kernel 'hacking' via the modules' technology

Just an example

2.6.5-7.282-smp #1 SMP i686 i686 i386 GNU/Linux

c03a8a00 D sys_call_table

Read/write data

2.6.32-5-amd64 #1 SMP x86_64 GNU/Linux

ffffffff81308240 R sys_call_table

Read-only data