CS5250 Cheatbook

Function of OS:

1. Manage a computer system's resources
2. Provide user interface
3. Provide service for applications
4. Hardware Abstraction
5. Protection (Kernel vs User mode)
6. Sharing (between users/processes) and data exchange
7. Caching (Hardware and Software)
8. Interrupt + Trap (Exception) Handling

Types of OS based on functionality:

1. Batch OS (data centre)
2. Interactive (Android)
3. RTOS (Embedded System, Contiki)
4. Hybrid

Types of OS based on its structure:

1. Monolithic

Single large process, running in a single address space, which is the kernel space, loaded as a single binary file during bootloading

2. Microkernel

Smaller the monolithic, only supports IPC, virtual memory, scheduling, protection, interrupt handling), the rest like FS, memory mgmt, etc is handled by users as services and they need to communicate using IPC.

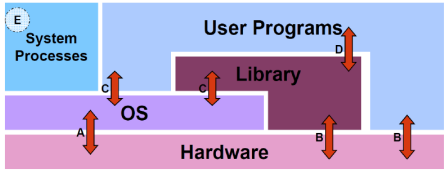
Kernel is smaller and better security advantage because less attack vectors (OS only has the minimal stuff to do its jobs)

Problem: a lot of communications between user processes. Providing services in a microkernel system are expensive compared to the normal monolithic system.

Context switches or function calls needed when the drivers are implemented as procedures (require 2 OS kernel crossing for an IPC)

3. Network OS
4. Distributed OS
5. Exokernels (even simpler than microkernel, users can access the hardware directly)

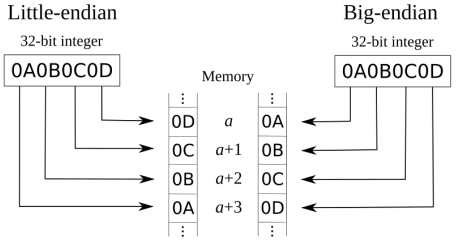
Generic OS Structure



- **A**: OS executing machine instructions
- **B**: normal machine instructions executed (program/library code)
- **C**: calling OS using ***system call interface***
- **D**: user program calls library code
- **E**: system processes
  - Provide high level services, usually part of OS

Intel x86 Programming

- Endianness (Little Endian just means it will store the LSB first, vice versa with Big Endian)



List of Common Registers

Size (in Bits)			
64	32	16	8
RAX	EAX	AX	AH/AL
RBX	EBX	BX	BH/BL
RCX	ECX	CX	CH/CL
RDX	EDX	DX	DH/DL
RDI	EDI	DI	DIL
RSI	ESI	SI	SIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8∼R15	R8D∼R15D	R8W∼R15W	R8L∼R15L

Common purpose
32-bit: EAX (Accumulator), EBX (Base Index), ECX (Counter), EDX (Data), ESI (Source Index), EDI (Destination Index), ESP (Stack Pointer), EBP (Base Pointer), EIP (Instruction Pointer)

There are other control registers (CR0 - 4), FLAGS registers, 6 16-bit segments registers (CS, DS, SS, ES, FS, GS)

64 bits registers can only be activated in long-mode: REX, a feature in the x86-64 architecture that allows for extended register access

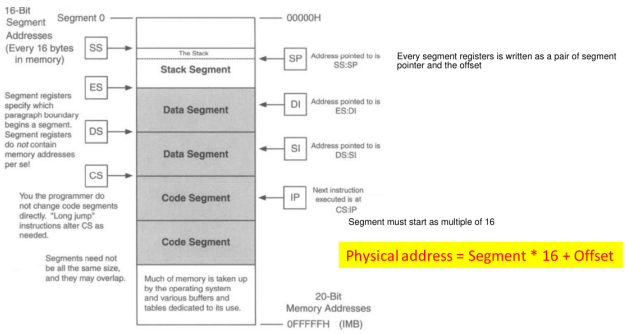
Operating Modes

1. Real Address Mode: no virtual memory (after power-up) => use real-address mode addressing
2. Protected Mode: virtual memory + privileges ring
3. Sys Management Mode: like protected but get more accesses to the system => use real-address mode addressing
4. Virtual-8086: for VM
5. Intel 64 IA-32e 64bit linear addressing

Addressing Modes to Linear Address

1. Flat Model (linear address => linear address space)
2. Segmented Model (Offset + Segment Selector)
3. Real-address mode (offset + segment selector where each segment is of equal size) => Physical Address = Segment * 16 + offset address

In this mode, the processor uses a 20-bit address bus to address memory directly, allowing it to access up to 2^{20} or 1 MB.



Real-Mode: Example: CS:IP, CS = 0xA01, IP = 0x20
Real address = CS * 16 + IP = 0xA030
Note: Offset limited to 16 bits
Each segment at most 64KB
Total memory limited to 1MB

Segmented Model in Protected Mode

1. Note that Segment Selector is 16 bit

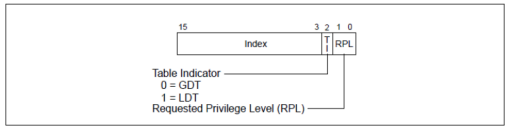
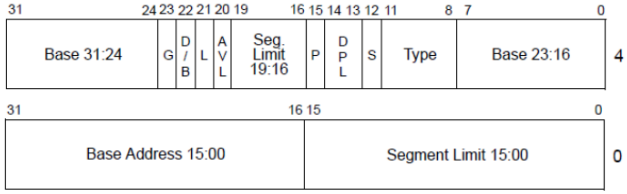
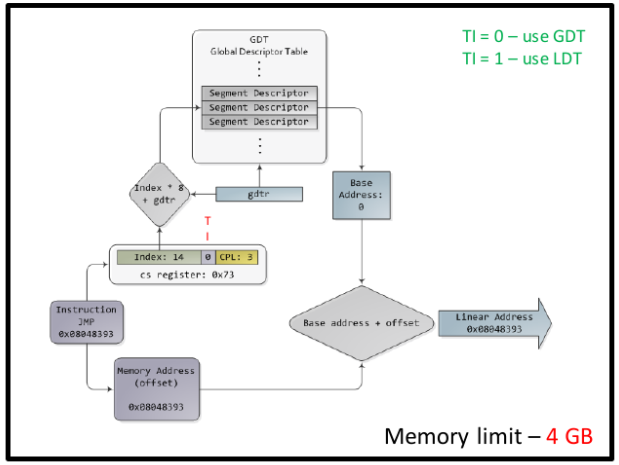


Figure 3-6. Segment Selector

2. Note that each segment descriptor in IA-32 is 8 bytes (since last 3 bits in segment selector are used)



Protected mode



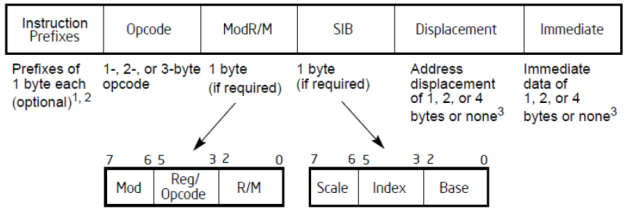
Protection Ring

0 => highest (for kernel)
3 => lowest (for user)

To transit between protection rings, check interrupt and exceptions handling

Instruction Encoding

1. Variable length encoding



One-byte opcode

Table A-2. One-byte Opcode Map: (00H — F7H) *

	0	1	2	3	4	5	6	7
0	ADD Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH ES[64]	POP ES[64]
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS[64]	POP SS[64]
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA[64]
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA[64]
4	INC[64] general register / REX[64] Prefixes eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH[64] general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA[64]/ PUSHAD[64]	POPA[64]/ POPAD[64]	BOUND[64] Gv, Ma	ARPL[64] Ew, Gw MOVSXD[64] Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc[64], Jb - Short-displacement jump on condition O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1[1A] Eb, Ib	Ev, Iz	Eb, Ib[64]	Ev, Ib	TEST Eb, Gb	Ev, Gv	XCHG Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	MOVS/B Yb, Xb	MOVS/W/D/Q Yv, Xv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
B	MOV immediate byte into byte register AL/R8B, Ib	CL/R9B, Ib	DL/R10B, Ib	BL/R11B, Ib	AH/R12B, Ib	CH/R13B, Ib	DH/R14B, Ib	BH/R15B, Ib
C	Shift Grp 2[1A] Eb, Ib	Ev, Ib	near RET[64] Iw	near RET[64]	LES[64] Gz, Mp VEX+2byte	LDS[64] Gz, Mp VEX+1byte	Grp 11[1A] - MOV Eb, Ib	Ev, Iz
D	Shift Grp 2[1A] Eb, 1	Ev, 1	Eb, CL	Ev, CL	AAM[64] Ib	AAD[64] Ib		XLAT/ XLATB
E	LOOPNE[64]/ LOOPNZ[64] Jb	LOOPE[64]/ LOOPZ[64] Jb	LOOP[64] Jb	JrCXz[64]/ Jb	IN AL, Ib	eAX, Ib	OUT Ib, AL	Ib, eAX
F	LOCK (Prefix)	INT1	REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3[1A] Eb	Ev

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) / r16(/r) / r32(/r) / mm(/r) / xmm(/r) / (In decimal) /digit (Opcode) / (In binary) REG =			AL / AX / EAX / MM0 / XMM0 / 0 / 000	CL / CX / ECX / MM1 / XMM1 / 1 / 001	DL / DX / EDX / MM2 / XMM2 / 2 / 010	BL / BX / EBX / MM3 / XMM3 / 3 / 011	AH / SP / ESP / MM4 / XMM4 / 4 / 100	CH / BP / EBP / MM5 / XMM5 / 5 / 101	DH / SI / ESI / MM6 / XMM6 / 6 / 110	BH / DI / EDI / MM7 / XMM7 / 7 / 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

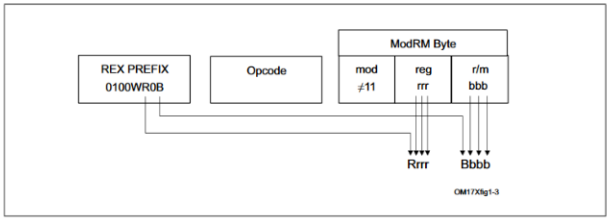


Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used

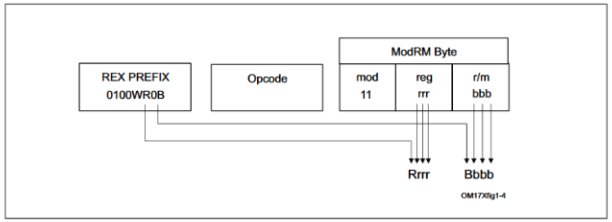


Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used

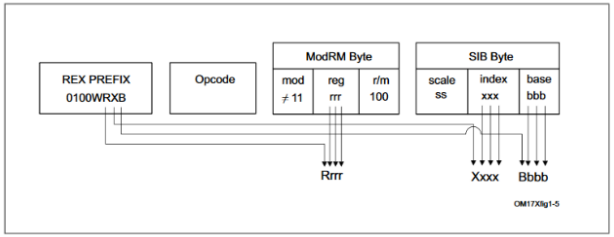


Figure 2-6. Memory Addressing With a SIB Byte

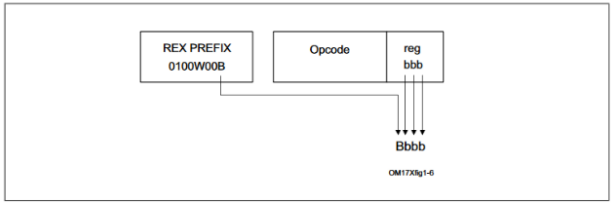


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

32/64-bit	B.R/M													
Mod	0.000 AX	0.001 CX	0.010 DX	0.011 BX	0.100 SP	0.101 BP	0.110 SI	0.111 DI	1.000 R8	1.001 R9	1.010 R10	1.011 R11	1.100 R12	1.101 R13	1.110 R14	1.111 R15
00	[r/m]				[SIB]	[RIP/EIP[1,2] + disp32]	[r/m]						[SIB]	[RIP/EIP[1,2] + disp32]	[r/m]	
01	[r/m + disp8]				[SIB + disp8]	[r/m + disp8]							[SIB + disp8]	[r/m + disp8]		
10	[r/m + disp32]				[SIB + disp32]	[r/m + disp32]							[SIB + disp32]	[r/m + disp32]		
11	r/m															

		B.Base														
Mod	X.Index	0.000 AX	0.001 CX	0.010 DX	0.011 BX	0.100 SP	0.101[1] BP	0.110 SI	0.111 DI	1.000 R8	1.001 R9	1.010 R10	1.011 R11	1.100 R12	1.101[1] R13	1.110 R14	1.111 R15
00	0.000 AX	[base + (index * s)]					[(index * s) + disp32]	[base + (index * s)]							[(index * s) + disp32]	[base + (index * s)]
	0.001 CX															
	0.010 DX															
	0.011 BX															
	0.100[2] SP	[base]					[disp32]	[base]							[disp32]	[base]
	0.101 BP	[base + (index * s)]					[(index * s) + disp32]	[base + (index * s)]							[(index * s) + disp32]	[base + (index * s)]
	0.110 SI															
	0.111 DI															
	1.000 R8															
	1.001 R9															
	1.010 R10															
	1.011 R11															
	1.100 R12															
	1.101 R13															
	1.110 R14															
	1.111 R15															

Ew => 16 bit, Eb => 8 bit, Ev => either 16 or 32 bits (Effective Address)

Intel Syntax: Destination <- Source (dont forget the table uses this)

```
mov    eax,1
instr    dest,source
mov    eax,[ebx + 8]
```

AT&T Syntax: Source -> Destination

```
movl   $10,%eax # note that const
prefixed by $, in hex $0xa
instr    source,dest
movl   8(%ebx),%eax
D(Rb, Ri, S) => Mem[Rb + S * Ri]
```

Example: 01 d8
01 => opcode ADD Ev, Gv
d8 => Ev = EAX, Gv = EBX
ADD eax, ebx, in AT&T => ADD %ebx, %eax

Example: 4e 8d 74 67 10
REX = 4e, W = 1, R = 1, X = 1, B = 0
8d => LEA Gv, M
74 => Gv => r14
67 => scale 2, r12, base rdi
10 => immediate
hence LEA r14, 0x10(rdi, r12, 2) or in AT&T, LEA 0x10(%rdi, %r12, 2), %r14

C to Object File Overview

Some important instructions in x86 (data can only be 1, 2 or 4 bytes in IA 32)

1. move (or movb, movw, movl, movq) from 1, 2, 4, 8 bytes
2. add, sub, imul, shl, shr, xor, and, or, sal, sar (right arithmethic shift maintains the sign or the first bit, but not for left)
3. one operand: inc, dec, neg, not
4. test => like & but only used to set the flags

4 bit registers:

1. CF: Carry Flag
2. SF: Sign Flag
3. ZF: Zero Flag
4. OF: Overflow Flag

cmp a, b => b - a, CF => set to 1 for MSB has a carry

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

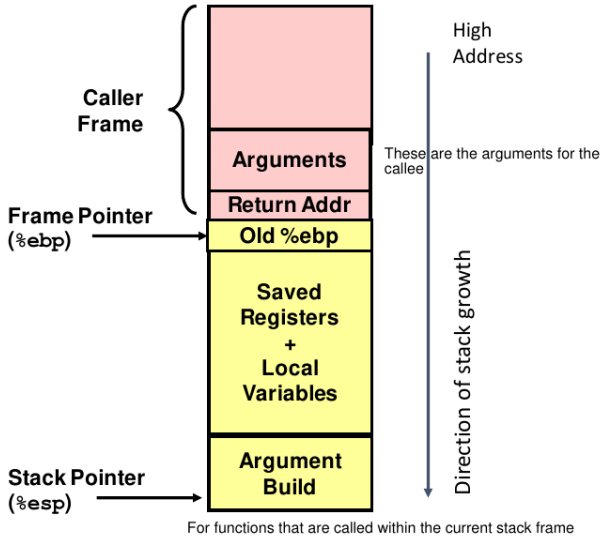
Typical function call:
call fn …
fn:
pushl %ebp
movl %esp,%ebp
…
mov %ebp,%esp (these 2 instructions are just leave)
pop %ebp
ret (return to call + 4)

IA32 Stack

1. Grows toward lower addresses

pop dest => pop and write to dest

IA32/Linux Stack Frame



Caller saved registers: EAX, ECX, EDX
Callee saved: EBX, EDI, ESI
return value: EAX
EBP and ESP are special registers
Arguments therefore retrieved by X(%ebp) => going up

64-bit Factsheet

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–r15	callee-saved registers	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%mmx0–%mmx7	temporary registers	No
%st0,%st1	temporary registers; used to return long double arguments	No
%st2–%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes

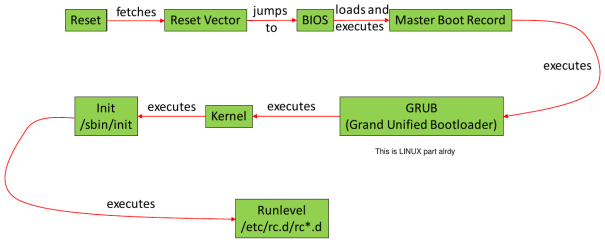
0 0 0 1 0 1 0 0 ⟶ Binary number

1 1 1 0 1 0 1 1 ⟶ One's complement

1 1 1 0 1 0 1 1
+ 1
1 1 1 0 1 1 0 0 ⟶ 2s complement

Boot Process

Overview



1. Reset Vector

The default value of the PC immediately after reset
• 80386 and later x86 processors – 0xFFFFFFF0 (16 bytes below 4 GB).
CS selector = 0xF000, CS base = 0xFFFF0000, IP = 0xFFF0h
=> for multiprocessor, choose 1 bootstrap processor

Entering Real Mode
• In real mode, memory is limited to 1MB (cos segment register in 16 bit wide)
• Upon power up, top 12 address lines asserted high, forcing address to 0xFFFxxxxx (addressing is only 20 bit)what => go to reset vector, goto to load ROM BIOS

Unreal Mode:
• The main idea is just to switch from real to protected mode, change the setting to 32 bit limit, and go back to real mode. The limit remains and now the offset can go to 32 bits instead of 16 bit.

2. BIOS setup

Control Memory Type Range Registers. In the prememory before the memory controller is set-up, use cache as memory, use no evict mode since you cannot evict the content and search in memory

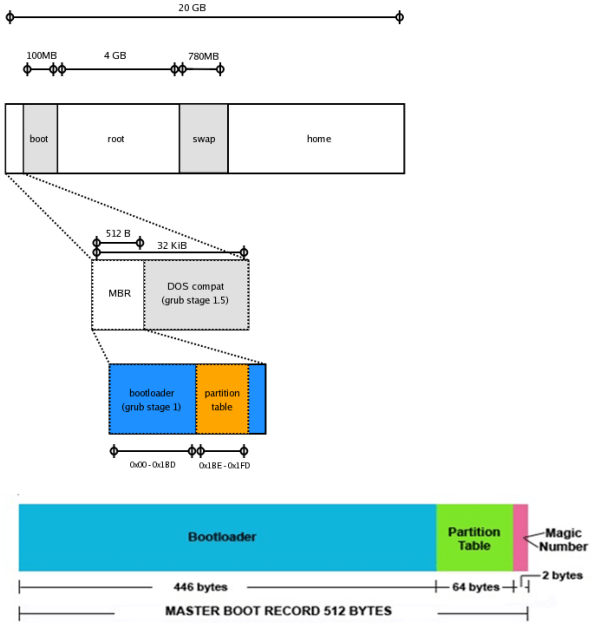
3. Loading MBR (Still real mode)

After setting up the memory controller => get bootloader, scan boot sequence to find master boot record (MBR)

But realize we have partitioning, maybe separate OS, protection, etc.

4. Loading GRUB

bootloader, locate => MBR => load boot.img => load diskboot.img at 0x2000 and loads the rest of GRUB



goes to grub_main() => stage 2 (transfer to protected mode to load it since > 1MB) and eventually need to go back to real mode

5. Load the (Linux) Kernel

vmlinux - a statically linked executable file that contains the entire kernel
vmlinuz – compressed vmlinux

Transfer to _start at arch/x86/boot/header.S => go to boot/main.c => go to unprotected mode for real => decompress => inside kernel already

6. Load init and start pid = 1

call start_kernel, do tons of initialization, mount initrd/initramfs as root => run_init_process, create init, pid = 1

systemd is alternative to init, everything is unit
Unit has a

1. Type (such as service, device, scope, socket, etc)
2. State
3. Optionally a Status

each unit mentions its dependencies and ordering (before/after)

UEFI (Unified Extensible Firmware Interface)

1. Alternative to BIOS

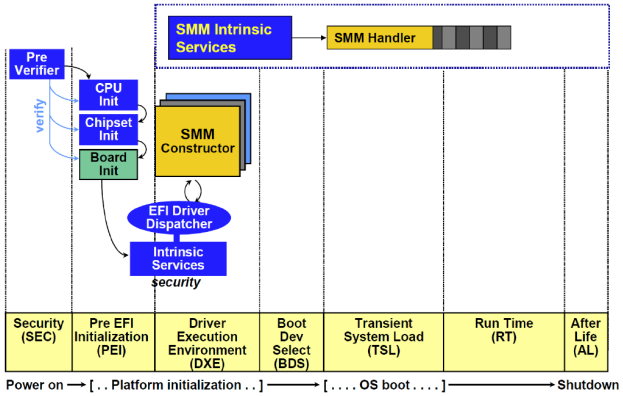
Alternative addressing from cylinder/head/sector to logical block addressing

$$LBA = (C \times HPC + H) \times SPT + (S - 1)$$

- *C*, *H* and *S* are the cylinder number, the head number, and the sector number
- *LBA* is the logical block address
- *HPC* is the maximum number of heads per cylinder (reported by disk drive)
- *SPT* is the maximum number of sectors per track (reported by disk drive)

$$C = LBA \div (HPC \times SPT)$$
$$H = (LBA \div SPT) \bmod HPC$$
$$S = (LBA \bmod SPT) + 1$$

Use new GUID table, globally unique identifier partition table (GPT), LBA 0 = MBR, with header including info such as current LBA, backup LBA, etc.



SEC: go to reset vector, verity integrity of code, set up cache-as-ram, etc

PEI: Init hardware

DXE: setup all the necessary drivers so that the actual OS's bootloader has most of what it needs to work with, load GRUB 2 or other fs.

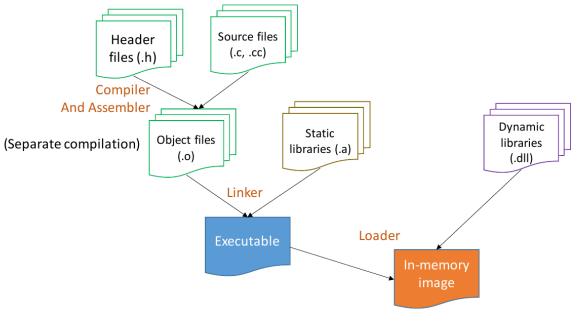
Note: secure boot, basically only boot those with certified boot code.

SMM => system management mode, UEFI must be loaded using FAT32 system in DXE step

Features	init	systemd
Quota Management	No	Yes
Automatic Service Dependency Handling	No	Yes
Kills users Process at logout	No	Yes
Swap Management	No	Yes
SELinux integration	No	Yes
Support for Encrypted HDD	No	Yes
Static kernel module loading	No	Yes
GUI	No	Yes
List all the child processes	No	Yes
Interactive booting	No	Yes
Portable to non x86	Yes	No
Adopted on	Several Distro	Several Distro
Parallel service startup	No	Yes
Resource limit per service	No	Yes
Easy extensible startup script	Yes	No
Separate Code and Configuration File	Yes	No
Automatic dependency calculation	No	Yes
Size	560 KB	N/A
Number of Files	75 files	900 files + glib + DBus
Lines of code – LOC	≈15,000	≈224,000

Comparison of init vs systemd, also systemd can start process in parallel

Linking and Loading



In executable files:

1. Linking: Combining a set of programs, including static library routines, to create a **loadable** image

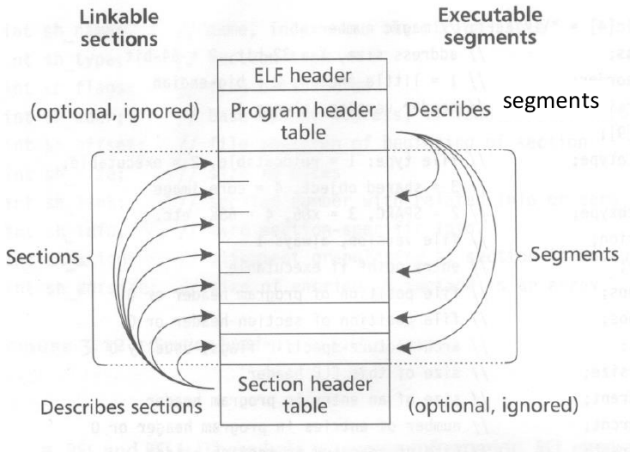
a) Resolving symbols defined within the set
b) Listing symbols needing to be resolved by loader

2. Loading: Starting from the loadable file, copy in dynamic libraries and construct the memory image in a new process

ELF (Executable and Linking File Format)

1. can be Relocatable (object files) => A collection of sections
2. Executable (just need to be loaded)
3. Core file
4. Shared Object (Shared library)

Structure:



A single segment usually consist of several sections. E.g., a loadable read-only segment could contain sections for executable code, read-only data, and symbols for the dynamic linker.

ELF magic number: 4-byte magic number is "0x7F" followed by the string "ELF".
Every symbol's address is defined relative to a section.

Types of sections (sh_type)
• PROGBITS: This holds program contents including code, data, and debugger information.
• NOBITS: Like PROGBITS. However, it occupies no space.
• SYMTAB and DYNSYM: These hold symbol table.
• STRTAB: This is a string table.
• REL and RELA: These hold relocation information.
• DYNAMIC and HASH: This holds information related to dynamic linking.

Types of flags (sh_flags)
• WRITE: This section contains data that is writable during process execution.
• ALLOC: This section occupies memory during process execution.
• EXECINSTR: This section contains executable machine instructions.

Various type of sections:

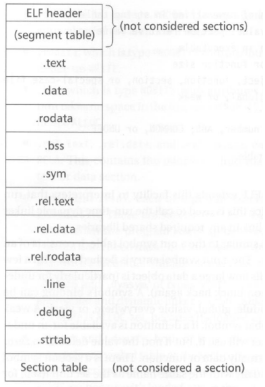
- `.text`:
  - This section holds executable instructions of a program.
  - Type: PROGBITS
  - Flags: ALLOC + EXECINSTR
- `.data`:
  - This section holds initialized data that contributes to the program's image.
  - Type: PROGBITS
  - Flags: ALLOC + WRITE
- `.rodata`:
  - This section holds read-only data.
  - Type: PROGBITS
  - Flags: ALLOC
- `.bss` :
  - This section holds data with no initial values. The system will initialize the data to zero when the program begins to run.
  - Type: NOBITS Doesn't occupy any "real" bits
  - Flags: ALLOC + WRITE
- `.rel.text`, `.rel.data`, and `.rel.rodata`:
  - These contain the relocation information for the corresponding text or data sections.
  - Type: REL
  - Flags: ALLOC is turned on if the file has a loadable segment that includes relocation.
- `.symtab`:
  - This section hold a symbol table.
- `.strtab`:
  - This section holds strings.
- `.init`:
  - This section holds executable instructions that contribute to the process initialization code.
  - Type: PROGBITS
  - Flags: ALLOC + EXECINSTR
- `.fini`:
  - This section hold executable instructions that contribute to the process termination code.
  - Type: PROGBITS
  - Flags: ALLOC + EXECINSTR
- Programming language specific.
  - C does not need these two sections. However, C++ needs them.

- `.interp`:
  - This section holds the pathname of a program interpreter.
  - Type: ALLOC
  - Flags: PROGBITS
  - If this section is present, rather than running the program directly, the system runs the interpreter and passes it the ELF file as an argument.
  - This facility runs non-text programs.
  - In practice, this is used to run the run-time dynamic linker to load the program and to link in any required shared libraries.
    - Variants of "`ld.so`" – the loader.

- `.debug`: used by gdb
  - This section holds symbolic debugging information.
  - Type: PROGBIT
- `.line`:
  - This section holds line number information for symbolic debugging, which describes the correspondence between the program source and the machine code (ever used gdb?)
  - Type: PROGBIT
- `.comment`
  - This section may store extra information.

- `.got`:
  - This section holds the global offset table.
    - Crucial for shared library.
  - Type: PROGBIT
- `.plt`:
  - This section holds the procedure linkage table.
  - Type: PROGBIT
- `.note`:
  - This section contains some extra information.

Overall, it looks like:



Names and Symbols in ELF

String table: just array of null-terminated character sequences

Symbol table: static link time symbolic definition and references, the section index is relative to .text

Dynamic symbol table: for sharable libraries and dynamic executables

Symbol lookup:

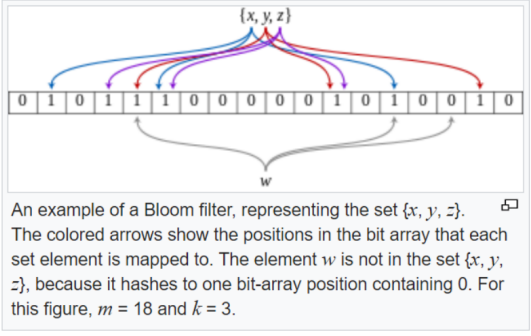
1. If you know the index, go to symbol table and retrieve the index
2. If you know the name, use the hash algo and check the bloom filter mechanism

These hash functions are defined in .gnu.hash sections, containing 4 parts:

- Header: four 32-bit words
  - **`nbuckets`**: number of hash buckets
  - **`symndx`**: number of symbols of the dynamic symbol table that has been hashed To be correct, this is the first index in the .dynsym that is hashed
    - .dynsym may still contain other symbols not hashed
  - **`maskwords`**: number of words in the Bloom filter section
  - **`shift2`** – a shift count used in the Bloom filter
- Bloom Filter
- Hash Buckets
- Hash Values

Bloom Filter: probabilistic to check if an item is a member of a test, can be false positive but no false negative

e.g.



An example of a Bloom filter, representing the set $\{x, y, z\}$. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set $\{x, y, z\}$, because it hashes to one bit-array position containing 0. For this figure, $m = 18$ and $k = 3$.

GNU Hash

1. use k = 2

- `H1 = dl_new_hash(symbol_name)`
- `H2 = H1 >> shift2`

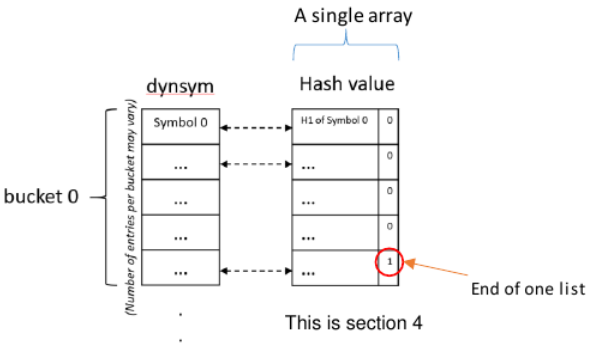
C = size of one mask word in bits

- `N = ((H1 / C) % maskwords)`

- `BITMASK = (1 << (H1 % C)) | (1 << (H2 % C))`
- For Insertion: `bloom[N] |= BITMASK`
- For Test: `(bloom[N] & BITMASK) == BITMASK`

Then the algorithm will check if positive,
• Dynamic symbol table organized by hash buckets
• Compute n – index into the hash bucket
Scan the hash chain until it stops (to avoid name clash, use name mangling)
(dl_new_hash(symname) % nbuckets) == n => get first item in dynsym using **dynsym[buckets[n]]**

Hash values:
One entry for every (hashed) symbol of .dynsym.
• The top 31 bits of each entry contains the top 31 bits of the corresponding symbol's hash value.
• The least significant bit is used as a stopper bit i.e. = 1 when a symbol is the last symbol in a given hash chain.



Relocation

Type of relocation info

1. offset
2. info => Upper half – symbol table index + Lower half – relocation type (the address computation required)
3. addend

```
Relocation section '.rela.text' at offset 0x5d0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000006  000900000002 R_X86_64_PC32     0000000000000000 example_of_global_var - 4
00000000000d  000500000000a R_X86_64_32      0000000000000000 .rodata + 0
000000000017  000b00000002 R_X86_64_PC32     0000000000000000 printf - 4
```

ELF gives relocation so the linker and loader can fix it

Name	Value	Field	Calculation
R_X86_64_NONE	0	none	none
R_X86_64_64	1	*word64*	S + A
R_X86_64_PC32	2	*word32*	S + A - P
R_X86_64_GOT32	3	*word32*	G + A
R_X86_64_PLT32	4	*word32*	L + A - P
R_X86_64_COPY	5	none	none
R_X86_64_GLOB_DAT	6	*wordclass*	S
R_X86_64_JUMP_SLOT	7	*wordclass*	S
R_X86_64_RELATIVE	8	*wordclass*	B + A
R_X86_64_GOTPCREL	9	*word32*	G + GOT + A - P
R_X86_64_32	10	*word32*	S + A
R_X86_64_32S	11	*word32*	S + A
R_X86_64_16	12	*word16*	S + A
R_X86_64_PC16	13	*word16*	S + A - P
R_X86_64_8	14	*word8*	S + A
R_X86_64_PC8	15	*word8*	S + A - P
R_X86_64_DTPMOD64	16	*word64*	
R_X86_64_DTPOFF64	17	*word64*	
R_X86_64_TPOFF64	18	*word64*	
R_X86_64_TLSGD	19	*word32*	
R_X86_64_TLSLD	20	*word32*	
R_X86_64_DTPOFF32	21	*word32*	
R_X86_64_GOTTPOFF	22	*word32*	
R_X86_64_TPOFF32	23	*word32*	
R_X86_64_PC64 †	24	*word64*	S + A - P
R_X86_64_GOTOFF64 †	25	*word64*	S + A - GOT
R_X86_64_GOTPC32	26	*word32*	GOT + A - P
R_X86_64_SIZE32	32	*word32*	Z + A
R_X86_64_SIZE64 †	33	*word64*	Z + A
R_X86_64_GOTPC32_TLSDESC	34	*word32*	
R_X86_64_TLSDESC_CALL	35	none	
R_X86_64_TLSDESC	36	*word64*×2	
R_X86_64_IRELATIVE	37	*wordclass*	indirect (B + A)
R_X86_64_RELATIVE64 ††	38	*word64*	B + A
Deprecated	39		
Deprecated	40		
R_X86_64_GOTPCRELX	41	*word32*	G + GOT + A - P
R_X86_64_REX_GOTPCRELX	42	*word32*	G + GOT + A - P

† This relocation is used only for LP64.
†† This relocation only appears in ILP32 executable files or shared objects.

S: Symbol Name address, A: Addend, Offset: Location from main / the code, if PLT, use PLT entry

e.g.

```
Relocation section '.rela.text' at offset 0x270 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000a  000b00000002 R_X86_64_PC32     0000000000000000 ext_var - 4
000000000013  000500000002 R_X86_64_PC32     0000000000000000 .rodata - 4
000000000018  000d00000004 R_X86_64_PLT32    0000000000000000 foo - 4
```

Then first relocation is address of ext_var - 0x4 - (code section beginning + 0xa)

Executable File Segments

- PT_LOAD: This segment is a loadable segment.
- PT_DYNAMIC: This array element specifies dynamic linking information.
- PT_INTERP: This element specified the location and size of a null-terminated path name to invoke as an interpreter.

Loadable: readonly or read/write
(There are more)

Linker: combining multiple ELF file with multiple segments into 1 executable with text, data, etc segments

Dynamic Linking

1. one common code to be used by everyone

Linux static library => bunch of object files concatenated and statically compiled

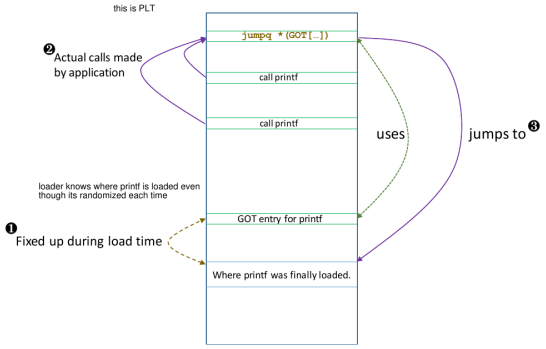
How it works overview:

- Compiler ensures every call to the same dynamically linked routine will go to the same PLT entry
  - PLT[0] is reserved
- Each (64) bit PLT entry consists of:
  - `jmpq` ***GOT**[entry corresponding to the function] (take the content and jump there)
    - Called a "trampoline".
  - `pushq` <GOT entry number of corresponding function>
    - Starts from 0, i.e. GOT[3].
  - `jmp` PLT[0]
- **GOT**[entry corresponding to the function] = address of "`pushq`" *initially* (basically just jump to next line)

So either go to pushq and go to PLT[0] to fix the dynamic library and all the entries or jump to the GOT immediately if loaded

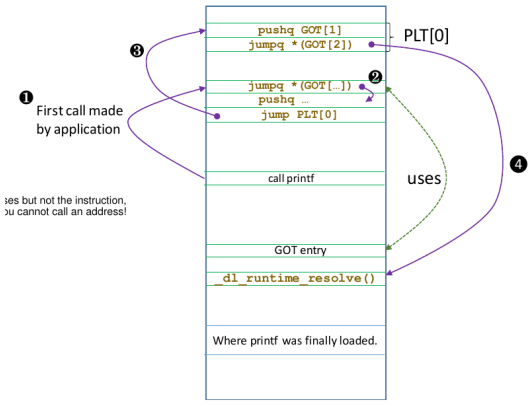
Eager Binding (immediately load all the libraries when the executable is loaded)

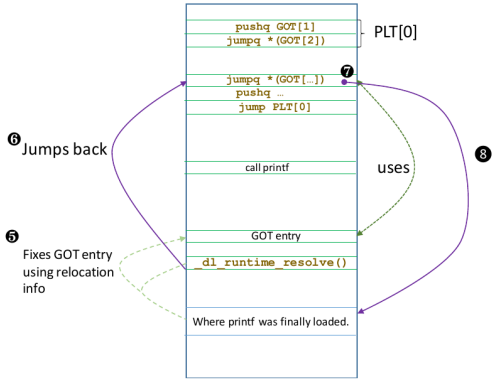
“BIND NOW”



Lazy Binding

Lazy = not immediately change GOT entry at load time





GOT[0] => Address of .dynamic, GOT[1] => Pointer to a linked list of dynamic, GOT[2] => function to fix dynamic library

for position independent code due to address space layout randomization (aslr), it will change all address to relative addressing

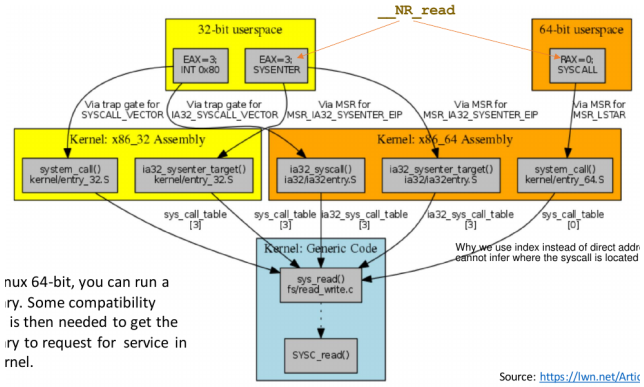
loading ELF is done during exec syscall, if there is PT_INTERP => dynamic linking required, will create image for the loader and load the library.

Syscall

To implement a new syscall => define a new symbol and add them to to this table arch/x86/entry/syscalls/syscall_{32|64}.tbl

```c
#include <linux/kernel.h>

asmlinkage long sys_hello(void)
{
        printk("Hello world\n");
        return 0;
}
```

x86 syscalls:
old: use a software interrupt, int 0x80 => int is interrupt instruction
new: use SYSENTER/SYSEXIT, in 64 bits AMD, use SYSCALL/SYSRET



- Registers on kernel entry via a SYSCALL:
  - `%rax` - system call number
  - `%rcx` - return address
  - `%r11` - saved rflags
  - `%rdi` - arg0
  - `%rsi` - arg1
  - `%rdx` - arg2
  - `%r10` - arg3
  - `%r8` - arg4
  - `%r9` - arg5

During boot time, the program will hardwire the address of syscall entry to Machine Specific Register, IA32_LSTAR to be specific

MSRs are 64 bits long and identified by a 32-bit integer, existence can be confirmed using cpuid instruction, use RSMSR and WRMST which are privileged instructions (only by kernel)

There are only two ways to enter the kernel
• Via interrupt
• Using SYSCALL/SYSENTER
• Important difference: a system call always transits from user to kernel mode, while an interrupt/exception/trap can happen even inside Ring 0 (You don't need to call syscall in kernel mode, you can just call the function directly)

Stacks used by the kernel

1. Per thread kernel stack => 16 KB, syscall should be handled here since each syscall is called by a thread
2. Entry trampoline stack
3. Interrupt stack
4. hard irq stack (for interrupts)
5. soft irq stack

Task State Segment => structure on x86-based computers that must be defined as part of the hardware specs, used to support hardware task switching but not used in 64 bit, used as pointers to stacks instead

1. consists of 7 Interrupt stack table pointer
2. one stack for each protection ring, RSP0 => entry trampoline stack (to clean up all the syscall values on the per thread stack, a small piece of code which is constructed on the fly on the stack when the address of a nested function is taken)
3. RSP2 => Scratch to contain user stack pointer on SYSCALL

Linux only has one TSS for each CPU and uses them for all tasks
In Linux:
FS => thread local storage
GS => per CPU data structure
The assembly:

```
SYM_CODE_START(entry_SYSCALL_64)
        UNWIND_HINT_EMPTY

        swapgs
        /* tss.sp2 is scratch space. */
        movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
        SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
        movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)

        /* 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 */
        pushq   $__USER_CS                      /* pt_regs->cs */
        pushq   %rcx                            /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
        pushq   %rax                            /* pt_regs->orig_ax */

        PUSH_AND_CLEAR_REGS rax=$-ENOSYS

        /* IRQs are off. */
        movq    %rax, %rdi
        movq    %rsp, %rsi
        call    do_syscall_64           /* returns with IRQs disabled */
```

1. GS segment register is used for quick access to the per CPU region so as to get the per-CPU variables quickly .
2. saves the user mode stack pointer into a scratch location, namely the sp2 of the TSS which is also not used since Ring 2 is not used (tss_sp2)
3. Restore the full kernel page table Use %rsp as a scratch since it was already saved up (kernel_cr3)
4. Switch to true kernel stack: the per-thread kernel stack.
5. Saves registers to pt_regs – which is created on top of the stack.
6. call service dispatcher

Returning

```
        /*
         * We win! This label is here just for ease of understanding
         * perf profiles. Nothing jumps here.
         */
syscall_return_via_sysret:
        IBRS_EXIT
        POP_REGS pop_rdi=0

        /*
         * Now all regs are restored except RSP and RDI.
         * Save old stack pointer and switch to trampoline stack.
         */
        movq    %rsp, %rdi
        movq    PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp
        UNWIND_HINT_EMPTY

        pushq   RSP-RDI(%rdi)   /* RSP */
        pushq   (%rdi)          /* RDI */

        /*
         * We are on the trampoline stack.  All regs except RDI are live.
         * We can do future final exit work right here.
         */
        STACKLEAK_ERASE_NOCLOBBER

        SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi

        popq    %rdi
        popq    %rsp
SYM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SYM_L_GLOBAL)
        ANNOTATE_NOENDBR
        swapgs
        sysretq
```

1. Switch to trampoline stack
2. So that we can call a procedure (stackleak_erase()) to clean up per-thread stack.

vsyscall

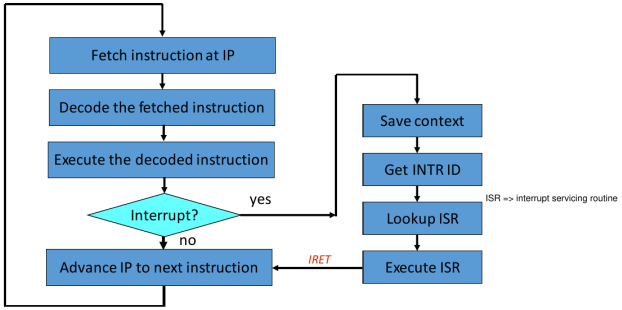
1. Faster syscall: do not actually enter the kernel
2. Linux kernel maps a page containing some kernel variables and implementation of some system calls into the user space (Key: read only)

Used for functions such as getcpu(), gettimeofday(), time(), etc. The page containing vsyscall is readable, executable but not writable.

virtual dynamic shared object => vsyscall but allows ASLR

Interrupts

Overview

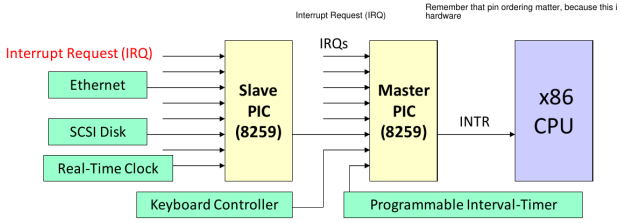


Types based on Intel:

1. Async (I/O, etc)
2. Sync = Exceptions
    a. Programmed Exceptions (syscalls, used by int and int3) => tbh almost the same like trap, but requested by users.
    b. Processor detected => Faults (instruction is retried, like page faults), Traps (instr not retried, deliberately set, usually used to transfer the control flow to debugger, requested by hardware), Aborts (major error)

Most error => translate to signal => kernel's job is just to send the signal to the current process

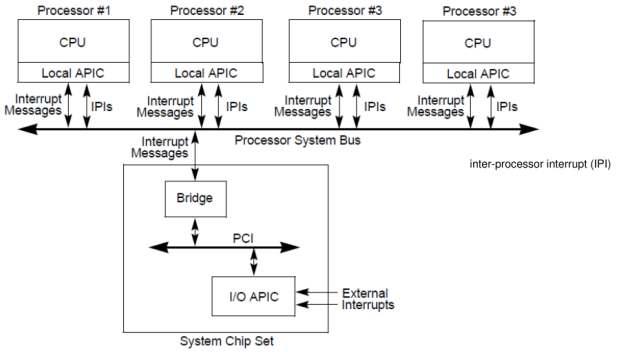
Intel 8259 Porgrammable Interrupt Controller



Initialize:

1. IR0 - IR7
2. send INT signal to INTR pin => processor sends 3 INTA signals, send x86 CALL opcode, low byte and high byte of call address (then you can CALL using 16 bit address)

APIC: Advanced PIC to handle multiprocessor system



Basically devices connect to I/O APIC (or directly to Local APIC), distribute over bus to Local APIC => a programmable local vector table is looked up to translate these local interrupts sources to system-wide interrupt vectors

Typical Interrupt

Vector	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20	#VE	Virtualization Exception	EPT violations[6]
21-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

IRQ 0 = Vector 32

Masked => ignore maskable external interrupts

Non-maskable interrupt (NMI) => an external hardware interrupt that cannot be ignored

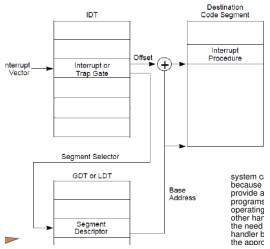
• The interrupt vector is an index (0-255) into the interrupt descriptor table
• Vectors usually IRQ# + 32
• Below 32 reserved for non-maskable interrupt and exceptions
• Maskable interrupts can be assigned as needed
• Vector 128 used for Linux syscall

x86 handling interrupts

system calls are faster than interrupts because they have a smaller overhead and provide a direct path for user-mode programs to request services from the operating system's kernel. Interrupts, on the other hand, have a higher overhead due to the need to be processed by the interrupt handler before they can be passed on to the appropriate ISR.

The Interrupt Descriptor Table Register (IDTR) points to a table (IDT) consisting of either:
• Task-gate descriptor (still used for 32 bits hardware task switch)
• Interrupt-gate descriptor : Disables further interrupts
• Trap-gate descriptor: Further interrupts still allowed

In Linux, if IST = 0, use Ring 0 stack, otherwise use interrupt stack table for Double Fault, NMI, Debug Interrupt, Machine Check Exception. (Use this because you don't want the interrupt to touch the user stack)



For linux, now all the IDT is implemented using interrupt gates, Task gate is only used for hardware task switching in 32 bits, or double fault in 64 bit.

all interrupts use **rsp0 entry stack** (except the special 4 mentioned before) => which will trampoline to **hardirq_stack**

Rules of interupts + exception nestings
• an exception (e.g. page fault, system call) can not preempt an interrupt; if that occurs it is considered a bug
• an interrupt can preempt an exception
• an interrupt can not preempt another interrupt (it used to be possible)

For interrupt: if there is a privilege change => do similar to SYSCALL (swapgs + kernel cr3)

Linux Handling of the Interrupt Routines

1. Top half - immediate handler (critical processing) => similar to exceptions

Do the absolute minimal to handle the interrupt and return as quickly as possible
• IRQ disabled → acknowledge IRQ + send signal to the process→ execute device handler → IRET

Limitations:
1. cannot sleep or do something to sleep
2. cannot down() semaphore
3. cannot refer to current
4. cannot do kmalloc kernel which can sleep, must be atomic but can fail
5. cannot call schedule / transfer data to /from user space

2. Bottom Half - deferrable functions

• Not time critical work
• IRQ enabled – can be interrupted
• Mechanisms: tasklet, softirq, work queue, kernel thread

1. Soft IRQ (Interrupt Context)

Like hardware IRQ, but done entirely in software. re-entrant code, can be done in multiple processors, can be interrupted.

Handlers are predefined beforehand in softirq_vec[] (timer, network, polling, etc)

Can be scheduled but ksoftirqd, after interrupts, exceptions, syscalls (must do its own locking)

2. Tasklets (Interrupt Context)

Implemented on top of Soft IRQ. It is not reentrant, i.e. taskslets of the same types must run serially.

Tasklets can dynamically registered.

3. Work Queues

Run in the process context, can sleep but cannot access user space. A worker thread grab a work from the queue.

4. Kernel Thread

Always in kernel mode, no user context => Each bottom half has its own context, each interrupt handling code done in separate thread.

Comparsion

	ISR	SoftIRQ	Tasklet	WorkQueue	KThread
Will disable all interrupts?	Briefly	No	No	No	No
Will disable other instances of self?	Yes	Yes	No	No	No
Higher priority than regular scheduled tasks?	Yes	Yes*	Yes*	No	No
Will be run on same processor as ISR?	N/A	Yes	Yes	Yes	Maybe
More than one run can on same CPU?	No	No	No	Yes	Yes
Same one can run on multiple CPUs?	Yes	Yes	No	Yes	Yes
Full context switch?	No	No	No	Yes	Yes
Can sleep? (Has own kernel stack)	No	No	No	Yes	Yes
Can access user space?	No	No	No	No	No

Exceptions use the kernel-level exception stack => cannot sleep otherwise other process might wanna use this stack too, Interrupts use a hard IRQ stack + soft IRQ stack (for 64 bit, soft == hard stack), one per processor

3 types of interrupts in Linux : I/O, timer, interprocessor

I/O interrupts

IRQ sharing if 2 devices are not used at the same time, IRQ assignment is only done at the last possible moment

Problem: lost IRQ, in SMP, other processors can mask out interrupts, so current processor think it can ignore it.

Some solutions: retry the irq, use backup processor to handle all lost interrupts, use interrupt distribution algo, use timer (if a processor doesn't receive interrupt, unmask and get it forcefully)

Signals

```
Signal     Value     Action   Comment
SIGHUP       1       Term     Hangup detected on controlling terminal
                              or death of controlling process
SIGINT       2       Term     Interrupt from keyboard
SIGQUIT      3       Core     Quit from keyboard
SIGILL       4       Core     Illegal Instruction
SIGABRT      6       Core     Abort signal from abort(3)
SIGFPE       8       Core     Floating point exception
SIGKILL      9       Term     Kill signal
SIGSEGV     11       Core     Invalid memory reference
SIGPIPE     13       Term     Broken pipe: write to pipe with no
                              readers
SIGALRM     14       Term     Timer signal from alarm(2)
SIGTERM     15       Term     Termination signal
SIGUSR1   30,10,16   Term     User-defined signal 1
SIGUSR2   31,12,17   Term     User-defined signal 2
SIGCHLD   20,17,18   Ign      Child stopped or terminated
SIGCONT   19,18,25   Cont     Continue if stopped
SIGSTOP   17,19,23   Stop     Stop process
SIGTSTP   18,20,24   Stop     Stop typed at terminal
SIGTTIN   21,21,26   Stop     Terminal input for background process
SIGTTOU   22,22,27   Stop     Terminal output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or
ignored.
```

Signal	Value	Action	Comment
SIGBUS	10,7,10	Core	Bus error (bad memory access)
SIGPOLL		Term	Pollable event (Sys V). Synonym for SIGIO
SIGPROF	27,27,29	Term	Profiling timer expired
SIGSYS	12,31,12	Core	Bad argument to routine (SVr4)
SIGTRAP	5	Core	Trace/breakpoint trap
SIGURG	16,23,21	Ign	Urgent condition on socket (4.2BSD)
SIGVTALRM	26,26,28	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2BSD)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2BSD)

user-supplied signal handler will execute, otherwise default action is taken (which is just core dump + termination)

```
void sig_handler(int signo)
{
  if (signo == SIGINT)
    printf("received SIGINT\n");
}

int main(void)
{
  if (signal(SIGINT, sig_handler) == SIG_ERR)
```

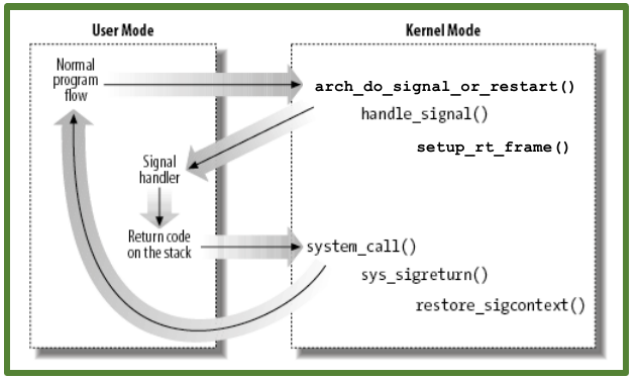
Signal handling

1. Signal generation

Kernel send signal to user processes, update relevant processes data structures => basically in task_struct, there is a signal_struct that shows pending signal list

2. Signal delivery

Before interrupt routine returns to user mode, check pending singal, then handle the signal => then eventually do sigreturn() in the user space stack to restore the sigal context of a process



Future:

Flexible REturn and Event Delivery

1. just automatic swapgs
2. return for ring 0 to ring 0 => ERETS, ERETU => go to ring 3
3. bypass IDT using new context and new MSR

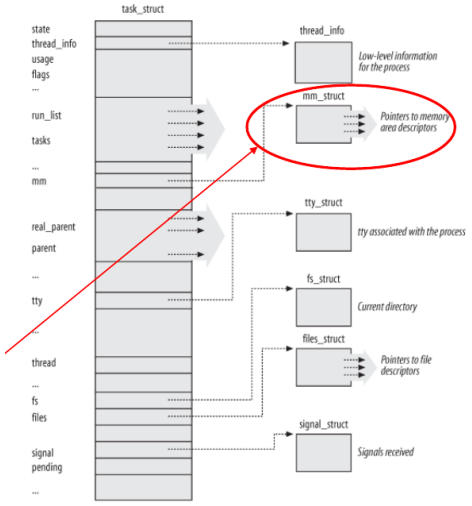
AMD: Reentrant interrupt. Check if an interrupt needs to be protected => if bit = 1 and protected => generate double fault, otherwise multiple interrupts can go together

Processes

process -> multiple threads of controls (can be user/kernel level)

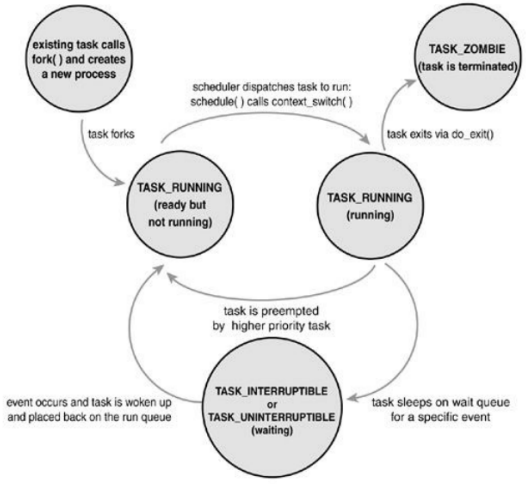
Linux => everything is a task, process is a task with a single thread, a task can have multiple tasks.

Process Control Block = task_struct in linux



thread_info (contains architecture dependent info about the thread) and thread_struct (contains CPU-specific state of this task, more detailed too)

Task States



TASK_TRACED => stopped by debugger

Process in linux forms a tree

Parents of Processes

• Real parent: The process descriptor of the process that created it or to the descriptor of process 1 if the parent process no longer exists.

• Parent: The process that will handle the termination signal of the child. May be different due to tracing.

Process Creation

1. Copy-on-Write (COW): share until you need to write
2. clone(): more flexible than fork(), allows child process to share parts of the execution context like memory space, fds, etc.
3. vfork(): similar to clone but parent will block until the child exits / exec.

important time variables:

1. jiffies: a counter that stores the numberof elapsed ticks since the system starts
2. xtime: current time and date

Done by local and global timer interrupt (usually every 1ms)

Scheduling

Assumption: a program is either I/O bound or CPU bound

Important Metrics:

1. Turnaround time: time completion - time submission
2. Response time: average time elapsed from submission to first response
3. Throughput: jobs / second
4. Fairness: equal CPU / user or process
5. Completion time: time to complete
6. Average completion time = (total waiting time of all processes) + (total execution time of all processes) / number of processes => tbh just total turnaround time / number of processes
7. Waiting time: turnaround time - execution time

Type of scheduling algorithms

1. FCFS: Simple, Short jobs get stuck behind long ones
2. Round Robin: each process gets time quantum, use it until it finishes or surrenders: Better for short jobs, Fair, need to balance context switch overhead and timeslice

a. RR has worse response time compared to FCFS if all the jobs are of same length

3. Shortest Job First (Shortest Time to Completion)
4. Shortest Remaining Time: preemptive version of SJF
    a. SJF/SRTF are the best you can do at minimizing average response time
    b. But can lead to starvation, need to predict future

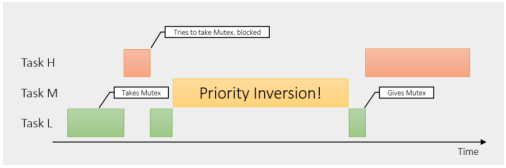
Predict remaining time using

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$

5. Priority: just allocate based on priority, can be preemptive and nonpreemptive
    a. Problem: starvation, solution: aging
    b. SJF is a priority scheduling where priority is the predicted next CPU burst time
    c. In linux, its called niceness. Values it between -20 to 19 (-20 least nice, must be scheduled first)

Priority Inversion



Fix: Process L temporarily granted the high priority of process H: Priority inheritance

6. Multi Level Feedback Scheduling
- Multiple queues, each with different priority
- Higher priority queues often considered “foreground” tasks
- Each queue has its own scheduling algorithm
- Foreground → RR, background → FCFS
- Sometimes multiple RR priorities with quantum increasing exponentially (highest queue: 1ms, next: 2ms, next: 4ms, etc.)

• Adjust each job’s priority as follows
• Job starts in highest priority queue
• If entire CPU time quantum expires, drop one level
• If CPU is yielded during the quantum, push up one level (or to top)

Result approximates SRTF, scheduling between queues can be done in fixed priority or time slice.

Short term vs long term scheduler
The long-term scheduler decides what processes should be put into the ready queue in the first place for the short-term scheduler, so that the short-term scheduler can make fast decisions on a good mix of a subset of ready processes (the rest are held in memory / disk)

Implement fairness: could give each queue some fraction of the cpu, could increase priority of jobs that don’t get service, etc

7. Lottery Scheduling

Give each job some number of lottery tickets, on each time slice, pick randomly.

To avoid starvation, every job gets at least one tiket (so everyone in expectation makes progress)

Pro: easier to tune if load changes

Type of Processes:
1. Normal Processes

Initial scheduler => just search for one task, look for next task, repeat

O(n) scheduler => time is divided up into epoch, each task has a goodness factor, the best task is chosen to run next

O(1) scheduler => divide into timeslice, global priority from 0 - 139, maintain active and expire 140 linked list heads each and check the priority bitmap

Completely Fair Scheduler, with n running tasks, each task would be having 1/n amount of CPU-time (weighted fair queueing)

Uses Time-ordered Red-Black Tree
each node is a job, indexed by virtual time

Balance: the path from the root to the farthest leaf is no more than
twice as long as the path from the root to the nearest leaf

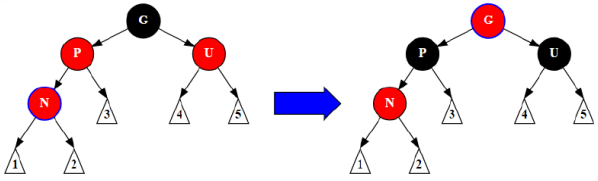
Every red-black tree with n internal nodes has a height of at most 2 × log(n + 1)

Property 1: Every node must be either red or black
Property 2: The root of the tree must be black
Property 3: All leaves ("nodes" containing NIL/NULL) is black (the leaf node is just N value)
Property 4: If a node is red, then both its children are black
Property 5: Every path from a node to a descendant leaf must contain the **same** number of black nodes (black depth)

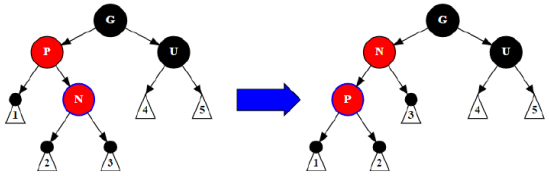
Insertion:

1. Colour the node to be red

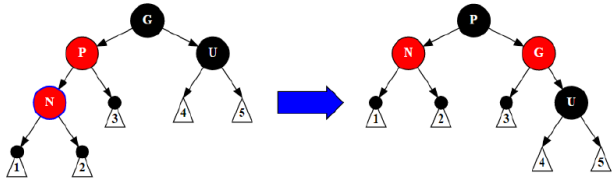
Check each of these cases
Case 1: (if root)
• The node inserted, N, is the root
• Violates Property 2 (root must be black)
• Change colour of N to black
• Otherwise, go to Case 2
Case 2: (child with black parent)
• If the parent of N, i.e., P, is black, then we are done; return; otherwise go to case 3
Case 3: (child with black parent)
• P is red, and U is not null and is red
• Change P and U to black and G to red, and go to Case 1
• Otherwise, go to Case 4



Case 4: (basically avoid zig zag pattern)
• P is red but U is black
• If N is the right child of P, and P is the left child of G then remedy is left rotation on P
• If N is the left child of P, and P is the right child of G then remedy is right rotation on P



(must go to case 5)
Case 5:
• P is red but U is black, by default => null == black
• If N is left child of P, and P is left child of G, then remedy is right
rotation on G
• Otherwise, rotate left on G
• Switch colours of P and G



Deletion is ignored 🙂

Vruntime weight

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

≈ 1024 x (1.25)^-nice

• A period is a length of time in which every task ran at least once.

```
if #runnable tasks > some threshold (default 8)
        period = #runnable task * min_granularity (default 0.75ms)
else
        period = base (default 6ms)
```

• The time slice given to each task is weighted by its weight

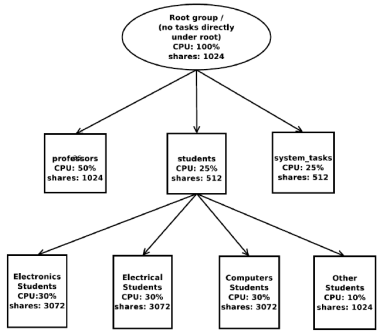
Time slice for a task = period * (weight of task) / (total weight of runqueue)

Note of the min period (if < 6ms, always use period = 6ms)

vruntime += (time process ran) * 1024 / (load weight of this process)

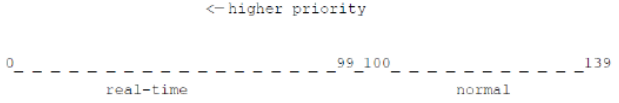
if you are sleeping, compute current vruntime and min_vruntime, after waking up, add the delta to current vrun_time

in more modern group scheduling, you can schedule an entity (consisting of many tasks) => must be within one domain since you don't want the task to be in random CPU scattered around, instead of a task



for smp, tasks can be moved from one rb_tree to another to rebalance it

2. RT process (doesn't have hard realtime processes, use **preemptible FIFO or RR** with static priorities)



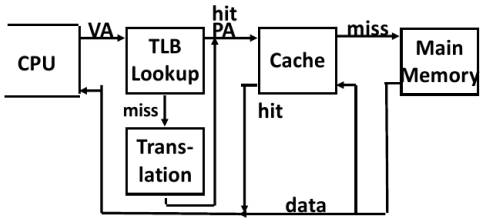
schedule() called every scheduler_tick()
for real time scheduler, its just priority scheduling and can only get replaced if

Memory Management

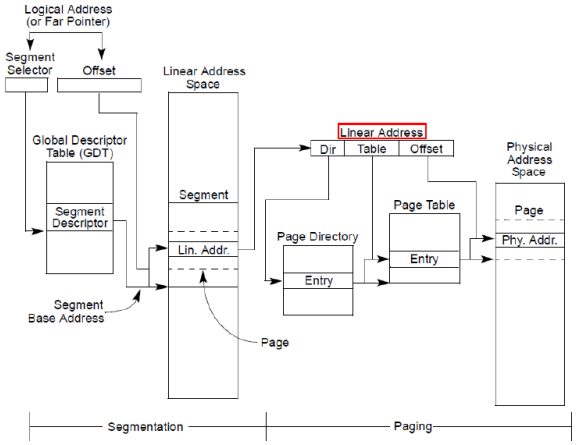
Page Table

1. each process needs its own address space, it has its own page table

TLB: Make translation faster, typically only has 128 to 256 entries. Each entry indicate what is the permitted access, dirty bit, etc.



Memory in Linux



Even though in reality, linux only make every 1 segment (effectively removing segmentation)

Paging

1. Each page is 4 KB, root pointed by CR3

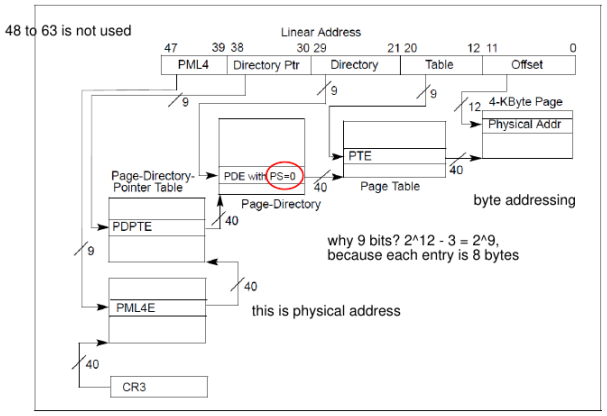


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

For 2 MB paging, only 3 level paging, for 2GB only 2 level paging, for 5 level paging, just use bit 48 - 56 => realize that each physical address is 52 bits long only

- All addresses used here are physical addresses
- PML4E entry address: (this is like concatenation)
  - ✓ Bits 51-12 from CR3
  - ✓ Bits 11-3 from bits 47-39 of linear address
  - ✓ Bits 2-0 are all 0
- PDPTE entry address:
  - ✓ Bits 51-12 from PML4E
  - ✓ Bits 11-3 from bits 38-30 of linear address
  - ✓ Bits 2-0 are all 0
- PDE entry address:
  - ✓ Bits 51-12 from PDPTE
  - ✓ Bits 11-3 from bits 29-21 of linear address
  - ✓ Bits 2-0 are all 0
- PTE entry address:
  - ✓ Bits 51-12 from PDE
  - ✓ Bits 11-3 from bits 20-12 of linear address
  - ✓ Bits 2-0 are all 0
- Final address:
  - ✓ Bits 51-12 from PTE
  - ✓ Bits 11-0 from linear address

In Linux

• Intel PML4 => Linux Page Global Directory (PGD)

• Intel Page Directory Pointer Table => Linux Page Upper Directory (PUD)

• Intel Page Directory => Linux Page Middle Directory (PMD)

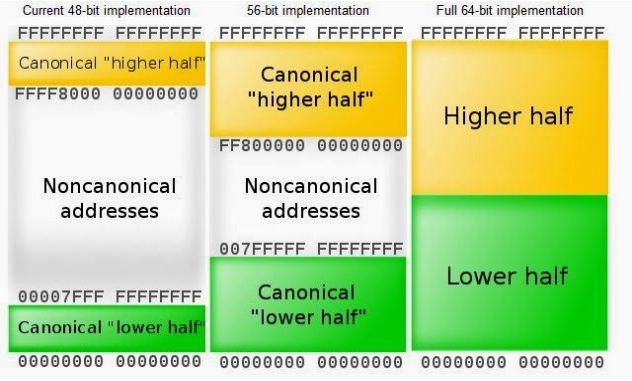
• Intel Page Table => Linux Page Table (PTE)

In paging, they use physical address (to avoid recursive virtual address translation)

First 12 bits in CR3 => Process Context ID (or address space ID), used to distinguish one process' address space from another

Canonical Address

in 4 level paging, only first 48 bits are used, bit 63-48 must be sign extension of bit 47



Roughly, the memory layout looks like the following:

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [47:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
...
ffffffffff600000 - ffffffffff600fff (=4 kB) legacy vsyscall ABI
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

•Some parts of user space of any process maps to parts of physical memory

•ALL of physical memory maps to a fixed region in kernel space

• For 32-bit, can be configured; normally __PAGE_OFFSET is 0xC00000000

• For 64-bit, __PAGE_OFFSET is 0xffff880000000000 => Direct Mapping

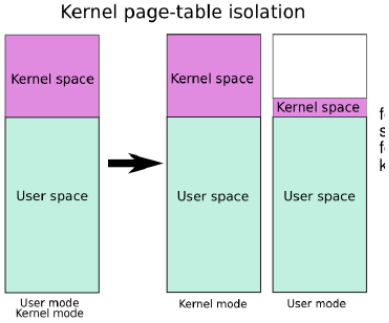
The kernel uses a single set of page tables and a copy of it is available in every process

fixmap => linear mapping to physical address, but doesn't need to follow __PAGE_OFFSET

Kernel is responsible to flush TLB, may not be needed if the same set of page tables are used

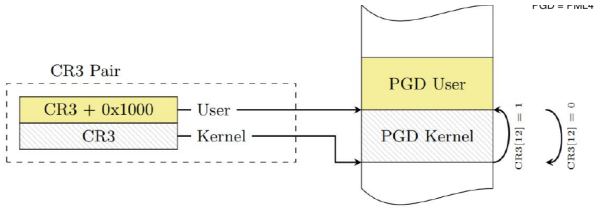
Page Frame => Physical, Virtual => Page, multiple page frames => folio

Page Table Isolation (PTI) => aka KAISER



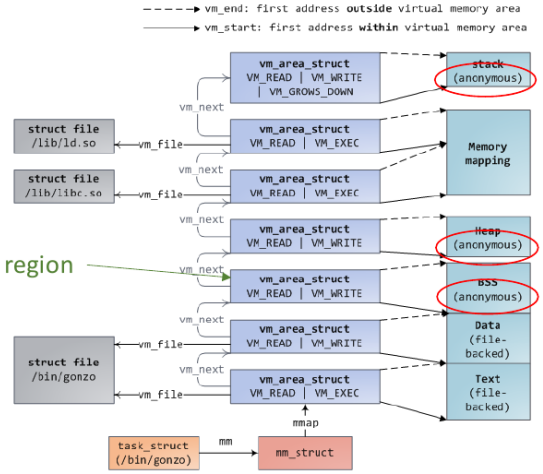
One set of page table for use in kernel mode includes both kernel-space and user-space. The second set of page table for use in user mode contains a copy of user-space and a minimal set of kernel-space handling system calls and interrupts.

why kernel requires full user space page table, for copy_to_user. To make switching from user to kernel faster, CR3 comes in pair

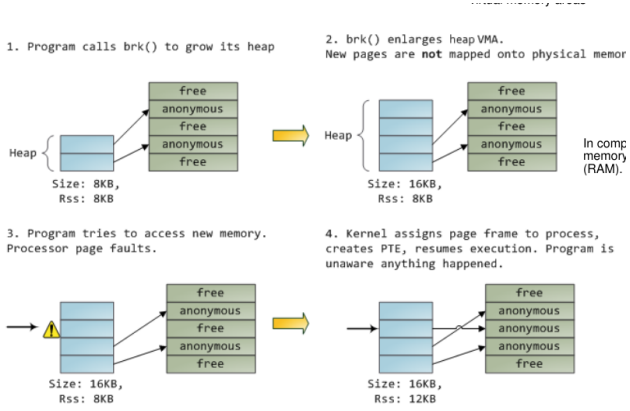


User is for usermode, kernel is for kernel mode

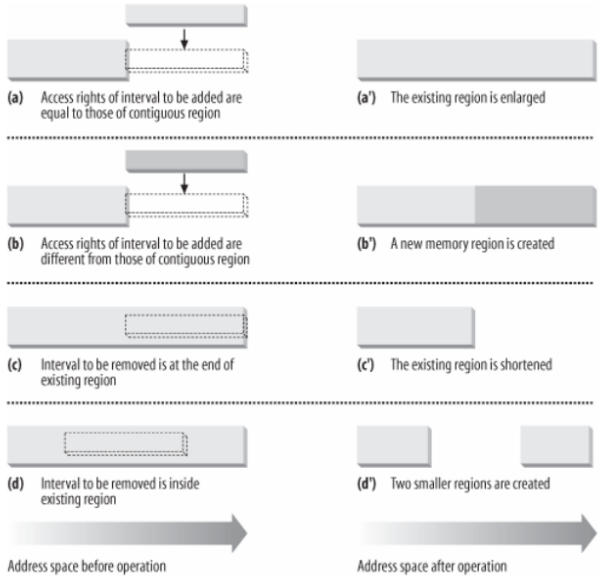
virtual address space are allocated using brk(), looks like the following



Lazy Expansion of BRK



Virtual Memory Area operations



Searching the VMA => searching the linked list linearly

Memory Zone

1. Direct memory access

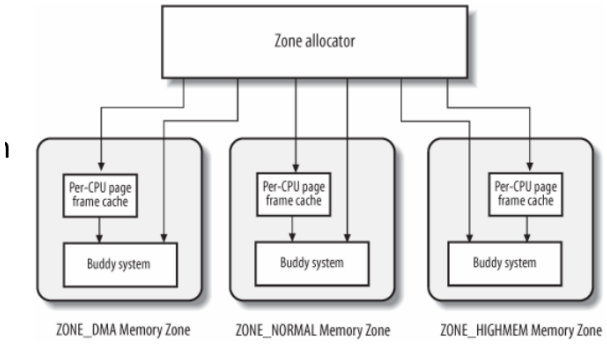
Hardware subsystem direct access to memory independent of CPU

DMA works only with physical addresses

• OS must set it up

2. Normal, Highmem, device, etc

Zone allocator

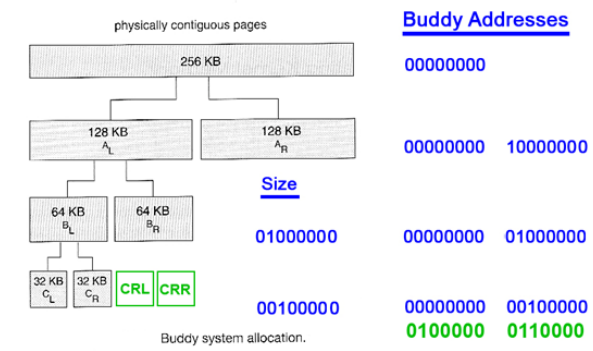


Buddy Allocation System: little external fragmentation, but can suffer from internal

All free page frames are grouped into 11 lists of blocks that contain groups of 1 (2^0), 2 (2^1), 4 (2^2), 8 (2^3), 16 (2^4), 32 (2^5), 64 (2^6), 128 (2^7), 256 (2^8), 512 (2^9), and 1024 (2^{10}) contiguous page frames

allocating memory is done in reverse, if you request 128 blocks, then you search from 128 => 256 => 512, etc

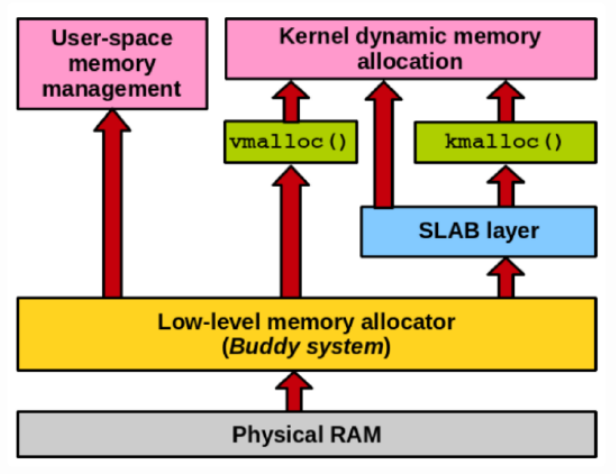
kernel will merge 2 blocks that are buddies (same size)



i.e. the buddy must only differ in bit 2 * b * 2^{12} (size of a page)

hot cache: lots of free pages so kernel can allocate without asking the allocator, good for processes writing a lot to the page frames, cold cache: few free pages, good for DMA

Memory Allocator



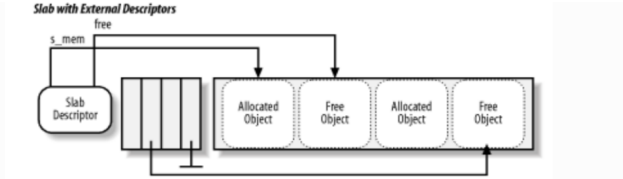
kmalloc => contiguous physical memory, vmalloc => contiguous physical memory

SLOB allocator:
use list of different size objects and use best fit

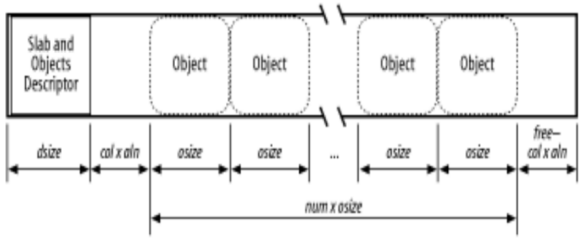
SLAB allocator:
In general, 26 general cache descriptors associated with memory areas of size 32 bytes to 128Kbytes.

To read the data list => first free => go to index 1 => go to pointer 1 => pointing to free block 4, go to pointer 4 (done).

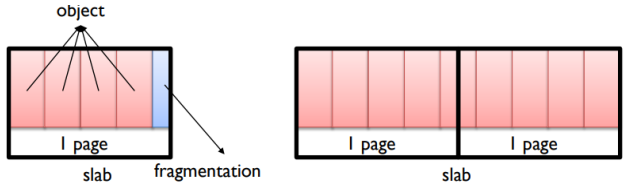
Similarly for allocated object (i.e. its 1 ll for free, 1 ll for allocated)



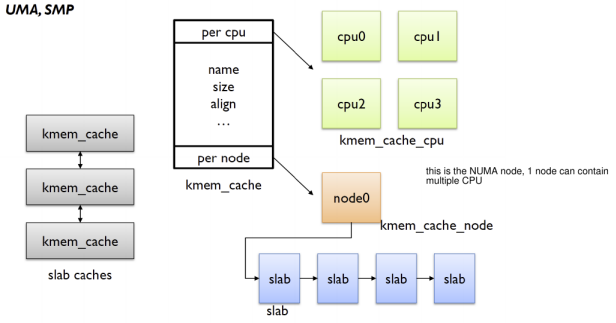
- Only used for free objects
- An integer that points to the next free object
- The last free object uses a terminator value
- Internal descriptors - stored in the slab
- External descriptors - stored in generic caches



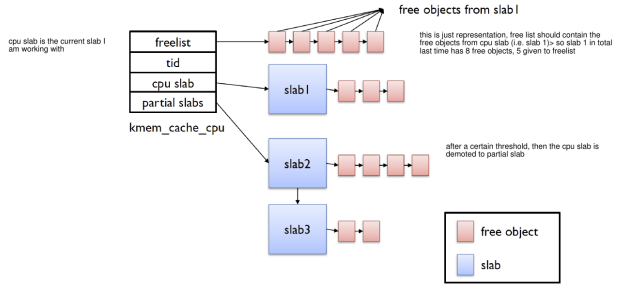
Slab colouring : add different offset in each page so that they will get cached to different cache line to prevent false sharing and cache trashing



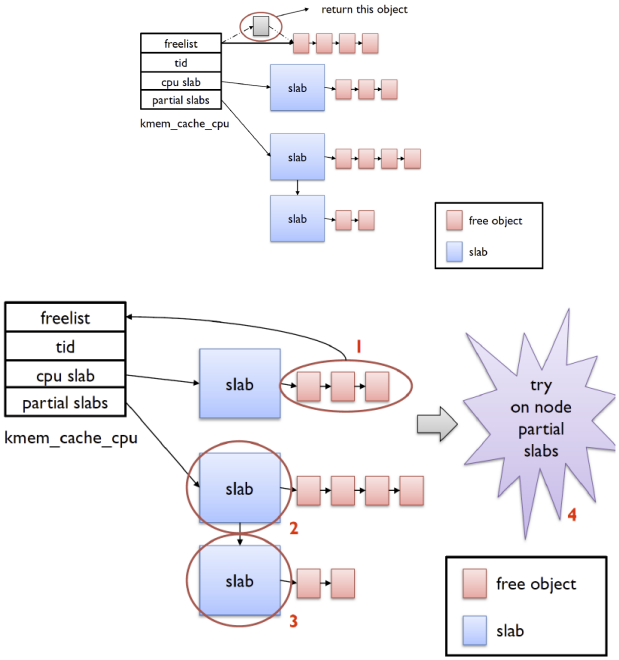
SLUB allocator
used mainly for NUMA (non uniform memory access)
kmem_cache is global ds

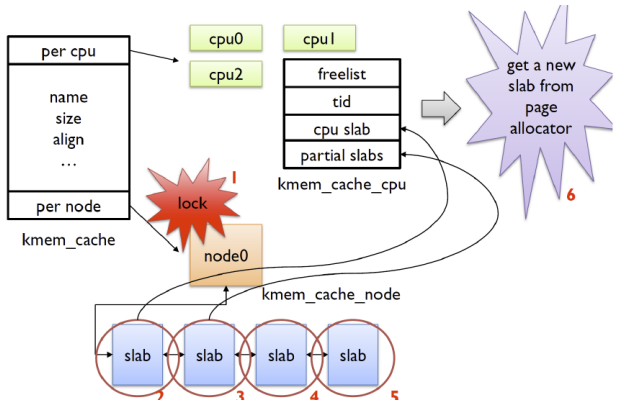


Per-CPU lists



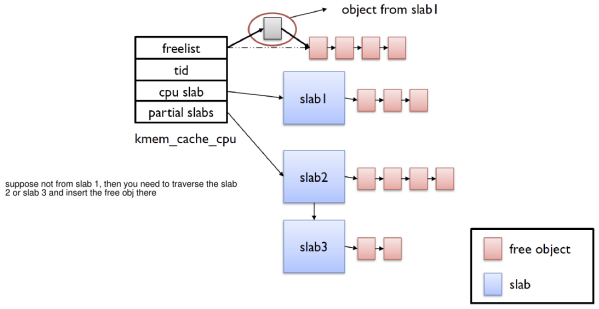
Allocation: Fast Path



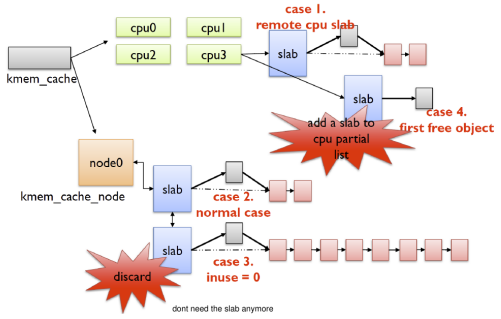


When you allocate new slab, put them in partial slab and linked them to global slab, cpu slab is used for frequently used slab.

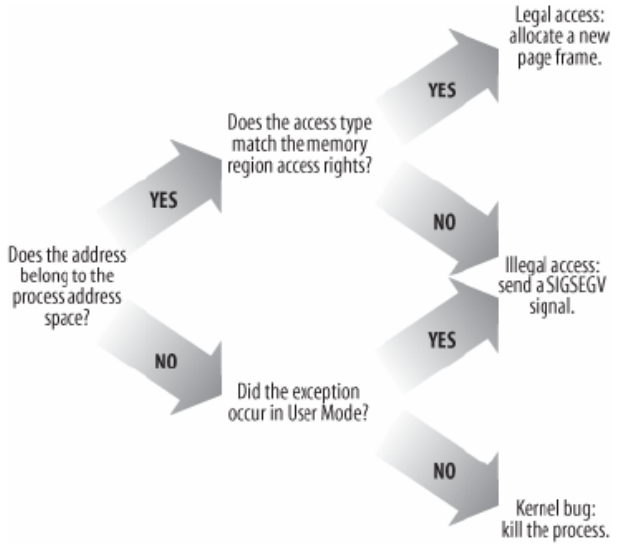
Free: Fast Path



Free: Slow Path



Page Fault



Kernel is not pageable (swapped to memory) because kernel maybe holding locks, etc and tedious to implement.

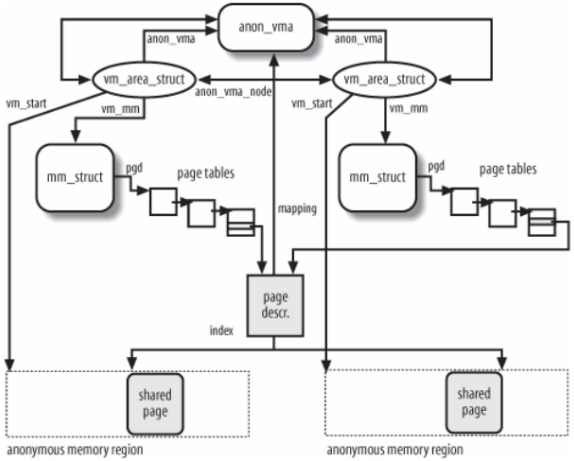
Kernel only allows page fault usually if it is copying to and from the userspace.

Page Frame Reclamation Algo

1. Basically page frame is either reclaimable (swappable, syncable, discardable) or not reclaimable

Mapped page=> part of a file (can be swapped to the disk directly), anon if its not part of a file (heap, stack, etc) and must be swapped to swap area

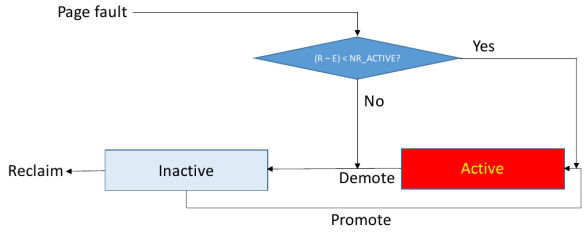
Reverse Mapping



The idea is
• When first page frame is assigned to an anonymous region, a new **anon_vma** is created
• When a page frame already referenced by one process is inserted into a PTE of another process, the kernel simply inserts the anonymous memory region of the second process in the doubly linked circular list of the anon_vma data structure pointed to by the first process
• To find all other PTEs for a page frame to be reclaimed is a linear traversal of the list

For mapped pages, instead of anon_vma, just use address_space

Page Reclamation (capturing working set)



Pages that are accessed multiple times on the inactive list are promoted to the active list
Minimum Access Distance = NR_inactive + (R – E) => refault distance

R > E and R, E are sum of activations + evictions at time t

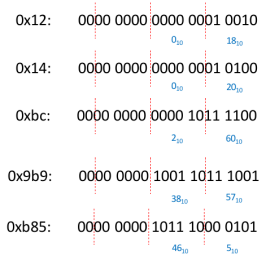
The idea is that if we have R - E pages, we would not evict the pages but activate the pages instead, so the heuristic is that if we use NR_active as extra slots, then if (R- E) < NR_ACTIVE, put in active

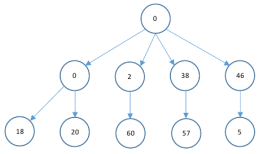
The counter E is called shadow entry, put in radix tree slot of evicted page

The Page Cache

Linux uses write-back caching, i.e. changes always written to cache, make it dirty or not and periodically flush to buffers

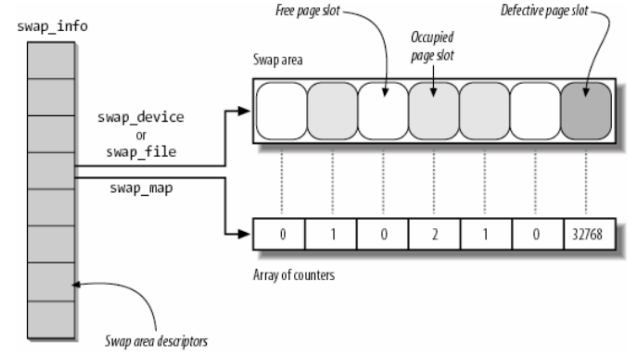
page_tree: for fast query about the existence of a page in the page cache (inside address space or page frame)





take 6 bits at a time of the page index (position of page within a file) until level 6 (consider bit 0 to 35), and a page takes 12 bits

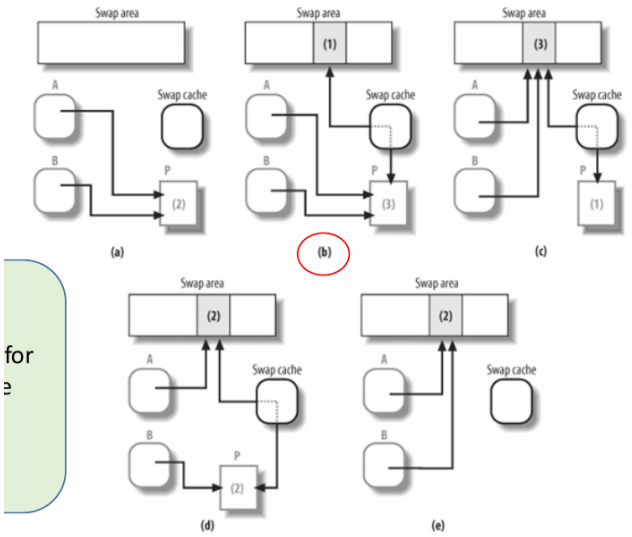
Swapping



counter indicates how many processes use this page

A page that is swapped out identified by page slot index, swap area it is in, stored as unsigned long

Race condition: swap cache
To solve multiple swap in or concurrent swap in + out
Key rule: nobody can start a swap-in or swap-out without checking whether the swap cache already includes the affected page



When shared pages are swapped in, they are therefore retained in the swap cache until all processes have requested the page from the swap area and are all thus aware of the new position of the page in memory.

Without the aid of the swap cache, the kernel is not able to determine whether or not a shared memory page has already been swapped back in, and this would inevitably result in redundant reading of data.

Swapping is batched because the overhead is high

Swapping In Pages

1. Check swap cache, if page is there, skip to Step 3
2. Read in pages from disks
   a. For each page, check page cache by looking up radix tree, check swap cache before trying to read from disk
3. Check page cache by looking up radix tree, check swap cache again – to make sure we got all pages in correctly
   a. Tricky race conditions and various technical issues can trip up 2
4. Free swap area page slots
5. Fix up page table entries of process

Checking up => insert to cache, at the end, remove the pages from the swap cache => if you try to get the swap before finished being read, sleep first.

Swapping out pages

- Insert pages into swap cache
- Get allocation of swap area page slots
- Update page table entries
- Write pages into swap area

then remove the pages from the swap cache

Regular maintenance:

1. kernel swap daemon (kswapd)
2. cache_reap => periodically clear unused slabs

zswap is a Linux kernel feature that provides a compressed write-back cache for swapped pages, as a form of virtual memory compression.

Synchronization

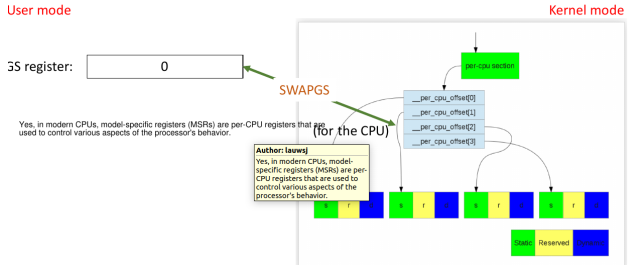
Overview

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local CPU or All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data structures through pointers	All CPUs

Linux per-CPU variables

Macro or function name	Description
`DEFINE_PER_CPU(type, name)`	Statically allocates a per-CPU array called `name` of `type` data structures
`per_cpu(name, cpu)`	Selects the element for CPU `cpu` of the per-CPU array `name`
`__get_cpu_var(name)`	Selects the local CPU's element of the per-CPU array `name`
`get_cpu_var(name)`	Disables kernel preemption, then selects the local CPU's element of the per-CPU array `name`
`put_cpu_var(name)`	Enables kernel preemption (`name` is not used)
`alloc_percpu(type)`	Dynamically allocates a per-CPU array of `type` data structures and returns its address
`free_percpu(pointer)`	Releases a dynamically allocated per-CPU array at address `pointer`
`per_cpu_ptr(pointer, cpu)`	Returns the address of the element for CPU `cpu` of the per-CPU array at address `pointer`

Reduce contention
the KERNEL_GS_BASE MSR (Model-Specific Register) is a per-CPU register that is used by the kernel to store the base address of the per-CPU data structure.



Atomic Operations

```
atomic_t v;
atomic_set(&v, 5);                  /* v = 5 (atomically) */
atomic_add(3, &v);                  /* v = v + 3 (atomically) */
atomic_dec(&v);             /* v = v - 1 (atomically) */
printf("This will print 7: %d\n", atomic_read(&v));
```

Implemented in assembly or defined in the ISA

Barrier

Memory barrier: prevent read-write reordering, optimization barrier: prevent C statements to be mixed around
MP: memory protection
UP: Uni processor

`mb()`	Memory barrier for MP and UP
`rmb()`	Read memory barrier for MP and UP
`wmb()`	Write memory barrier for MP and UP
`smp_mb()`	Memory barrier for MP only
`smp_rmb()`	Read memory barrier for MP only
`smp_wmb()`	Write memory barrier for MP only

Spin Locks

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
spin_lock(&mr_lock);            /* critical section ... */
spin_unlock(&mr_lock);
```

Ticket Lock: (basically get a ticket and wait)

```
1  ticketLock_init(int *next_ticket, int *now_serving)
2  {
3      *now_serving = *next_ticket = 0;
4  }
5
6  ticketLock_acquire(int *next_ticket, int *now_serving)
7  {
8      my_ticket = fetch_and_inc(next_ticket);
9      while(*now_serving != my_ticket) {}
10 }
11
12 ticketLock_release(int *now_serving)
13 {
14     *now_serving++;
15 }
```

Atomic operations!

•If data is only accessed in interrupt context and is local to one specific CPU we can use interrupt disabling to synchronize

If data is accessed from other CPUs we need additional synchronization
• Spin locks
• Spin locks can not be acquired in interrupt context because this might deadlock

• Non-interrupt code acquires spin lock to synchronize with other non-interrupt code and disables interrupts to synchronize with local invocations of the interrupt handler

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags);  /* critical section ... */
spin_unlock_irqrestore(&mr_lock, flags);
```

`spin_lock_irqsave()`
- disables interrupts locally
- acquires the spinlock using instructions required for SMP

`spin_unlock_irqrestore()`
- Restores interrupts to the state they were in when the lock was acquired

Variation: spin_lock_bh() => disable softirq, needed for code outside softirq that manipulates data inside softirq

• Do not try to re-acquire a spinlock you already hold! It leads to self deadlock!
• Spinlocks should not be held for a long time: Excessive spinning wastes CPU cycles! (use semaphore otherwise)
• Do not sleep while holding a spinlock!

Semaphore

1. Wait Queue + Usage Count

down() => acquire
up() => release

variation: down_interruptible(), down_trylock()

Reader-Writer Spinlock

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;

read_lock(&mr_rwlock);
/* critical section (read only) ... */
read_unlock(&mr_rwlock);

write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_rwlock);
```

(multiple readers, 1 writer)
Reader cannot be upgraded to writer variant, so need to release and acquire again

For semaphore (simpler version of lock, apparently the lock can do priority inheritance, etc)

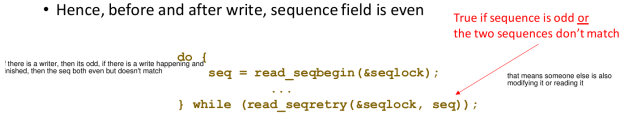
```
struct rw_semaphore mr_rwsem;
init_rwsem(&mr_rwsem);

down_read(&mr_rwsem); /* critical region (read only) ... */
up_read(&mr_rwsem);

down_write(&mr_rwsem); /* critical region (read and write) ... */
up_write(&mr_rwsem);
```

Big reader lock
Specialized form of reader/writer lock
• Very fast to acquire for reading
• Very slow to acquire for writing
• Good for read-mostly scenarios

Seqlock
Similar to Read-Write Locks but writers get much higher priority
Reader:

```
do {
    seq = read_seqbegin(&seqlock);
        ...
} while (read_seqretry(&seqlock, seq));
```

- Writer must
  - Acquire spinlock and increase the sequence field (in `write_seqlock()`)
  - Do its write
  - Increase the sequence field once more then release spinlock (in `write_sequnlock()`)
  - Hence, before and after write, sequence field is even



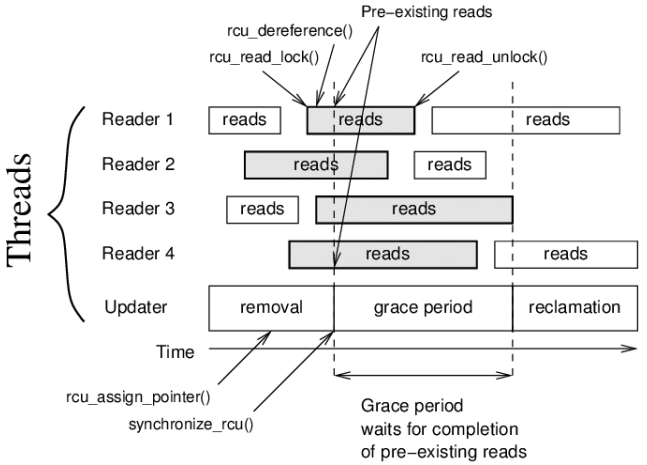
The data structure to be protected does not include pointers that are modified by the writers and dereferenced by the readers
• Otherwise, a writer could change the pointer under the nose of the readers

The code in the critical regions of the readers does not have side effects
• Otherwise, multiple reads would have different effects from a single read

Big Kernel Lock
A global kernel lock

Read Copy Update
• Allows many readers and many writers to proceed concurrently
• RCU is lock free
• Only data structures that are dynamically allocated and referenced by means of pointers can be protected by RCU

rcu read lock => preempt disable
rcu read unlock => preempt enable (must be also called before process switch, start idle loop, go to user mode)



- Writer makes a copy of data structure by dereferencing the pointer
  - All changes made strictly to the copy, never to the source
- Once finished, writer changes the pointer to the data structure atomically (actually it has grace period, to allow the pre existing reading threads all to finish)
- So readers only see old or new copy of data structure but not both

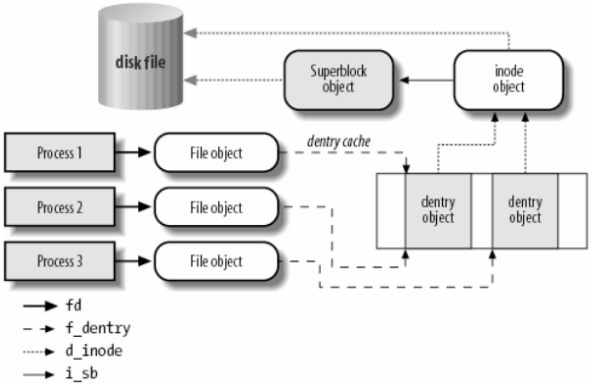
Kernel control paths accessing the data structure	UP protection	MP further protection
Exceptions	Semaphore	None
Interrupts	Local interrupt disabling	Spin lock
Deferrable functions	None	None or spin lock
Exceptions + Interrupts	Local interrupt disabling	Spin lock
Exceptions + Deferrable functions	Local softirq disabling	Spin lock
Interrupts + Deferrable functions	Local interrupt disabling	Spin lock
Exceptions + Interrupts + Deferrable functions	Local interrupt disabling	Spin lock

Completion
Semaphore with minimized data race (apparently semaphore can be destroyed in the middle of operation by another thread), can use complete() for up(), wait for completion for down()

basically its optimized for event telling them to wait, while semaphore is optimized for non-contention

Filesystem

VFS exposes a uniform API regardless of the underlying implementation or physical realities



Common file Model

• The superblock object: stores information about a mounted filesystem: A filesystem is mounted if it is properly set up for use and access => doubly linked list filesystem control block

• The inode object: stores the general information about a specific file (file control block) => there is inode cache

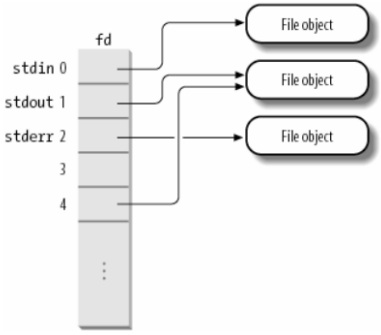
• The file object: stores information about the interaction between an opened file and a process: Exists only in kernel memory during the period when a process has the file opened

• The dentry object: stores information about the linking of a directory entry with the corresponding file => normalization of directory in the memory => there is dentry cache

VFS inode is an in-memory representation of a file or directory that is used by the Linux kernel, while FS inode is a data structure used by a specific file system to represent a file or directory on disk.

VFS inode contains pointer to i_op and i_fop for performing inode and file operations abstractions.

Each process maintains a list of opened files and fds



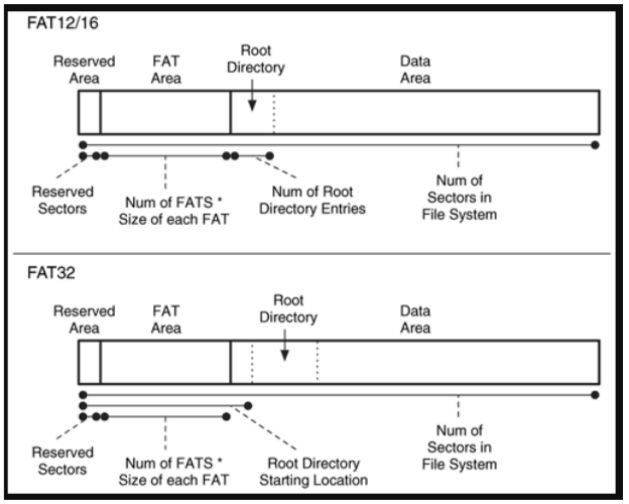
Common fs in linux

Name	Mount point	Description
bdev	none	Block devices
binfmt_misc	any	Miscellaneous executable formats
devpts	*/dev/pts*	Pseudoterminal support (Open Group's Unix98 standard)
eventpollfs	none	Used by the efficient event polling mechanism
futexfs	none	Used by the futex (Fast Userspace Locking) mechanism
pipefs	none	Pipes
proc	*/proc*	General access point to kernel data structures
rootfs	none	Provides an empty root directory for the bootstrap phase
shm	none	IPC-shared memory regions
mqueue	any	Used to implement POSIX message queues
sockfs	none	Sockets
sysfs	*/sys*	General access point to system data
tmpfs	any	Temporary files (kept in RAM unless swapped)
usbfs	*/proc/bus/usb*	USB devices

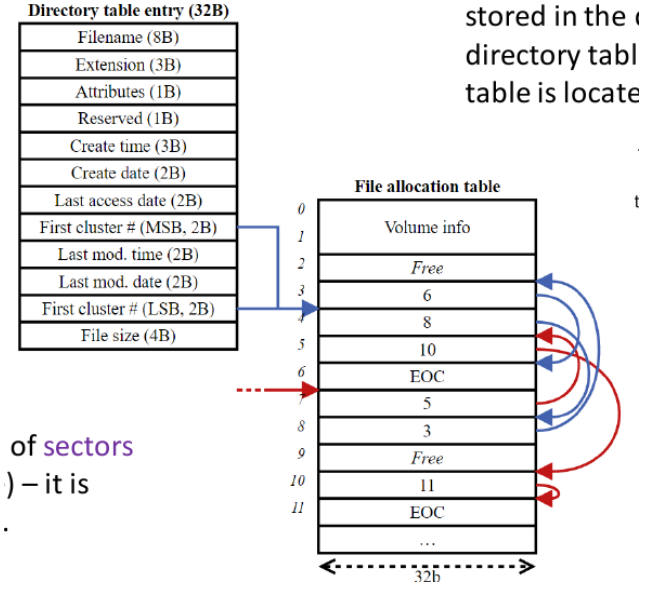
Multiple mounting => one superblock object

FAT

FAT (8 bit) → FAT12 → FAT16 → FAT32 → exFAT (64 bits)



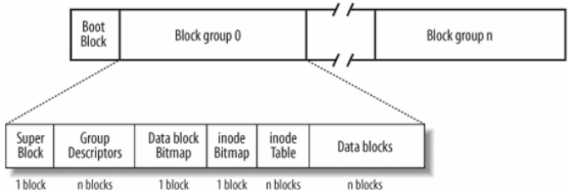
FAT32 entry with short file name



EXT2

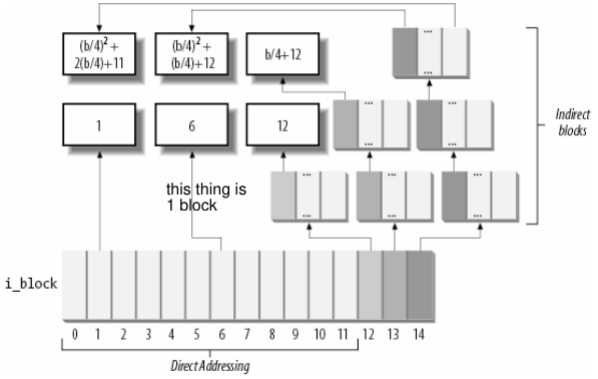
1. No journaling
2. Maximum individual file size can be from 16 GB to 2 TB

3. Overall file system size can be from 2 TB to 32 TB



Block size is selectable at creation. From 1024 to 4096 bytes.

block group size is configurable, with at most 8x block size since bitmap 1 block => 1 byte = 8 bits



Blocks = ceil(NUMBER_OF_BYTES / 1024)

indirect ptr blocks = ceil((Blocks - 12) / 256)
doubly indirect ptr blocks = ceil ((Blocks - b/4 - 12) / (256 * 256))
trebly indirect ptr blocks = ceil ((Blocks - $(b/4)^2$ - b/4 - 12) / (256 * 256 * 256))

Directory is just another file with special bit

VFS and ext2

Type	Disk data structure	Memory data structure	Caching mode
Superblock	`ext2_super_block`	`ext2_sb_info`	Always cached
Group descriptor	`ext2_group_desc`	`ext2_group_desc`	Always cached
Block bitmap	Bit array in block	Bit array in buffer	Dynamic
inode bitmap	Bit array in block	Bit array in buffer	Dynamic
inode	`ext2_inode`	`ext2_inode_info`	Dynamic
Data block	Array of bytes	VFS buffer	Dynamic
Free inode	`ext2_inode`	None	Never
Free block	Array of bytes	None	Never

Attempts to allocate each new directory in the group containing its parent directory

Also attempts to place files in the same group as their directory entries

EXT3

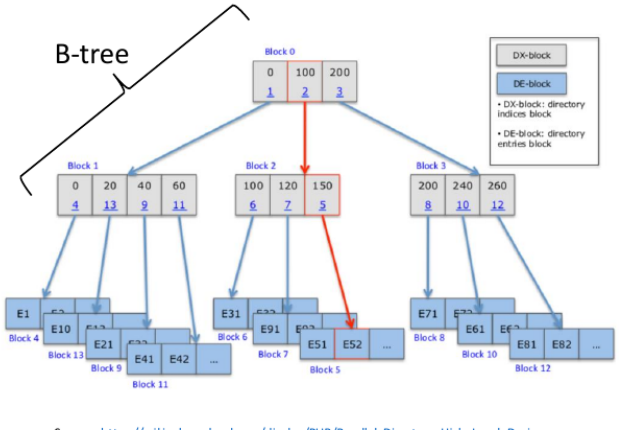
1. Journaling (Journal, save content + metadata, ordered => save metadata, writeback => not immediately written to journal)

Benefit of journaling => write to journal first, if crashes, can recover

1. data is committed to the journal first then
2. start I/O, once finished remove journal entry

2. Maximum individual file size can be from 16 GB to 2 TB
3. Overall ext3 file system size can be from 2 TB to 32 TB

initially in ext2, directory stored in FAT table like manner, in ext3 you can convert them to HTree (Hashed Btree)



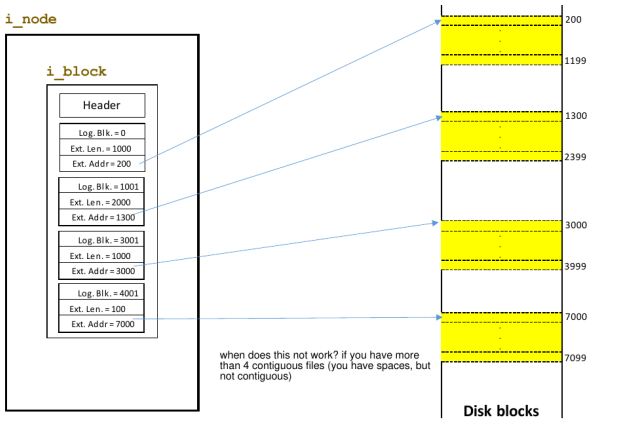
EXT4

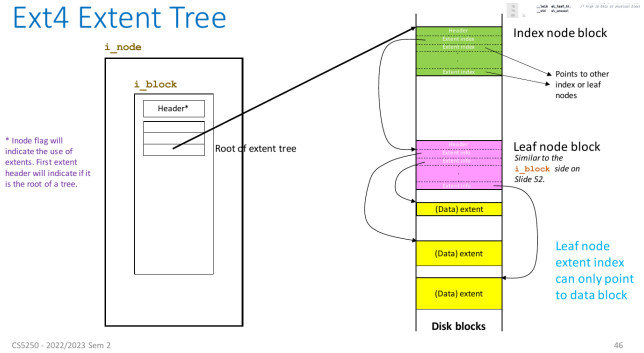
1. Improved performance and reliability
2. Maximum individual file size can be from 16 GB to 16 TB
3. Overall maximum ext4 file system size is 1 EB (exabyte). 1 EB = 1024 PB (petabyte). 1 PB = 1024 TB (terabyte).
4. Directory can contain a maximum of 64,000 subdirectories (as opposed to 32,000 in ext3)

Intro to extent:

1. A range of continuous disk blocks

- `i_block` (60 bytes) can be used to store
  - the same block structure (direct, indirect, doubly indirect, triply indirect) as ext2/3
  - Or an extent header and 4 extents (12 bytes each)
    - Indicated in `i_flags`
  - Or the root of an extent B-tree





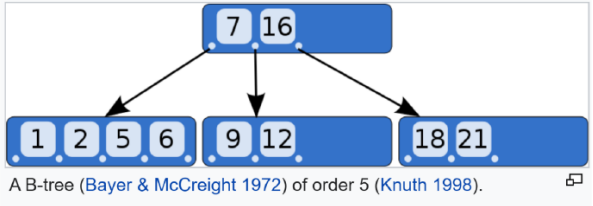
Too many improvements:

1. metadata checksumming, faster fs checking, journal checksumming, delayed allocation, etc

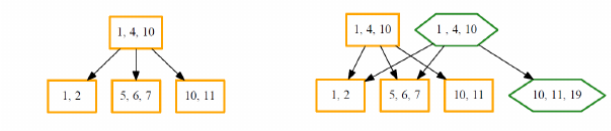
BTRFS

1. B-Tree based + CoW

B-Tree also stored data in intermediary node, no double linked list in leaf nodes



CoW friendly B-tree



For every node that is deleted, create a parent, pointing to ori nodes

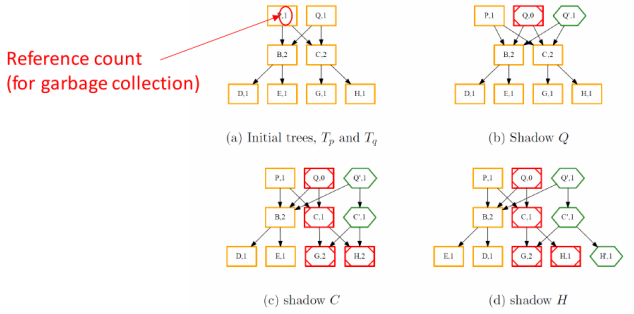


Figure 8: Inserting a key into node H of tree T_q. The path from Q to H includes nodes $\{Q, C, H\}$, these are all COWed. Sharing is broken for nodes C and H; the ref-count for C is decremented.

The original node RC Q is decremented since its not pointed anymore. Since its 0, it can be ignored => total counter must be preserved before and after splitting. (i.e. if initial 1 => after splitting 0 and 1)

A btrfs B-tree only has three types of data structures: keys, items, and block headers.

- Internal tree nodes hold only `[key, block-pointer]` pairs.
- Leaf nodes hold arrays of `[item, data]` pairs.

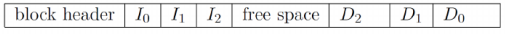
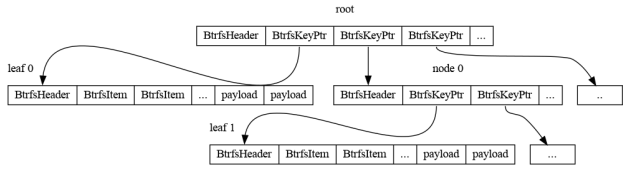


Figure 10: A leaf node with three items. The items are fixed size, but the data elements are variable sized.

item contains offset + size

overview

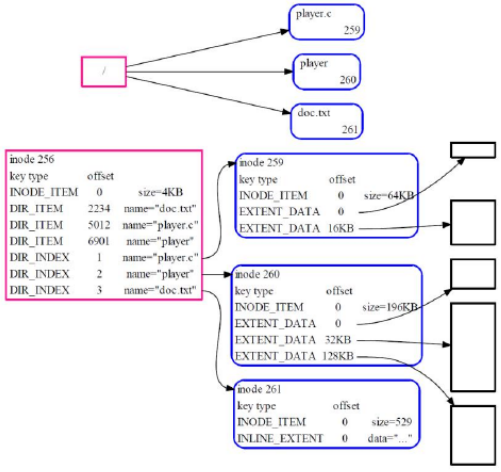


small file is stored using btree
large file is stored using extents

Inodes are stored in an inode item at offset zero in the key, and have a type value of one.

Extent:

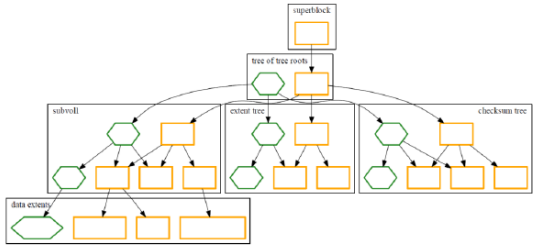
- a [disk block, length] pair to record the area on disk
- a [logical file offset, length] pair to record the file area.



FS trees (Just for info)

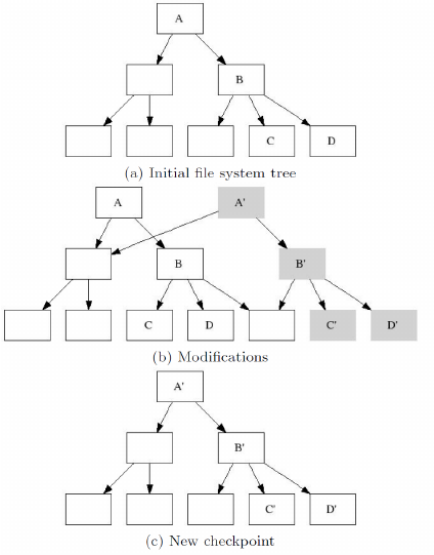
- A superblock located at a fixed disk location is the anchor. It points to a tree of tree roots, which indexes the b-trees making up the filesystem. The trees are:
  - Sub-volumes: store user visible files and directories.
  - Extent allocation tree: tracks allocated extents in extent items, and serves as an on-disk free-space map.
  - Checksum tree: holds a checksum item per allocated extent.
  - Chunk and device trees: indirection layer for handling physical devices. Allows mirroring/striping and RAID.
  - Reloc tree: for special operations involving moving extents.

Actual fs after modification



(b)

Once in a while, checkpointing occurs to merge the forests



(a) Initial file system tree

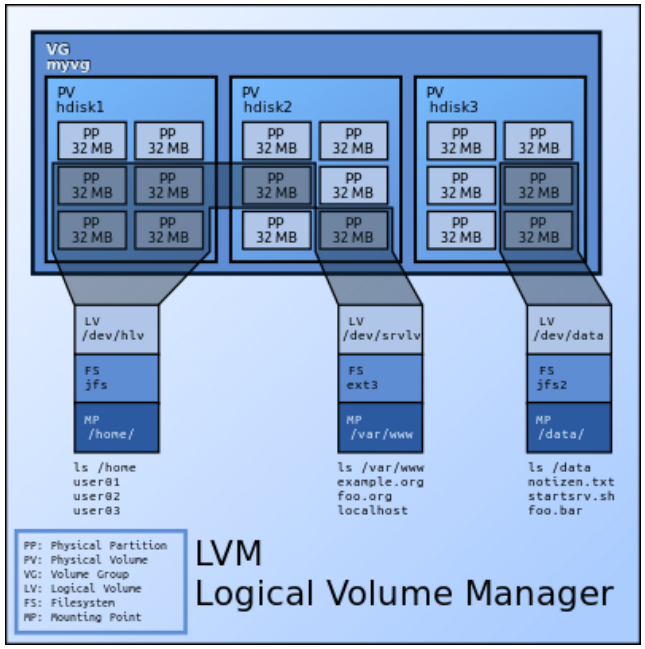
(b) Modifications

(c) New checkpoint

Fig. 4. Checkpoints.

Logical Volume

In Linux, Logical Volume Manager is a device mapper framework that provides logical volume management for the Linux kernel



Device Driver

I/O port => 16 bit address to transfer 8, 16 or 32 bits of data to a distinct register

I/O interface is just a group of I/O ports + controller

Port is connected to a certain register / pin such as control register, status register, input and output registers.

Interface combine these pins into an interface like USB, VGA, etc.

A device controller is a system that handles the incoming and outgoing signals of the CPU by acting as a bridge between CPU and the I/O devices => receive signal from the device and modify the registers accordingly, and the other way around.

Memory-mapped I/O
I/O is embedded inside the memory space, use load and store to modify it

Device Driver
Basically black boxes to hide hardwares and handle it

Usually its policy free, i.e. it doesn't handle logic what kind of fs it supports, the user management, etc (this is handled by another program)

This is often loaded as a module (using insmod and rmmod) and then registered as a device using mknod -m 666 /dev/etx_device (c/b) (major number) (minor number)

Character Device
1. A stream of bytes 9support open, close, read, write usually), example is Serial and consol (ttyS0, console, etc)
2. Cannot be access randomly, only serially
3. On init, a character device driver specify much what range of device numbers it will handle

register_chrdev_region(MKDEV(CDRV_MAJOR, 0), CDRV_MAX_MINORS,"cdrv_device_driver");

Block Device
1. Array of storage blocks
2. read in unit of blocks
3. Has random access

Network Device
1. abstraction of data packets, support protocol and streams related to packet transmission

There are other devices such as USB, FireWire, Magnetic Tape Device, etc.

Issue with modules: kernel only does simple check, can insert random modules that affect hardware and global resources.

sysfs
1. Everything is a file

- The components of sysfs
  - **block** – block devices
  - **devices** – all hardware devices recognized by the kernel
  - **bus** – buses
  - **drivers** – the device drivers
  - **class** – the main category of devices
  - **power** – for power management
  - **firmware** – handle firmware

Name	Type	Major	Minor	Description
/dev/fd0	block	2	0	Floppy disk
/dev/hda	block	3	0	First IDE disk
/dev/hda2	block	3	2	Second primary partition of first IDE disk
/dev/hdb	block	3	64	Second IDE disk
/dev/hdb3	block	3	67	Third primary partition of second IDE disk
/dev/ttyp0	char	3	0	Terminal
/dev/console	char	5	1	Console
/dev/lp1	char	6	1	Parallel printer
/dev/ttyS0	char	4	64	First serial port
/dev/rtc	char	10	135	Real-time clock
/dev/null	char	1	3	Null device (black hole)

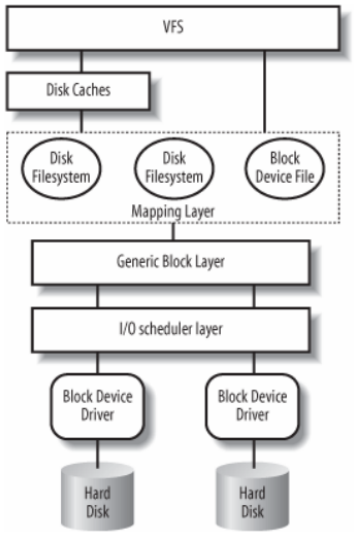
Each device has a device object that has an associated kobject (within the drivers directory in sysfs)

Each class of devices has a class object, a device can belong to multiple class_dvice and each class is expected to offer the same functionality.

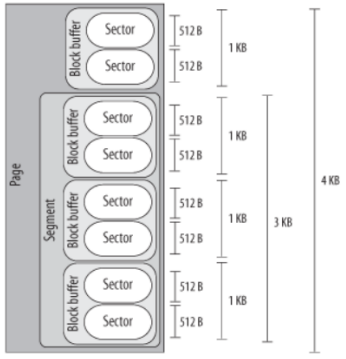
Major => 12 bits
minor => 20 bits

For I/O device:
1. either poll (check whether I/O request is finished)
2. use interrupt

Block Device Driver Overview



- Hardware transfers data in sectors
- VFS uses logical units called blocks
- Block device drivers can work with adjacent data in segments
- Disk caches work with pages



Typical sizes

Block Layer + I/O Scheduling

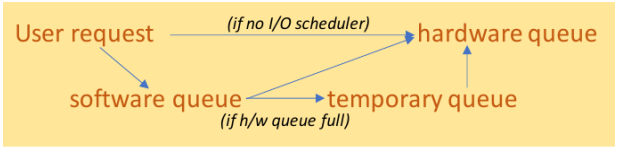
Multi-Queue Block IO Queueing (blk-mq)

Slow devices like hard disk

- A single queue to store block IO requests with a single lock

The purpose of blk-mq is to improve the performance, scalability, and efficiency of block devices in the Linux kernel. Uses 2 queues mainly, hardware and software.

I/O Scheduler tries to group I/O operations for best performance



Various I/O Schedulers

1. NOOP: simplest, no sorting, either FIFO or LIFO
2. Complete Fairness Queuing (CFQ) elevator: maintains a large number of queues (default 64) => each queue is sorted by sectors to reduce disk seeking => like elevator
3. Budget Fair Queueing I/O Scheduler (BFQ)

Each process doing I/O on a device has a weight and a queue: Every queue has a budget, measured in number of sectors

- A queue (process) is given exclusive access to a device, and issues requests to the device until
  1. The queue becomes empty
     - Not immediately, will wait a while to give the process chance to issue more requests
  2. The budget is exhausted
  3. A timeout (to take care of random access)
- If queue is marked "low latency", special heuristics are used instead to ensure low latency
- Early Queue Merge (EQM): used to merge requests to a device
  - A request targeting a device is checked with the next in line on the current active queue of the device (which may belong to another process). If hit, merge!
- Queues are scheduled (selected) based on Worst-case Weighted fair Queueing (WF2Q+)
  - Completely Fair Scheduler also based on similar idea
  - Also uses a red-black tree

4. mq-deadline

4 queues (1 read sorted by sector, 1 write sorted by sector, 1 read sorted by deadline, 1 write sorted by deadline)

- A read (write) request will enter BOTH the read-by-sector (write-by-sector) and read-by-deadline (write-by-deadline)
- A read request expires in 500 ms after entry into queue; 5 sec for write

First (using heuristic) determine whether to do read or write for the next batch of entries in the dispatch queue

If, say, read is chosen, check read deadline queue. If head of that queue has expired, dispatch that. Then try to choose requests adjacent to this from the sector queue.

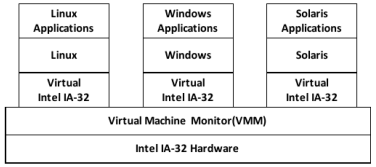
If no expired request, dispatch from where last left out in sector queue: Reset to head when end of queue is reached

5. Kyber

For SSD, NVMe

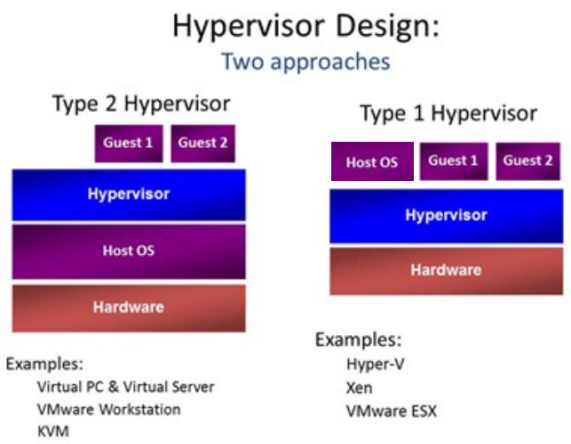
keep the queue short for better response time by tuning the number of entries of sync requests (read) and async request (write) in hardware queue

Virtualization



This is hypervisor, they are the same

virtual box is basically type 2 hypervisor

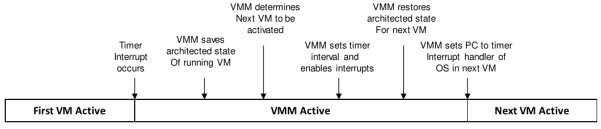


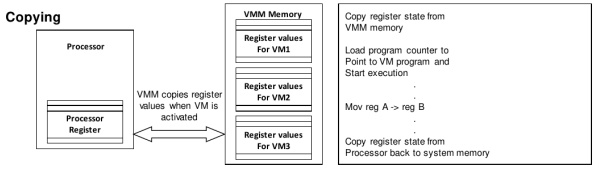
Type 1 is native, Type 2 is User-mode hosted

There are other types:

1. Language virtualization: JVM
2. Cross Platform emulation: Qemu
3. OS-level virtualization: Docker

VM timesharing





The VMM keeps track of the intended mode of operation of a guest virtual machine but it will always use user mode in executing the instructions from the guest virtual machine

A privileged instruction is defined as one that traps if the machine is in user mode and does not trap if the machine is in system mode

Three categories of special instructions:

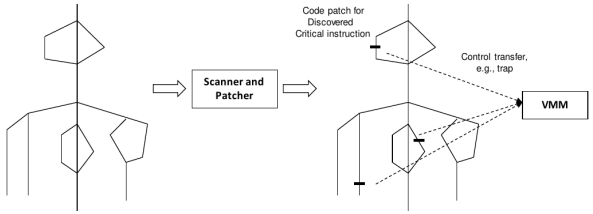
1. Control-sensitive: change the config resources in the system like WRMSR
2. behavior-sensitive: depends on the config like POPF
3. Innocuous: neigher control or behaviour sensitive

Critical instruction: sensitive but not privileged (like POPF) => if its privileged, at least during exception, vmm can handle it
=> virtualizability condition of Theorem 1

IA-32 => 17 critical instructions (protection system references => behaviour depends on the storage protection system, either user or kernel mode and sensitive register instruction => change resource-related registers)

Handling Critical Instruction

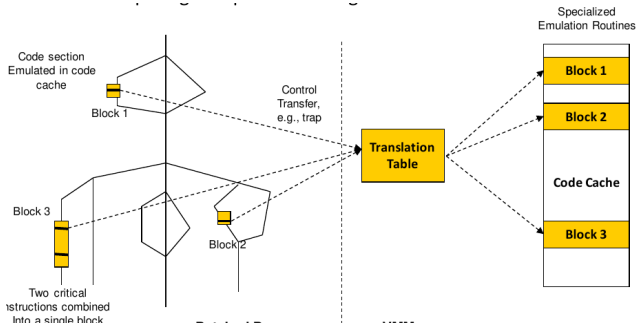
1. Patching



The VMM takes control at the head of each guest basic block and scan instructions in sequence until the end of the basic block is reached

basic block => unit straight-line code sequence with **no branches**

To reduce the overhead, and reuse multiple handlers, we have translation table for emulated code



Popek and Goldberg Virtualization Requirements

A potential virtual machine must satisfy three properties

- Efficiency
- Resource control
- Equivalence

Theorem 1. regarding (efficient) VMM construction

A virtual machine monitor may be constructed if the set of sensitive instruction is a subset of the set of privileged instructions (i.e. no critical instruction)

It means that an efficient virtual machine implementation can be constructed if instructions that could interfere with the functioning of the VMM **always trap in the user mode**

Theorem 2. A conventional (third-generation) computer is recursively virtualizable if
• it is virtualizable and
• a VMM without any timing dependences can be constructed for it

“Timing dependences” = none of the instructions’ execution depends on timing constraints, i.e., no real-time-ness

OS needs to handle timestamp counter itself, like pvclock (and the os must know it might be virtualized)

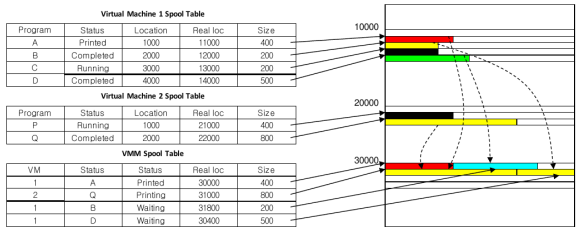
```
PerCPUTime = ((RDTSC() - tsc_timestamp) >> tsc_shift) * tsc_to_system_mul + system_time
```

Theorem 3. A *hybrid* VMM may be constructed for any third generation machine in which the set of user sensitive instructions are a subset of the set of privileged instructions:

Device Virtualization

1. Dedicated device: doesn’t need to be virtualized, just switch from one guest to another (time slicing)
2. Partitioned device: device is partitioned into smaller virtual disks that is available to each virtual machine
3. Shared Devices: just maintain a virtual state for each VM
4. Non existent physical device: just maintain in each VM
5. Spooled device (basically can process I/O while doing other operations)

Maintain VMM spool table in FIFO manner



Virtualizing I/O

1. intercept at I/O operation level
2. Device driver level
3. syscall level (most efficient, intercept at the very beginning)

For hosted vm, you can just translate the I/O for the host machine

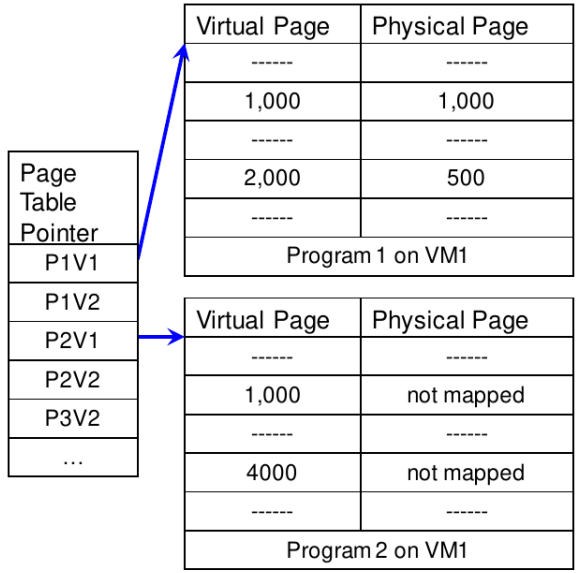
Memory Virtualization

“Architected Page Table” ➔ Hardware maintains page table & instructions regarding it are part of the ISA. The TLB is invisible to ISA and OS.

In newer OS, sometimes they use Architected TLB or software managed TLB and hardware is unaware of paging.

VMM manages virtual-to-physical mappings in a Shadow Page Table, and virtualizes the page table pointer.

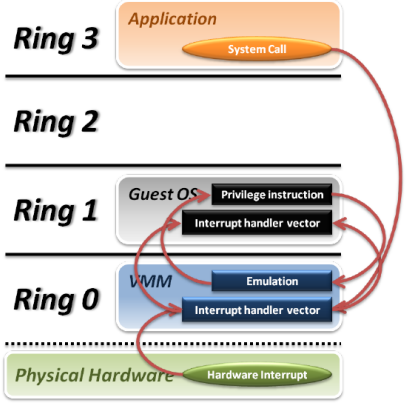
Assuming architedted page table



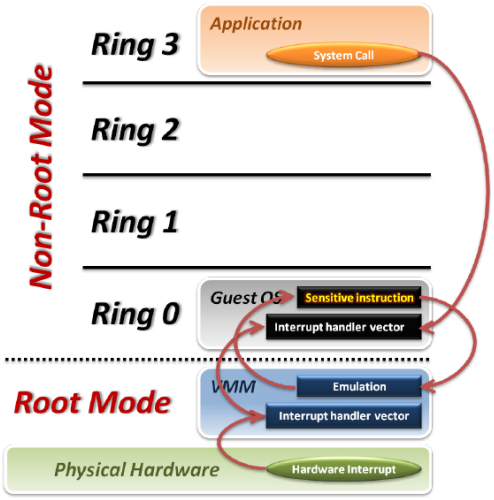
Intel Implementation

CPU Virtualization: Intel VT-x
Memory Virtualization: Extended Page Tables (EPT)
IO Virtualization: Intel VT-d

Trap and Emulate Model
Kernel to Ring 1 => VMM Ring 0

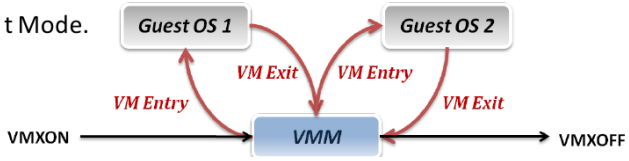


However in Intel VT-x, there is 1 more mode called: Root Mode & Non-Root mode



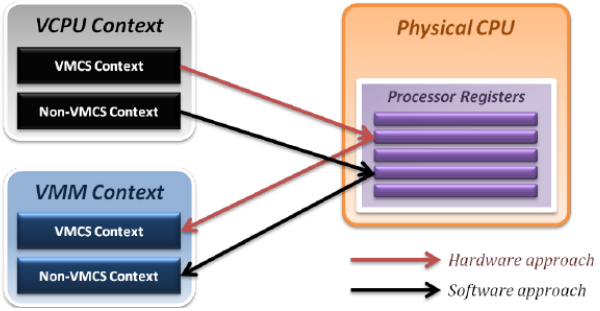
In this case, the Guest OS doesn't need to change anything except for sensitive instructions which will trap to Root Mode

OSes run directly on the hardware => VMM schedule which OS should run

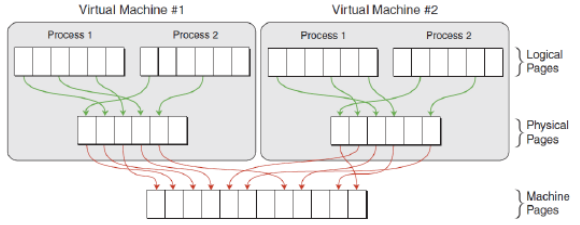


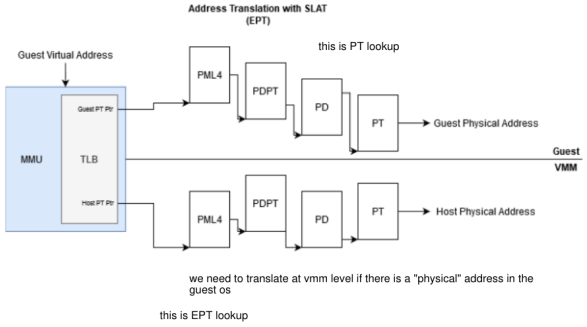
Extra DS for hardware:
VMCS (Virtual Machine Control Structure)

1. State Area: store host OS system in VM-entry, store guest OS in VM-exit
2. Control Area: Control instruction behaviors in Non-Root Mode. Control VM-Entry and VM-Exit process.
3. Exit Information: Provide the VM-Exit reason and some hardware information.



Extended Page Table





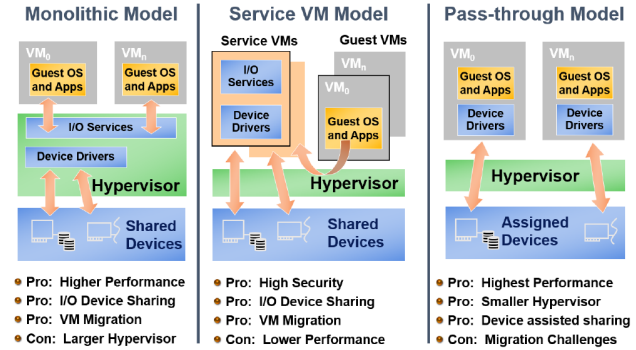
Basically requires 2 full traversal of page tables

EPT misconfig => EPT that contains unsupported value, EPT violation => no misconfig but an access using the guest physical address, both trigger VM Exits

In CR3, we have PCID. But not enough, we need Virtual Processor ID (VPID), Zero if root mode, non-zero if non-root.

TLB => checks VPID and PCID.

I/O Virtualization Options



Intel VT-d

1. Has DMA remapping + support all three models