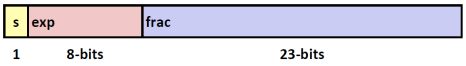
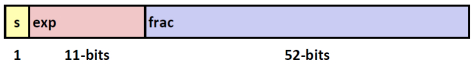
`a | b = ~(~a & ~b)`
`a ^ b = (a & ~b) | (~a & b)`
b = 1 byte, w = 2 bytes, l = 4 bytes, q = 8 bytes

≈ 7 decimal digits, $10^{\pm 38}$

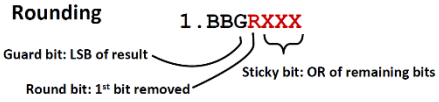


≈ 16 decimal digits, $10^{\pm 308}$



Normalized value (exp ≠ 000…0 and exp ≠ 111…1)
E = Exp – Bias. Bias = $2^{k-1} - 1$
Denormalized Value (exp = 000…0)
Exponent value: E = 1 – Bias (**equispaced**)
Infinity: exp = 111…1, frac = 000…0
NaN: exp = 111…1, frac ≠ 000…0

Rounding



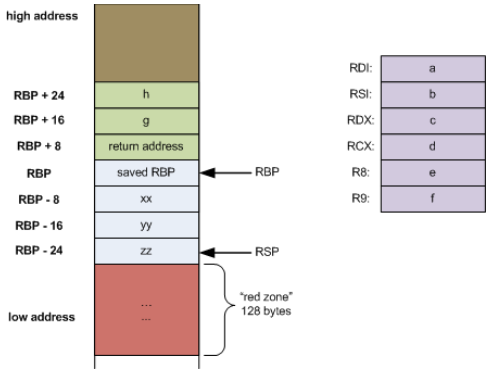
Round up conditions

- Round = 1, Sticky = 1 → > 0.5
- Guard = 1, Round = 1, Sticky = 0 → Round to even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

x86-64 linux calling convention:
Integer parameters:
`%rdi, %rsi, %rdx, %rcx, %r8 and %r9`
Others are stored in stack, pushed in reversed (right-to-left) order



CF Carry Flag (for unsigned) **SF** Sign Flag (for signed)
ZF Zero Flag **OF** Overflow Flag (for signed)

Implicitly set by arithmetic operations (but **not set by `leaq` instruction**):
`addq Src DestDest (t = a + b)`
CF set if carry out from most significant bit (unsigned overflow)
ZF set if t == 0
SF set if t < 0 (as signed)
OF set if two's complement (signed) overflow

Rules for turning on the carry flag
1. The carry flag is set if the addition of two numbers causes a carry out of the most significant bits added.
   1111 + 0001 = 0000 (carry flag is turned on)
2. The carry (borrow) flag is also set if the subtraction of two numbers requires a borrow into the most significant (leftmost) bits subtracted
   0000 - 0001 = 1111 (carry flag is turned on)

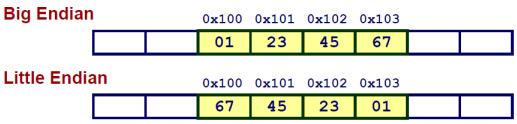
Rules for turning on the overflow flag
1. If the sum of two numbers with the sign bits off yields a result number with the sign bit on
   0100 + 0100 = 1000 (overflow flag is turned on)
2. If the sum of two numbers with the sign bits on yields a result number with the sign bit off
   1000 + 1000 = 0000 (overflow flag is turned on)

Note that different from above (1111 + 0001 = 0000), the result is correct even though CF is set
unsigned arithmetic -> CF | signed arithmetic -> OF

`cmp b, a` Computes *b* - *a* (just like `sub`). Sets condition codes based on result, but **does not change *b***
`test a, b` Computes *b* ∧ *a* just like `and`. Sets condition codes (only SF and ZF) based on result, but **does not change *b***

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)

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF) &~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF) | ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

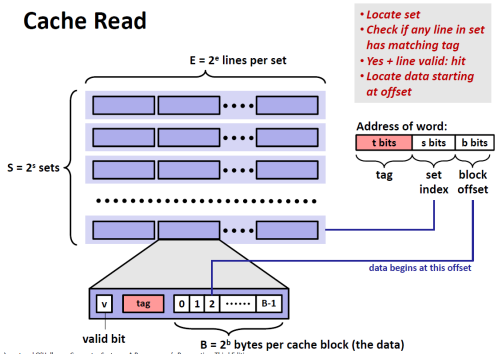


movzbl: zero-extend, byte -> long. **movslq**: sign-extend, long -> quad. Etc.

Buffer overflow attacks
Stack Smashing Attacks: overwrite normal return address. Code Injection Attacks: overwrite normal return address and jump to exploit code
Measures
Avoid overflow vulnerabilities: strcpy -> strncpy. Employ system-level protections: randomized stack offsets, nonexecutable code segments. Have compiler use stack canaries
Return-Oriented Programming Attacks
Work around stack randomization and marking stack nonexecutable. Does not overcome stack canaries

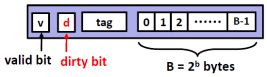
Internal Fragmentation: For a given block, internal fragmentation occurs if payload is smaller than block size
Caused by: Overhead of maintaining heap data structures | Padding for alignment purposes | Explicit policy decisions
Depends only on the pattern of previous requests, easy to measure
Can be reduced by changing our representations of the free list, either through encoding information in unused bits or reducing the size of our free list nodes.
External Fragmentation: Occurs when there is enough aggregate heap memory, but no single free block is large enough
Depends on the pattern of future requests, difficult to measure
Can be decreased by coalescing or using a best-fit algorithm

Cache Read



What about writes?

- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Each cache line needs a dirty bit (set if data has been written to)
- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location will follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

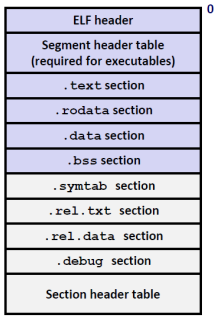


The main benefit of cache over main memory is that you can access data much quicker: ● Caches are built to be small and easy to access ● They are often on or very close to the CPU chip ● They take in the tens of cycles ● Sizes on the order of kilobytes.

Local optimizations work inside a single basic block: Constant folding, strength reduction, dead code elimination, (local) common subexpression elimination, ···
Global optimizations process the entire control flow graph of a function: Loop transformations, code motion, (global) CSE, ···

ELF Object File Format

- **Elf header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - Page size, virtual addresses memory segments (sections), segment sizes.
- **`.text` section**
  - Code
- **`.rodata` section**
  - Read only data: jump tables, ...
- **`.data` section**
  - Initialized global variables
- **`.bss` section**
  - Uninitialized global variables
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space



- **`.symtab` section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **`.rel.text` section**
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- **`.rel.data` section**
  - Relocation info for `.data` section
  - Addresses of pointer data that will need to be modified in the merged executable
- **`.debug` section**
  - Info for symbolic debugging (`gcc -g`)
- **Section header table**
  - Offsets and sizes of each section

Linker's Symbol Rules

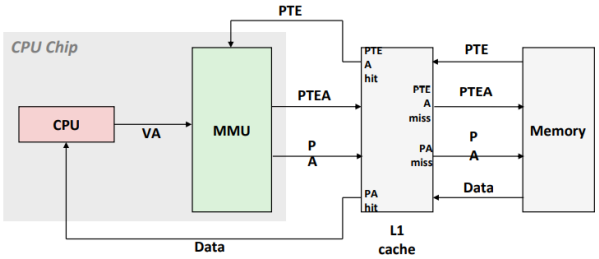
- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol
- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with `gcc -fno-common`

Program symbols are either strong or weak: ▪ Strong: procedures and initialized globals ▪ Weak: uninitialized globals

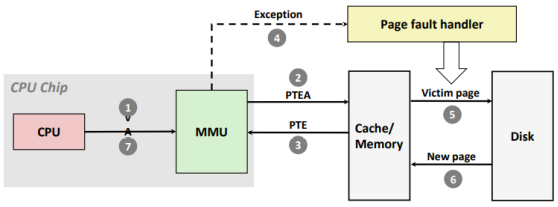
Benefits of virtual memory

Uses main memory efficiently: Use DRAM as a cache for parts of a virtual address space
Simplifies memory management: Each process gets the same uniform linear address space
Isolates address spaces: One process can't interfere with another s memory, User program cannot access privileged kernel information and code

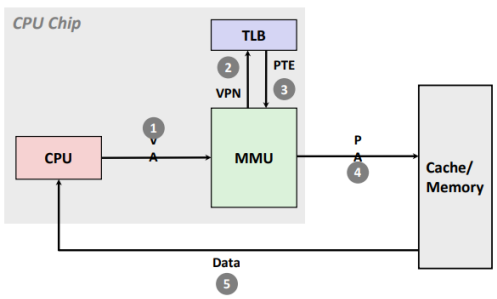
Virtual memory keeps address spaces separate by ensuring that each process has its own page table that maps virtual addresses to physical addresses. This is done inside the operating system. Therefore, multiple processes can access the same virtual address simultaneously--this is possible because the virtual address within each process would map to a different physical page in physical memory.
The OS can share information between address spaces by mapping virtual pages in each of the page tables to the same physical page of memory. This is most useful for code libraries--if multiple processes use the same code library, any process that needs the library can map



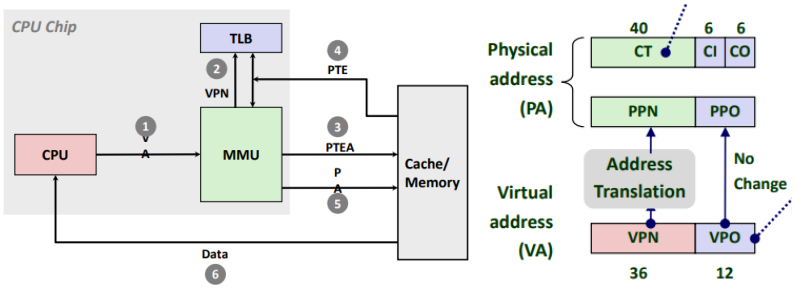
Address Translation: Page Fault

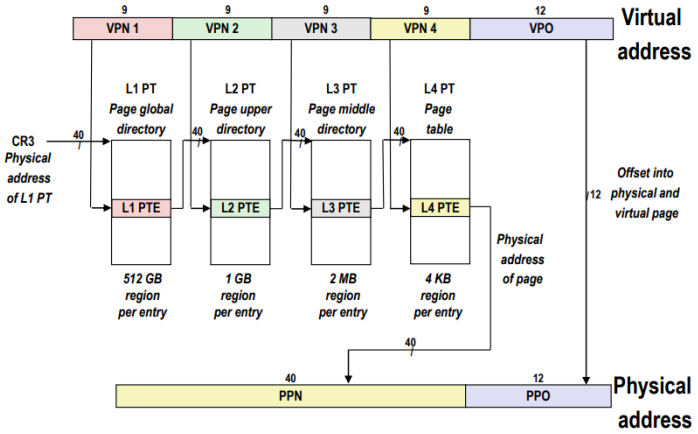


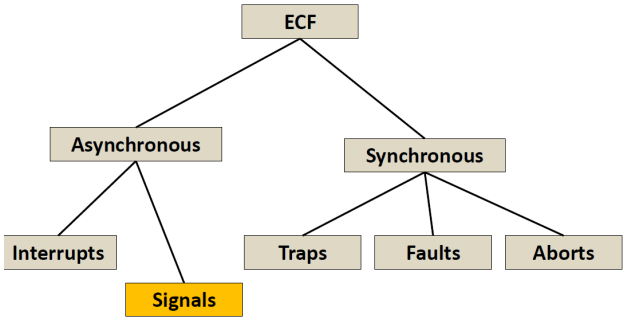
TLB Hit



TLB Miss

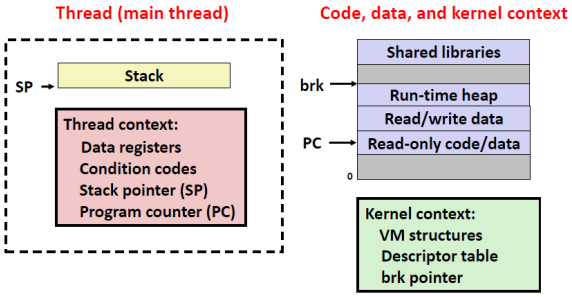






- **G0: Keep your handlers as simple as possible**
  - e.g., set a global flag and return
- **G1: Call only async-signal-safe functions in your handlers**
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe!
- **G2: Save and restore `errno` on entry and exit**
  - So that other handlers don't overwrite your value of `errno`
- **G3: Protect accesses to shared data structures by temporarily blocking all signals**
  - To prevent possible corruption
- **G4: Declare global variables as `volatile`**
  - To prevent compiler from storing them in a register
- **G5: Declare global flags as `volatile sig_atomic_t`**
  - *flag*: variable that is only read or written (e.g. flag = 1, not flag++)
  - Flag declared this way does not need to be protected like other globals

- **Process = thread + code, data, and kernel context**



Classical problem classes of concurrent programs: Races: outcome depends on arbitrary scheduling decisions elsewhere in the system | **Deadlock**: improper resource allocation prevents forward progress | **Livelock/Starvation/Fairness**: external events and/or system scheduling decisions can prevent sub task progress

Pros and Cons of Process-based Servers

- **+ Handle multiple connections concurrently**
- **+ Clean sharing model**
  - descriptors (no)
  - file tables (yes)
  - global variables (no)
- **+ Simple and straightforward**
- **– Additional overhead for process control**
- **– Nontrivial to share data between processes**
  - (This example too simple to demonstrate)

Pros and Cons of Event-based Servers

- **+ One logical control flow and address space.**
- **+ Can single-step with a debugger.**
- **+ No process or thread control overhead.**
  - Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado
- **– Significantly more complex to code than process- or thread-based designs.**
- **– Hard to provide fine-grained concurrency**
  - E.g., how to deal with partial HTTP request headers
- **– Cannot take advantage of multi-core**
  - Single thread of control

Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- **How threads and processes are different**
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

Semaphores

- ***Semaphore:*** **non-negative global integer synchronization variable**
- **Manipulated by *P* and *V* operations:**
  - *P(s):* [`while (s == 0) wait(); s--;`]
    - Dutch for "Proberen" (test)
  - *V(s):* [`s++;`]
    - Dutch for "Verhogen" (increment)
- **OS kernel guarantees that operations between brackets [ ] are executed indivisibly**
  - Only one *P* or *V* operation at a time can modify s.
  - When **`while`** loop in *P* terminates, only that *P* can decrement **`s`**
- **Semaphore invariant:** ***(s >= 0)***

Pros and Cons of Unix I/O

- **Pros**
  - Unix I/O is the most general form of I/O
    - All other I/O packages are implemented using Unix I/O functions
  - Unix I/O provides functions for accessing file metadata
  - Unix I/O functions are async-signal-safe and can be used safely in signal handlers
- **Cons**
  - Dealing with short counts is tricky and error prone
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone

Pros and Cons of Standard I/O

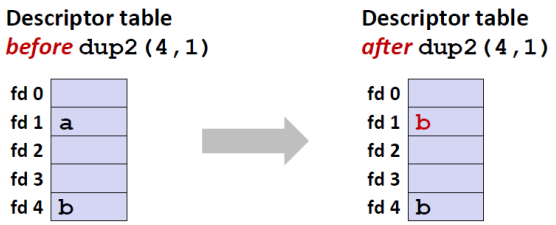
- **Pros:**
  - Buffering increases efficiency by decreasing the number of **`read`** and **`write`** system calls
  - Short counts are handled automatically
- **Cons:**
  - Provides no function for accessing file metadata
  - Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers
  - Standard I/O is not appropriate for input and output on network sockets
    - There are poorly documented restrictions on streams that interact badly with restrictions on sockets (CS:APP3e, Sec 10.11)

I/O Redirection

- **Question: How does a shell implement I/O redirection?**

```
linux> ls > foo.txt
```

- **Answer: By calling the `dup2(oldfd, newfd)` function**
  - Copies (per-process) descriptor table entry **`oldfd`** to entry **`newfd`**



Internet protocol software running on each host and router. Protocol is a set of rules that governs how hosts and routers should cooperate when they transfer data from network to network. Smooths out the differences between the different networks.
1. Provides a naming scheme: An internet protocol defines a uniform format for host addresses. Each host (and router) is assigned at least one of these internet addresses that uniquely identifies it. 2. Provides a delivery mechanism: An internet protocol defines a standard transfer unit (packet). Packet consists of header and payload. Header: contains info such as packet size, source and destination addresses. Payload: contains data bits sent from source host.

Basic Internet Components

- **Internet backbone:**
  - collection of routers (nationwide or worldwide) connected by high-speed point-to-point networks
- **Internet Exchange Points (IXP):**
  - router that connects multiple backbones (often referred to as peers)
  - Also called Network Access Points (NAP)
- **Regional networks:**
  - smaller backbones that cover smaller geographical areas (e.g., cities or states)
- **Point of presence (POP):**
  - machine that is connected to the Internet
- **Internet Service Providers (ISPs):**
  - provide dial-up or direct access to POPs

OSI Model

Application
Presentation
Session
Transport
Network
Data Link
Physical

Internet Model

Application	HTTP	SMTP	SSH	DNS
Security	TLS			
Transport	TCP			UDP
Addressing	IP			
Physical Link	Ethernet	WiFi		SDH

The stacked architecture of internet protocol consists of many protocols that interact with the protocols just above and below the layer of the given protocol. Each layer of a specific network model may be responsible for a different function of the network. Each layer will pass information up and down to the next subsequent layer as data is processed.
Advantages: **Interoperability** and allows for so many protocols supported by the current internet | **Portability**: Layered networking protocols are much easier to port from one system or architecture to another | **Compartmentalization of Functionality**: The compartmentalization or layering of processes, procedures and communications functions gives developers the freedom to concentrate on a specific layer or specific functions within that layer's realm of responsibility without the need for great concern or modification of any other layer.
Disadvantage: Overhead increased due to headers of each layer