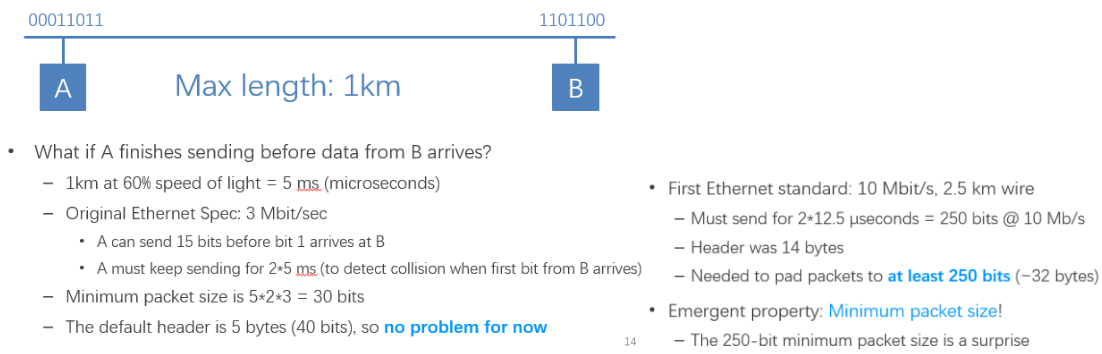
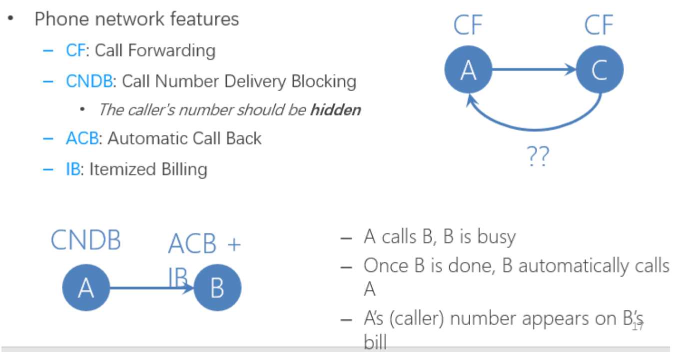
## Lec1 introduction to CSE

1. 复杂性指标 compare with computer system
   Programming/data structure – Loc(lines of code),
   operating system/Architecture – CPU cores
   Network – Nodes,
   Web service – clients

2. Problem type
   a) Emergent properties(surprise) : not considered at design time
   b) Propagation of effects(butterfly effect) : small change cause bif effect
   c) Incommensurate scaling : design for small model may not scale
   d) Trade-offs : watered effect
   e) Example:
      i. 传输网络包：packet size = 传输时间（距离/速度） * 带宽 * 2




      ii. 蝴蝶效应

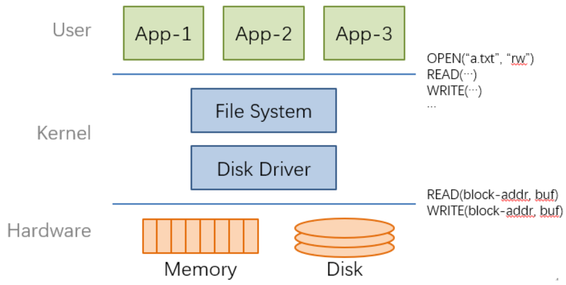



3. Coping with complexity : M.A.T.H

- Modularity
  – Split up system
  – Consider separately
- Abstraction
  – Interface/Hiding
  – Avoid propagation of effects
- Layering
  – Gradually build up capabilities
- Hierarchy
  – Reduce connections
  – Divide-and-conquer

# Lec2 iNode-based file system

1. File
   a) Properties: durable(持久的)，has a name
   b) High-level version of the mempry abstraction
   c) UNIX API: open, read, write, seek, close, fsync, stat, chmod, chown, rename, link, unlink, symlink, mkdir, chdir, chroot, mount, unnount




2. 7 software layers

| Layer | Purpose | |
|---|---|---|
| Symbolic link layer | Integrate multiple file systems with symbolic links. | ↑ |
| Absolute path name layer | Provide a root for the naming hierarchies. | user-oriented names |
| Path name layer | Organize files into naming hierarchies. | ↓ |
| File name layer | Provide human-oriented names for files. | machine-user interface |
| Inode number layer | Provide machine-oriented names for files. | ↑ |
| File layer | Organize blocks into files. | machine-oriented names |
| Block layer | Identify disk blocks. | ↓ |

   a) Block layer

```
procedure BLOCK_NUMBER_TO_BLOCK (integer b) returns block
    return device[b]
```

Super block: 1 superblock per file system, kernel read it when mount FS

Block size: trade-off，通常 512byte 或者 4K

Superblock contains

- Size of the blocks
- Number of free blocks
- A list of free blocks
- Index to next free block
- Lock field for free block and free inode lists
- Flag to indicate modification of superblock
- Size of the inode list
- Number of free inodes
- A list of free inodes
- Index to next free inode

   b) File layer

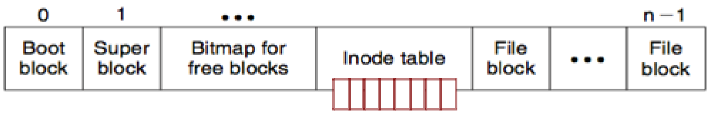
```
procedure INODE_TO_BLOCK (integer offset, inode instance i) returns block
    o ← offset / BLOCKSIZE
    b ← INDEX_TO_BLOCK_NUMBER (i, o)
    return BLOCK_NUMBER_TO_BLOCK (b)

procedure INDEX_TO_BLOCK_NUMBER (inode instance i, integer index) returns integer
    return i.block_numbers[index]
```

Store items larger than 1 block， inode->block number, indirect block

c) Inode number layer

```
procedure INODE_NUMBER_TO_INODE (integer inode_number) returns inode
    return inode_table[inode_number]
```

```
procedure INODE_NUMBER_TO_BLOCK (integer offset, integer inode_number)
                                                returns block
    inode instance i ← INODE_NUMBER_TO_INODE (inode_number)
    o ← offset / BLOCKSIZE
    b ← INDEX_TO_BLOCK_NUMBER (i, o)
    return BLOCK_NUMBER_TO_BLOCK (b)
```




```
structure inode
    integer block_numbers[N]
    integer size
    integer type
    integer refcnt
```

用 inode 就足够操作 file 了，map inode number to inode

d) File name layer

```
procedure NAME_TO_INODE_NUMBER (character string filename, integer dir) returns integer
    return LOOKUP (filename, dir)
```

```
procedure LOOKUP (character string filename, integer dir) returns integer
    block instance b
    inode instance i ← INODE_NUMBER_TO_INODE (dir)
    if i.type ≠ DIRECTORY then return FAILURE
    for offset from 0 to i.size – 1 do
        b ← INODE_NUMBER_TO_BLOCK (offset, dir)
        if STRING_MATCH (filename, b) then
            return INODE_NUMBER (filename, b)
        offset ← offset + BLOCKSIZE
    return FAILURE
```

File name 用来隐藏文件管理的 metadata

e) Path name layer

```
procedure PATH_TO_INODE_NUMBER (character string path, integer dir) returns integer
    if (PLAIN_NAME (path)) return NAME_TO_INODE_NUMBER (path, dir)
    else
        dir ← LOOKUP (FIRST (path), dir)
        path ← REST (path)
        return PATH_TO_INODE_NUMBER (path, dir)
```

例子：“project/paper”，上下文是 working directory

f) Absolute path name layer

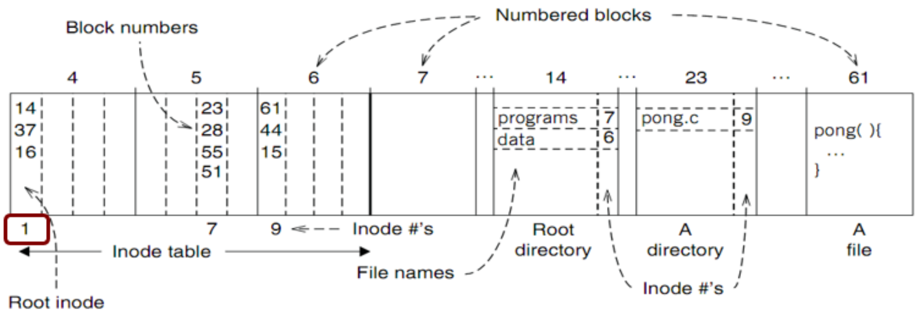
```
procedure GENERALPATH_TO_INODE_NUMBER (character string path) returns integer
    if (path[0] = "/") return PATH_TO_INODE_NUMBER(path, 1)
    else return PATH_TO_INODE_NUMBER(path, wd)
```

‘/’ 是根目录，‘/.’ 和 ‘/..’ 都是 link 到 ‘/’ 的

g) Symbolic link layer

- MOUNT
  - Record the device and the root inode number of the file system in memory
  - Record in the in-memory version of the inode for “/dev/fd1” its parent’s inode
  - **UNMOUNT** undoes the mount
- Change to the file name layer
  - If **LOOKUP** runs into an inode on which a file system is mount, it uses the root inode of that file system for the lookup

Symlink 是软连接




访问顺序：root directory(inode 1) -> block 14(root directory) -> inode 7(block 5) -> block 23(/program) -> inode 9(block 6) -> block 61(pong.c)

## 3. Link

**LINK**: *shortcut* for long names

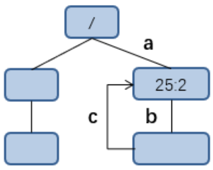
- LINK("Mail/inbox/new-assignment", "assignment")
- Turns strict hierarchy into a directed graph
  - Users cannot create links to directories -> acyclic graph
- Different names, same inode number

**UNLINK**

- Remove the binding of filename to inode number
- If UNLINK last binding, put inode/blocks to free-list
  - A reference counter is needed

a) Link(旧名字，新名字)

b) 除了 '.' 和 '..' ,不允许 link 有循环




- LINK ("/a/b/c", a")
- Cause a cycle!
- Refcnt of a is 2

c) Rename 操作

```
1    UNLINK (to_name)
2    LINK (from_name, to_name)
3    UNLINK (from_name)
```

```
1    LINK (from_name, to_name)
2    UNLINK (from_name)
```

第一个方法如果在 1&2 之间 fail 了，to_name 会丢失，rename 会失败(要求原子性)

第二个方法如果在 1&2 之间 fail 了，一定要 increase inode 的 refcnt

# Lec3 file system API

1. Review questions
   a) Is file name part of file? Data or metadata.
      文件名不是文件的一部分。Inode 层看是 data，应用层看是 metadata
   b) What is the actual content of a directory? Size?
      目录中是 name – inode 对，大小与文件名的长度和文件的数量有关

2. Directory 的数据结构
   通常目录的 size 都不会很大

```
struct ext4_dir_entry {
  uint32_t inode_number;
  uint16_t dir_entry_length;
  uint8_t  file_name_length;
  uint8_t  file_type;
  char     name[EXT4_NAME_LEN];
}
```

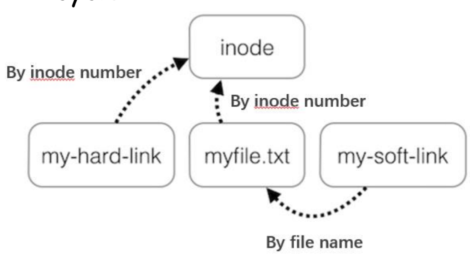
```
0d01 7300 0c00 0102 2e00 0000
b1e7 7200 0c00 0202 2e2e 0000
2b01 7300 0c00 0101 6100 0000
2c01 7300 0c00 0101 6200 0000
2d01 7300 0c00 0101 6300 0000
2e01 7300 c40f 0101 6400 0000
```

```
File Type
  0x0: Unknown
  0x1: Regular file
  0x2: Directory
  0x3: Character device file
  0x4: Block device file
  0x5: FIFO
  0x6: Socket
  0x7: Symbolic link
```

```
0d01 7300 0c00 0102 2e00 0000
```

```
0d01 7300: inode number
     0c00: entry length is 12 bytes
       01: file name length is 1 byte
       01: file type is regular file
2e00 0000: file name (2e -> ".")
```

3. Hard link 和 soft link
   Eg：add link "assignment" to "Mail/new_assignment"
   a) Hard link 并不会生成新的 file，只是添加了"文件名-inode"的 bind；目标 inode 的 refcnt 会增加；即使原先的文件被删除，link 仍然有效；不同的 hard link 之间是平等的
   b) Soft link 会生成新的 file，文件内容是"Mail/new_assignment"；目标 inode 的 refcnt 不会增加；若原先文件被删除，link 会失效；可以构造 Symlink(a,a)这样的 cycle




4. File system API
   a) API：open, read, write, close；  create, rename, link, unlink;   symlink
      mkdir, chdir;   mount, unmount; sync
      这些都被 implement 成了 system call

b) File meta-data：inode

```
structure inode
    integer block_numbers[N]
    integer size
    integer type
    integer refcnt
    integer userid
    integer groupid
    integer mode
    integer atime
    integer mtime
    integer ctime
```

- Types of permission
  - Owner, group, other
  - Read, write, execute
- Time stamps
  - Last access (by READ)
  - Last modification (by WRITE)
  - Last change of inode (by LINK)

c) File descriptor 文件描述符

每一个 process 都有自己的 fd name space

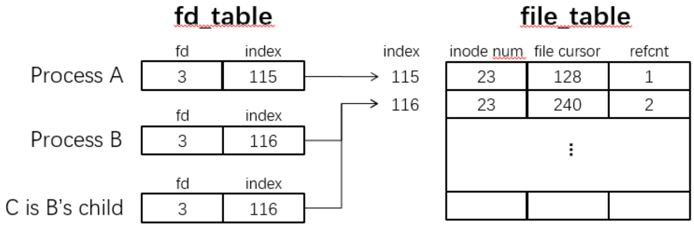
Fd 也可以支持 keybord，display 等 IO device，操作 fd 0 和 fd 1 就可以

Standard in: fd 0, standard out fd 1, standard error: fd 2

d) File cursor

修改：可以通过 seek 操作修改

共享：父子进程是共享 file cursor 的，两个 process 打开同一个文件不共享




- Process A, B and C all open just one file with inode number 23
- Process A and B open the same file, not share file cursor
- Process B and C share the file cursor

## 5. Implementation

### a) Read

```
procedure READ (fd, character array reference buf, n)
    file_index ← fd_table[fd]
    cursor ← file_table[file_index].cursor
    inode ← INODE_NUMBER_TO_INODE (file_table[file_index].inode_number)
    m = MINIMUM (inode.size – cursor, n)
    atime of inode ← NOW ()
    if m = 0 then return END_OF_FILE
    for i from 0 to m – 1 do {
        b ← INODE_NUMBER_TO_BLOCK (i, inode_number)
        COPY (b, buf, MINIMUM (m – i, BLOCKSIZE))
        i ← i + MINIMUM (m – i, BLOCKSIZE)
    file_table[file_index].cursor ← cursor + m
    return m
```

### b) Open

```
procedure OPEN (character string filename, flags, mode)
    inode_number ← PATH_TO_INODE_NUMBER (filename, wd)
    if inode_number = FAILURE and flags = O_CREATE then      // Create the file?
        inode_number ← CREATE (filename, mode)                // Yes, create it.
    if inode_number = FAILURE then
        return FAILURE
    inode ← INODE_NUMBER_TO_INODE (inode_number)
    if PERMITTED (inode, flags) then    // Does this user have the required permissions
        file_index ← INSERT (file_table, inode_number)
        fd ← FIND_UNUSED_ENTRY (fd_table)    // Find entry in file descriptor table
        fd_table[fd] ← file_index            // Record file index for file descriptor
        return fd                            // Return fd
    else return FAILURE                      // No, return a failure
```

6. Timeline

a) Open & read: open("/foo/bar", O_RDONLY)

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[2] |
|---|---|---|---|---|---|---|---|---|---|---|
| open(bar) | | | read | read | read | read | read | | | |
| read() | | | | | read<br>write | | | read | | |
| read() | | | | | read<br>write | | | | read | |
| read() | | | | | read<br>write | | | | | read |

b) Create

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[2] |
|---|---|---|---|---|---|---|---|---|---|---|
| create (/foo/bar) | | read<br>write | read | read<br>write | read<br>write | read | read<br>write | | | |
| write() | read<br>write | | | | read<br>write | | | write | | |
| write() | read<br>write | | | | read<br>write | | | | write | |
| write() | read<br>write | | | | read<br>write | | | | | write |

7. Writing order

a) 三个写操作：allocate new blocks, write new data, update size

b) 先做 allocate new blocks，如果这时候断点，会永远丢失一个 block

先做 write new data，可能会存在两个文件同时申请一个 block 同时写(小影响)

先做 update size，如果这时候断点导致文件 size 不会，可能会读到乱码(更改 inode 以后就对外可见了)

c) 解决方法：日志恢复，fsck(file system check)对比 bitmap 和 inode 是否对应

d) Delete after open but before close：直到所有 process 调用 close 才真正删除，close 和 delete 的时候要检查 refcnt

8. Sync

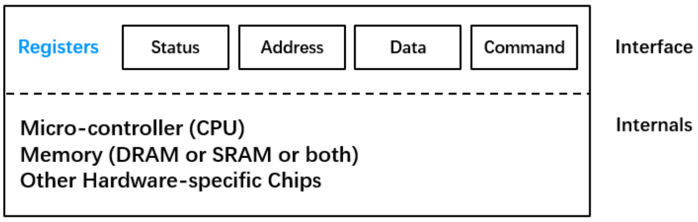
Block cache

- Cache of recently used disk blocks
- Read from disk if cache miss
- Delay the writes for batching
- Improve performance
- Problem: may cause inconsistency if fail before write

# Lec4 disk IO

1. Write 和 close 的步骤
    a) Write：allocate new blocks -> update inode's size and mtime(modify)
    b) Close: free entry in fd_table -> decrease refcnt in file_table; free entry in file_table if counter = 0

2. Canonical IO protocol 经典的协议




```
While(STATUS == BUSY)
   ; //wait until device is not busy
Write data to DATA register and address to ADDRESS register
Write command to COMMAND register
   (Doing so starts the device and executes the command )
While(STATUS == BUSY)
    ; //wait until device is done with your request
```

    a) CPU 一直 polling 到 devive 准备好接收指令
    b) When main CPU is involved with the data movement -> programmed IO(PIO)
    c) OS 空转等待 device 返回

    问题：polling 太浪费 CPU，解决：使用 interrupt

3. Interrupt

    - Instead of polling, the OS can issue a request, put the calling process to sleep, and context switch to another task
    - When the device finishes, it will raise a hardware interrupt
    - The CPU jumps into the OS at a pre-determined **interrupt service routine (ISR)** or more simply an **interrupt handler**
    - The handler is just a piece of OS code that will finish the request

    问题：interrupt 会引起 livelock
    解决：hybrid

    - Default using interrupts
    - When an interrupt happens, handle it and polling for a while to solve subsequence requests
    - If no further request or time-out, fall back to interrupt again
    - Used in Linux network driver with the name **NAPI** (New API)

    优化：interrupt coalescing 合并

    - A device which needs to raise an interrupt first waits for a bit before delivering the interrupt to the CPU
    - While waiting, other requests may soon complete, and thus multiple interrupts can be **merged** into a single interrupt delivery, thus lowering the overhead of interrupt processing

4. DMA

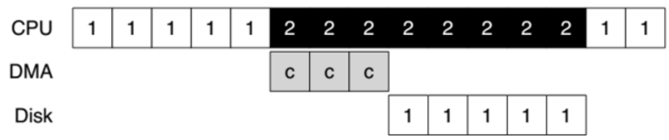
a) 直接访问 memory 的原理

DMA 直接内存存取原理是指外部设备不通过 CPU 而直接与系统内存交换数据的接口技术。把外设的数据读入内存或把内存的数据传送到外设，一般都要通过 CPU 控制完成，如 CPU 程序查询或中断方式。利用中断进行数据传送，可以大大提高 CPU 的利用率。

但是采用中断传送有它的缺点，对于一个高速 I/O 设备，以及批量交换数据的情况，只能采用 DMA 方式，才能解决效率和速度问题。DMA 在外设与内存间直接进行数据交换，而不通过 CPU，这样数据传送的速度就取决于存储器和外设的工作速度。

通常系统的总线是由 CPU 管理的。在 DMA 方式时，就希望 CPU 把这些总线让出来，即 CPU 连到这些总线上的线处于第三态--高阻状态，而由 DMA 控制器接管，控制传送的字节数，判断 DMA 是否结束，以及发出 DMA 结束信号。DMA 控制器必须有以下功能：

i. 能向 CPU 发出系统保持（HOLD）信号，提出总线接管请求；
ii. 当 CPU 发出允许接管信号后，负责对总线的控制，进入 DMA 方式；
iii. 能对存储器寻址及能修改地址指针，实现对内存的读写操作；
iv. 能决定本次 DMA 传送的字节数，判断 DMA 传送是否结束
v. 发出 DMA 结束信号，使 CPU 恢复正常工作状态。




With DMA

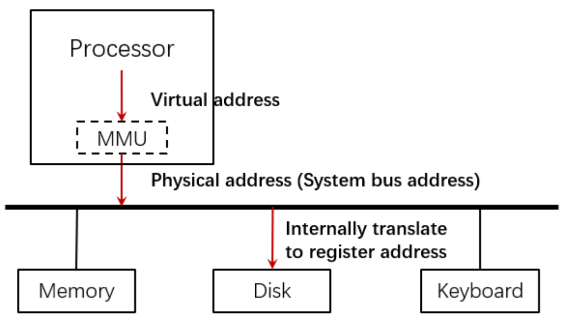
b) 优点：

– Relieve the CPU’s load to execute other program
– Reduce one transfer (original two)
– Take better advantage of long message if the bus supports
– Amortize the overhead of the bus protocol

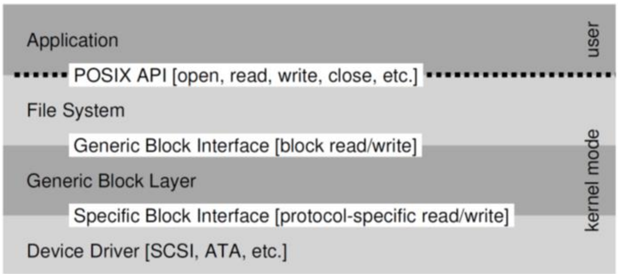
5. Methods of Device Interaction

a) PIO through IO instruction:
   i. in, out instruction
   ii. 必须在 kernel mode 执行

b) Memory-mapped IO:
   i. using LOAD and STORE
   ii. 也可以在 user mode 执行



# Lec5 bus & naming scheme




1. Case study: IDE disk driver
   a) Wait for drive to be ready (0x1F7)
      i. Read status register until READY and not BUSY
   b) Write parameters to command registers (0x1f2 – 0x1f6)
      i. Sector count, logical block address(LBA), driver numebr
   c) Start the IO (0x1F7)
      i. Write READ/WRITE command to command register
   d) Data transfer (for write)
      i. Wait drive status REDAY and DRQ, write data to data port
   e) Handle interrupts
   f) Error handling
      i. After operation, read status register, if ERROR, read error reg

```
Address 0x1F0 = Data Port
Address 0x1F1 = Error
Address 0x1F2 = Sector Count
Address 0x1F3 = LBA low byte
Address 0x1F4 = LBA mid byte
Address 0x1F5 = LBA hi byte
Address 0x1F6 = 1B1D TOP4LBA: B=LBA,
D=drive
Address 0x1F7 = Command/status
```

```
Status
Register
(0x1F7):
  7: BUSY
  6: READY
  5: FAULT
  4: SEEK
  3: DRQ
  2: CORR
  1: IDDEX
  0: ERROR
```

```
Error
Register
(0x1F1):
  7: BBK
  6: UNC
  5: MC
  4: IDNF
  3: MCR
  2: ABRT
  1: TONF
  0: AMNF
```

```
BBK  = Bad Block
UNC  = Uncorrectable data error
MC   = Media Changed
IDNF = ID mark Not Found
MCR  = Media Change Requested
ABRT = Command aborted
TONF = Track 0 Not Found
AMNF = Address Mark Not Found
```

2. Process of disk write and read

**Q:** What is the process of disk write?

- CPU waits for disk to be ready (ide_wait_ready)
- CPU sends **WRITE** request to disk (which block to write? LBA)
- CPU transfers data to the disk (`outsl(0x1f0, b->data, 512/4)`)
- CPU waits (CPU switches to another thread)
- …
- Disk finishes write (data maybe in disk's buffer)
- Disk sends an interrupt to CPU
- CPU wakes up the waiting thread

**Q:** What is the process of disk read?

- CPU waits for disk to be ready (ide_wait_ready)
- CPU sends **READ** request to disk (which block to read? LBA)
- CPU waits (CPU switches to another thread)
- … (can be a long time)
- Disk finishes read (data now in disk's buffer)
- Disk sends an interrupt to CPU
- CPU reads data from disk's buffer to memory (`insl(0x1f0, b->data, 512/4)`)
- CPU wakes up the waiting thread

3. Bus : a hardware layer

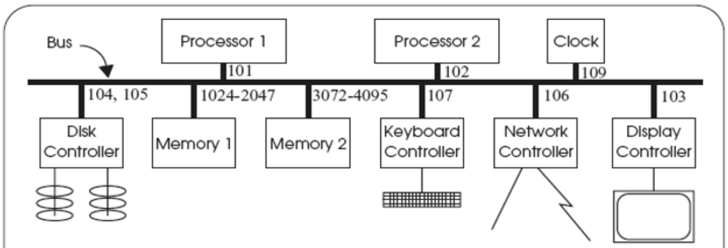



**Figure 2–16:** A computer with several modules connected by a shared bus. The numbers are the bus addresses to which the attached module responds.

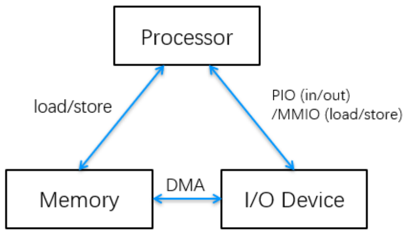
a) Features 特征：a set od wires, broadcast link, bus arbitration protocol
b) Sample: LOAD 1742， R1
   i. Process #2 -> all bus modules {1742, READ, 102}
   ii. Memory 1 recognize, acknowledge and processor 2 releases the bs
   iii. Memory 1 get the value, value <- READ(1742)
   iv. Memory 1 -> all bus modules {102, value}
   v. Processor 2 copy the data to register R1, ack, and memory 1 release
c) Sync data transfer 和 async data transfer
   i. Sync：source 和 destination cooperate through shared clock
   ii. Async：… through explicit signal line
d) DMA on bus

```
bus address    control register
    121        sector_number
    122        DMA_start_address
    123        DMA_count
    124        control

R1 ← 11742; R2 ← 3328; R3 ← 256; R4 ← 1;
STORE 121,R1                    // set sector number
STORE 122,R2                    // set memory address register
STORE 123,R3                    // set byte count
STORE 124,R4                    // start disk controller running

disk controller #1 ⇒ all bus modules:        {3328, block[1]}
disk controller #1 ⇒ all bus modules:        {3336, block[2]}
etc . . .
```

4. Summary

CPU interacts with physical memory
- Through system bus that connects each other
- Using physical address to name memory content

CPU interacts with a device
- Also using physical address (aka., bus address)
- Polling, interrupt and DMA
- I/O instruction (PIO) or MMIO




a) 物理地址是怎么 assign 的？
   i. Memory physical address 用 BIOS，像 keyboard 之类的 device 一直在 fixed，其他的 device 由 OS assign

5. Naming scheme
a) 组成：set of possible names, set of possible values, look-up algorithm
b) 上下文
   i. Context 和 name 分离：inode number 的上下文是 file system
   ii. Context 是 name 的一部分：邮件地址
   iii. 只有唯一可能 context 的 name space 被称为 universal name spaces：例如信用卡号，UUID，email address

cse@sjtu.edu.cn:
- Name = "cse"
- Context = "sjtu.edu.cn"

/ipads.se.sjtu.edu.cn/courses/cse/README :
- Name = "README"
- Context = "/ipads.se.sjtu.edu.cn/courses/cse"

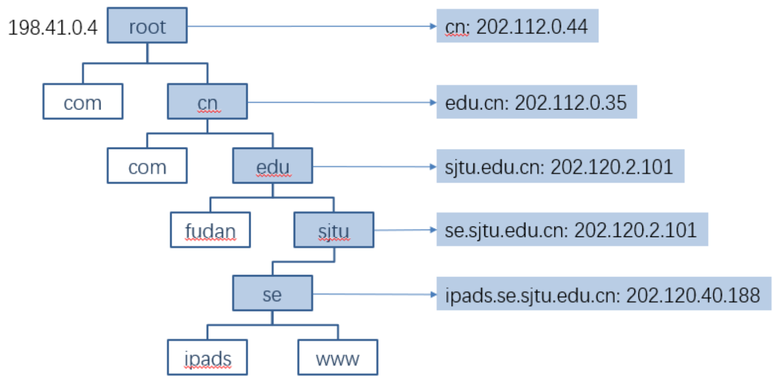
Unix cmd: "**rm foo**":
- Name = "foo", context is current dir
- **Question**: how to find the binary of "rm" command?

Read memory 0x7c911109:
- Name = "0x7c911109",
- Context is thread's address space

# Lec6 DNS

1. The design of DNS
   a) Name 是域名，value 是 IP，look-up algorithm 是 DNS
   b) 不直接用 IP 的原因：不能自由选择 ip，不够 user-friendly
   c) 对应关系：IP 和域名都可以一对多，对应关系也可以改变
      i. 域名可以选择最近的 IP，可以进行负载均衡；IP 可以有多个域名做保险
      ii. 域名和 IP 的对应关系改变通常对 client 不可见

2. Look-up algorithm
   a) Name server 的根节点根节点是 ICANN 管理的（不盈利组织）
   b) 自低向上一层层询问查找的算法：delegation




   c) 容错性：通常每个域都有多台 name server，delegation 算法也会给出可用的 list
   d) 性能调优
      i. 不一定每一次都要去 root 结点，initial DNS request can go any name server，在 SJTU 的机器就从 SJTU name server 开始查找
      ii. Recursive 递归查找，通常性能被非递归好
      iii. Caching，相关参数 TTL(time to live)，TTL 大小的选择是 trade-off，24h 是折中的数字
   e) 其他性能
      i. 至少两台一样的 replica server
      ii. Organization's name server: SJTU 多台在校内，至少一台在校外
   f) 对比：hostname 和 filename

- They are both for more **user friendly**
  - File name -> inode number
  - Hostname -> IP address
  - The file name and hostname are **hierarchical**; inode num and IP address are **plane**
- They are both **not** a part of the object
  - File name is not a part of a file (stored in directory)
  - Hostname is not a part of a website (stored on name server)
- Name and value binding
  - File: 1-name -> N-values (no); N-name -> 1-value (yes)
  - DNS: 1-name -> N-values (**yes**); N-name -> 1-value (yes)

3. Behind the DNS design
   a) 层次结构设计的优点 hierarchical design
      - Hierarchies delegate responsibility
      - Each zone is only responsible for a small portion
      - Hierarchies also limit interaction between modules
   b) DNS 设计的优点
      i. Global name
      ii. Scalable in performance: simplicity(easy for PC), caching, delegation
      iii. Scalable management: hierarchy，每个 zone 都有各自的 policy
      iv. Fault tolerant
   c) DNS 设计的问题
      i. Policy 谁来管理 root zone
      ii. Significant load on root server：负载过大，访问不存在域名造成 DoS 攻击
      iii. Security ：谁能改变域名和 IP 之间的 bind
   d) DNS 攻击

4. Some problems

A. To answer your query, M must contact one of the root name servers.

B. If M answered a query for www.cslab.scholarly.edu in the past, then it can answer your query without asking any other name server.

C. M must contact one of the name servers for cslab.scholarly.edu to resolve the domain name.

D. If M has the current Internet address of a working name server for scholarly.edu cached, then that name server will be able to directly provide an answer.

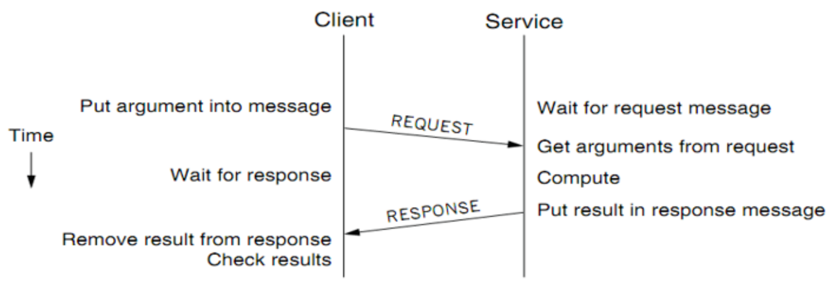
E. If M has the current Internet address of a working name server for cslab.scholarly.edu cached, then that name server will be able to directly provide an answer.

*Ex. 4.5*
A is true only when M's cache misses *and* the prefix of the name has nothing in common with the domain that M serves.
B will be true only if the time-to-live of the DNS record cached by *M* has not expired. If the binding found by M in its cache for www.cslab.scholarly.edu has expired, then M must contact at least one other name server to resolve the domain name.
C is true only if M has no valid information in its cache for www.cslab.scholarly.edu.

D is not usually true, because the scholarly.edu name service will normally refer *M* to the cslab.scholarly.edu name service, rather than answer the query itself. (Though it is possible to configure a name service to directly contain the data for one or more of its subdomains.)
E will normally be true, though it is possible that the name is a synonym that requires cslab.scholarly.edu to ask yet another name server for help.

# Lec7 RPC & NFS

1. Function call 可能产生的问题
    - Errors in callee may corrupt the caller's stack
        - The caller may later compute incorrect results or fails
    - Callee may return somewhere else by mistake
        - The caller may lose control completely and fail
    - Callee may not store return value in R0
        - The caller may read error value and compute incorrectly
    - Caller may not save temp registers before the call
        - The callee may change the registers and causes the caller incorrect
    - Callee may have disasters (e.g. divided by 0)
        - Caller may terminate too, known as fate sharing
    - Callee may change global variables that it shouldn't change
        - Caller and callee may compute incorrectly or fail altogether
        - Even other procedures may be affected
    - Procedure call contract provides only soft modularity
        - Attained through specifications
        - Cannot force interactions among modules to their defined interfaces
2. C&S 架构




3. RPC (Remote procedure call)

Client program

```
procedure MEASURE (func)
    SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
    response ← RECEIVE_MESSAGE (NameForClient)
    start ← CONVERT2INTERNAL (response)
    func ()        // invoke the function
    SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
    response ← RECEIVE_MESSAGE (NameForClient)
    end ← CONVERT2INTERNAL (response)
    return end – start
```

```
procedure TIME_SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE (NameForTimeService)
        opcode ← GET_OPCODE (request)
        unit ← CONVERT2INTERNAL(GET_ARGUMENT (request))
        if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
            time ← CONVERT_TO_UNITS (CLOCK, unit)
            response ← {"OK", CONVERT2EXTERNAL (time)}
        else
            response ←{"Bad request"}
        SEND_MESSAGE (NameForClient, response)
```

a) Client stub:
   i. put args into request; send request to server; wait for response
b) service stub
   i. wait for messag; get paras from request; call the produce(get_time)
      put resule into response; send response to client
c) message 的组成
   service ID, service parameter, using marshal/unmarshal
d) Marshal 和 unmarshal
   Marshal
   – Convert an object into **an array of bytes** with enough annotation so that the unmarshall procedure can convert it back into an object
e) RPC system component
   - 1. Standards for wire format of RPC message and data types
   - 2. Library of routines to marshal / unmarshal data
   - 3. Stub generator, or RPC compiler, to produce "stubs"
     – For client: marshal arguments, call, wait, unmarshal reply
     – For server: unmarshal arguments, call real function, marshal reply
   - 4. Server framework:
     – Dispatch each call message to correct server stub
   - 5. Client framework:
     – Give each reply to correct waiting thread / callback
   - 6. Binding: how does client find the right server?
f) Client framework
   - Keeps track of outstanding requests
     – For each, xid and caller's thread / callback
   - Matches replies to caller
   - Might be multiple callers using one socket
   - Usually handles **timing out** and retransmitting
g) Server framework
   - Type-1: Create a new thread per request
     – Master thread reads socket[s]
   - Type-2: Use a fixed pool of threads
     – Use a queue if too many requests
     – E.g., NFS server
   - Type-3: Just one thread for **serial execution**
     – Simplifies concurrency, e.g., the X server

4. RPC != PC
   a) RPC 能 redure fate sharing，但会引入 no response 的新问题，会占用更多时间
   b) RPC 的 failure handling: client 希望发送 at least once, server 希望发送 at most once，就要求发送 exactly once

c) 其他问题：language support(有一些语言不能很好的使用 RPC，全局变量，指针)
Security consideration

d) 为没什么要用 RPC 和 C&S 架构？
- Programmers make mistakes
- Mistakes propagate easily
- Enforce modularity

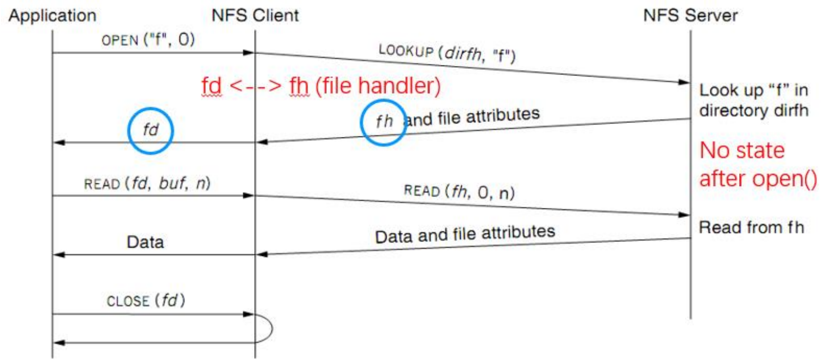
e) RPC 做了什么？有替代方法吗？
- E.g., socket? HTTP? Why not use them?
- Be more friendly for programmers

## 5. NFS(Network file system)

- Transparency: compatibility with existing applications
- OS independent: clients even in DOS
- Easy to deploy

a) 目的： `# mount -t nfs 10.131.250.6:/nfs/dir /mnt/nfs/dir` 和现有的 app 通用

b) Overview




c) Case 1 : rename after open

| | Program 1 on client 1 | Program 2 on client 2 |
|---|---|---|
| 1 | CHDIR ("dir1") | |
| 2 | fd ← OPEN ("f", READONLY) | |
| 3 | | RENAME ("dir1", "dir2") |
| 4 | | RENAME ("dir3", "dir1") |
| 5 | READ (fd, buf, n) | |

Time ↓

UNIX spec: program 1 should read "dir2/f", NFS should keep the spec

d) Case 2 : delete after open

| | Program 1 on client 1 | Program 2 on client 2 |
|---|---|---|
| 1 | | fd ← OPEN ("f", READONLY) |
| 2 | UNLINK ("f") | |
| 3 | fd ← OPEN ("f", CREATE) | |
| 4 | | READ (fd, buf, n) |

Time ↓

UNIX spec: on local FS, program 2 will read the old file

## File Handler for a Client

- File handler contains three parts
  - **File system identifier**: for server to identify the file system
  - **inode number**: for server to locate the file
  - **Generation number**: for server to maintain consistency of a file

- Can still work across server failures
  - E.g., server reboot
- **Q**: Why not put path name in the handle?

## Stateless on NFS server

- Stateless on NFS server
  - Each RPC contains all the information
- **Q**: What about states like file cursor?
  - Client maintains the states, including the file cursor
- Client can repeat a request until it receives a reply (at least once)
  - Server may execute the same request twice
  - Solution: each RPC is tagged with a transaction number, and server maintains some "soft" state: reply cache
  - **Q**: What if the server fails between two same requests?

## Cache on the Client

- NFS client maintains various caches
  - Stores a **vnode** for every open file
    - Know the file handles
  - Recently used vnodes, attributes, recently used blocks, mapping from path name to vnode
- Cache benefits
  - Reduce latency
  - Less RPC, reduce load on server
- **Cache coherence** is needed

## Coherence

- **Read/write coherence**
  - On local file system, **READ** gets newest data
  - On NFS, client has cache
  - NFS could guarantee read/write coherence for every operation, or just for certain operation
- **Close-to-open consistency**
  - Higher data rate
  - **GETATTR** when **OPEN**, to get last modification time
  - Compare the time with its cache
  - When **CLOSE**, send cached writes to the server