

## OS Includes:

a program  $\Rightarrow$  “kernel”

1. Manage all physical devices such as CPU, RAM, hard disk etc.
2. It exposes some functions as **system calls** for configuring the kernel or building things on-top.

## Some more programs

1. **drivers**: handle the interaction between kernel and external devices.
2. **shell**: renders a simple command-line user interface with a full set of commands.

**Some optional programs**: GUI, Browser, Paintbrush

**Process** is an execution instance of a program. It's not just a program: It also has states concerning the execution. e.g., which line of code is running; how much time is left before giving the CPU to another process.

**Process-Related Tools**: `ps` & `top`: `ps` reports information about every process in the system.

**Shell** e.g., `bash` in Linux

**Steps to execute a command**: 1. Get input ( `getchar()` ) 2. Syntax checking 3. `fork()` to create a new process 4. `exec()` to execute the program

**Process Hierarchy**: Forms a tree hierarchy.

**System Call** 1. Is a function call. 2. Exposed by the kernel. 3. Abstracts away most low-level details.

**Categorizing System Calls**: 1. Process Management 2. File System Management 3. Memory Management 4. Security 5. Device Management 6. Information Maintenance

**Note**: Functions like `printf` from the C library are **not** system calls.

**Examples of system calls**: `mkdir`, `rmdir`, `chown`, `chmod`, `open`, `close`, `brk`, `mmap`.

**Execution Context**: System calls execute in **kernel space**, while library functions (like those in the C library) execute in **user space**.

## Modern Computer Architecture

### File system

**Advantages**: sequential access is simple

**Disadvantages**: fragmentation, space management

Leads to: non-sequential access

**Program Counter**: A register that contains the memory address of the next instruction.

### Instruction Cycle:

**Fetch (IF)**: Fetch the next instruction from memory.

**Decode (ID)**: Prepare registers in readiness for the next step.

**Execute (EX)**: Perform the computation or compute the memory address.

**Memory Access (MEM)**: Read from or write to memory.

**Write Back (WB)**: Write results back to the register.

**CPU speed = CPU clock rate** e.g.  $1.8\text{GHz} = 1.8 \times 10^9$  clock cycles per second.

**Pipeline**: | Fetch | Decode | Execute | Write Back |

Sequential execution: One instruction finishes before the next one starts.

**Scalar Pipeline**: 上一轮fetch后decode时fetch下一个Cycle

**Superscalar Pipeline**: Builds upon the Scalar pipeline by adding instruction-level parallelism.  $\Rightarrow$  Multiple instruction cycles run in parallel.

# 一列最下没对

## Process (User space)

Process is a program in execution (Contains all accounting information)

It will stop early if sent a *Signal* to interrupt it. Multiple processes can work together to do more complicated tasks.

**Process identification:** Processes have a unique ID number *PID*.

Can be obtained using `getpid()` -> `(int) getpid()`

`getppid()` gets the parent process *PID*.

**Process creation:** `fork()` (fail返回-1) 执行顺序由Scheduler决定.

1. `fork` clones all user-space data and some kernel-space attributes. e.g., Program Counter, Program code, Memory, Opened files.
2. `fork`后, 父进程 accumulated run time ; 子进程从0开始run time
3. File locks 父进程不变, 子进程 None

**Program execution:** `exec*()`

`execl(PATH, char arg0, ..., NULL)` e.g., `execl("/bin/ls", "ls", NULL)`

可多次调用 `exe` 让一个 process 执行多个 program

1. It will **replace** user-space info: Program Code, Memory, Register values.
2. kernel info is **preserved**: PID, process relationships.\*

`fork() + exec*() + wait() = system()`

**Environment variables:** `main()` 函数最后会有一个\*\*`envp`是字符串数组存环境变量

变量名	描述	变量名	描述
<b>SHELL</b>	The path to the shell that you're using.	<b>HOME</b>	The full path to your home directory.
<b>PWD</b>	The full path to the directory that you're currently on.	<b>USER</b>	login name.
<b>EDITOR</b>	Default text editor	<b>PRINTER</b>	Default printer

类似python字典 `getenv("HOME")`;能得到返回值

`wait()`: Process using wait will be suspended until: 1. one of **child process** running terminated or 2. Signal received

立即**Return**如果无子进程或之前有终止的子进程(此时返回的是-1)

`waitpid(pid_t pid, int *status, int options)`

`pid: (+)`: 特定的子进程 `0`: Wait for any child in the same process group (调用同组) `-1`: 等待任意, 相当于`wait()`

## Process (Kernel Space)

运行中的 `fork()`:

(**Kernel**) **Update:** PID, Runtime = 0, Pointer to parent **Preserved:** arrays of opened FILES(0 stdin, 1 stdout, 2 stderr, 后面是打开的文件)

(**User**) **Copy:** Local variable, Global variable, Dynamically-allocated memory, Code + Constants

运行中的 `exec*()`:

(**Kernel**): reset the register values

(**User**): **Cleared** Local variable, Dynamically allocated memory, **Reset** Global variable, Code + Constants

`exit()`:

1. kernel free all the allocated memory & all on the user-space memory

**However:** **PID still in kernel** => **Zombie process** ( 可查 ) **Why need Zombie?** allow to read exit status

2. kernel sends **SIGCHLD** to parents. => help parents to deal with Zombie process. (用 `wait` 会在 **SIGCHLD** 后彻底清除子进程)

`exit()` **system call turns a process into a zombie when...**

1. The process calls `exit()`

2. The process returns from `main()`

3. The process terminates abnormally

**Why important to know above?**

1. process execution/suspension

2. System resource management Zombie占PID, and PID 共 32,768

**First process** => **init: PID=1 task** => **create more processes**

`init()` (`fork()` & `exec()`) -> SSH Server (`f&e`) -> shell (`f&e`) -> `top`

**Orphan process:** if parent terminate but child has started => hierarchy changes from tree to forest, and no one would know the termination of the orphan

=> 解决方法 (Solution):

Linux: re-parent to **init** => (在 `exit()` 过程中完成) Windows: 保持森林 (Maintain forest)

## 第三列有一个图片没写

**Context switching (上下文切换):** is switching procedure from one process to another.

**when?**

1. Process goes to blocking/waiting state {`wait()`, `sleep()`}.

2. A POSIX signal arrives, e.g., **SIGCHLD**. (POSIX 信号到达, 例如 **SIGCHLD**)

3. When OS scheduler says "time's up" (e.g., round-robin) [go to ready].

4. Priority reason. (优先级原因)

**流程 (Procedure):**

1. 出现像 `wait()` 的操作

2. Backup all current context of that process to the kernel-space's PCB.

3. Load the Context of that process from the main memory to CPU.

**问题: Context Switch is expensive (上下文切换是昂贵的)**

**Direct** costs in kernel: Save and restore registers, Switch process address space.

**Indirect** costs: cache misses, increase memory access latency. (增加内存访问延迟)

**User time** VS **System time** -> CPU time on kernel space

↳ CPU time on user-space (不包括等待时间)

**System call is expensive:** 1. Function calls cause overhead on stack. 2. Cause context switch from user-code to kernel-code.

**Real time** < **User** + **kernel**: multi-threading (Spent Core time are overlapping)

**Real time** > **User** + **kernel**: OS seldom schedules CPU to you.

## Signal

**Type 1.** From hardware interrupt / CPU exception(**kernel**), wrapped by the kernel as a POSIX signal - E.g., Ctrl-C -> SIGINT, segmentation fault -> SIGSEGV

**Type 2.** Generated directly from **kernel**. -E.g., from `exit()` -> SIGCHLD

**Type 3.** Generated from one **process** to another. - E.g., `kill 1234` from another process 885

**kill() function:** `int kill(pid_t pid, int sig);` e.g., `kill(getpid(), SIGTERM);`

**signal() function:** 修改信号的Default Handler(默认处理行为)

```
signal(SIGINT, sig_handler) `
void sig_handler(int sig) { if (sig == SIGINT) ... }
```

当我们收到

信号后会继续，除非是 `sleep()`, `pause()`

Signal 的状态是由数组表示的，如 SIGINT 对应的 bit (or mask)

receive SIGINT 第 1 位变 1, handle 后变回 0

**Pause():** 暂停直到收到信号 (Suspend until a signal is received)

**Asynchronous signal:** 不是 process 产生，取决于外部因素 (ctrl-C), 时间不确定

**Synchronous Signal:** 由 process 产生，时间确定

**alarm():** -> asynchronous timing for a process

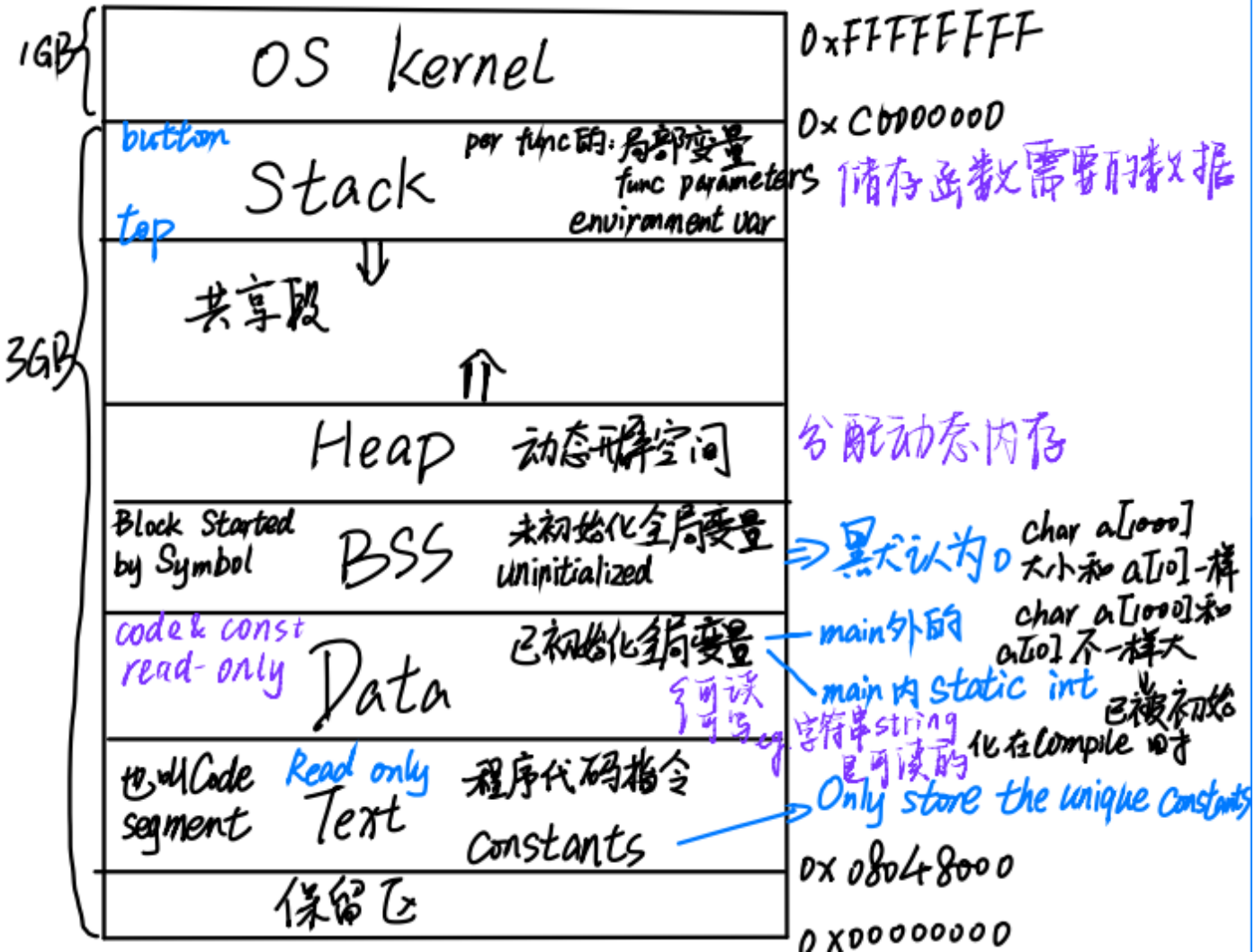
括号里一个数 x, x 秒后，进程中止 (发送 SIGALRM), 变为 Zombie

`alarm(0)` indicates canceling the current timer

**Interval timer** => `setitimer()` 周期性发送信号

**Memory Management**  $KB(2^{10})MB(2^{20})GB(2^{30})TB(2^{40})PB(2^{50})EB(2^{60})$

32-bit system maximum amount memory in a process is  $2^{32}$  bytes



## Stack in-depth

When a function is called - push a **Stack frame** (从stack底到顶, Parameters-return address, local vars)

When a function returns - Pop the **Pop the stack frame**. Set Stackptr = \*frameptr; \*frameptr stores the previous stack ptr.

The compiler hardcodes this mechanism into your program, is not done by the kernel

**Recursions avoid stack overflow**: Minimizing the number of function arguments, local vars, calls. Use global vars, malloc instead, trail recursion

## Heap

每次malloc都会创建一个空间, malloc后发生什么 1. 有一个结构存当前数组大小和link list ptr 2. 紧接着才是数组, 指针指向这里 由于存在之前的结构, 两个相邻数组指针相减大于数组大小

free: 最后就直接shrink, 中间的会存一个专门的free的link list, 头是head, 尾是Null

Segmentation fault

## Threading

For **Multi-threading**, they share the same code, same address space (share global vars), same heap (malloc), different registers and stack (local vars)

**Multi-thread examples**: Old Firefox - Single-process & each tab is a thread, Faster but crash one will crash all.

Chrome - 1 tab 1 process



Process	Thread		Process	Thread
Heavy weight	Light weight	Creation	<code>fork()</code>	<code>pthread_create()</code>
Costly creation	Cheap creation	I.D. Type	PID, an integer	“pthread_t”, a structure
Process can't access the memory area of other processes	Threads share the same memory area	Who am I?	<code>getpid()</code>	<code>pthread_self()</code>
Costly process switching	Cheap thread switching	Termination	<code>exit()</code>	<code>pthread_exit()</code>
One process multiple threads		Wait for child termination	<code>wait()</code> or <code>waitpid()</code>	<code>pthread_join()</code>
Different programs	Same program (different thread functions)	Kill	<code>kill()</code>	<code>pthread_kill()</code>

`pthread_create(&tid, NULL, func_name, func_parameters)` `pthread_join(tid, (void**) &t-output)`

第二个可以是NULL, 这个写法是手从 `pthread_exit()` 返回的值, `t-output`定义时定义的时指针, 输出值用 `*t-output`

**Kernel** is a multi-thread program, it create kernel threads & OS threads

## Scheduling

**CPU - bound process:** user-time > sys-time

**I/O - bound process:** user-time < sys-time

### Classes of scheduling

#### 1. Preemptive scheduling(抢占式调度, 也叫time-sharing)

What is it?	When a process/thread is chosen by the scheduler, the process would have the CPU until... -the process voluntarily waits for I/O, or -the process voluntarily releases the CPU, e.g., <code>exit()</code> , <code>yield()</code> . <b>-particular kinds of interrupts (e.g., periodic clock interrupt, a new process steps in) are detected.</b>
Pros	Good for systems that emphasize <b>interactiveness</b> . - Because every task will receive attentions from the CPU.
Cons	Bad for systems that emphasize the time in finishing tasks.

Evaluation: 1.Number of Context Switches 2. Turnaround Time (`Termination Time` - `Arrival Time`) 3.

Waiting Time

### Algorithms

**1. Non-preemptive SJF:** Arrive 的任务中选最快的, 但存在长期饥饿问题, 一个任务很早就准备好了, 但是后面新来的任务一直比它短(SJF无解)

**2. Preemptive SJF:** 每当出现新任务的时候, 如果这个时间小于当前任务的剩余时间就切换. waiting time和 turnaround time 降低, context switches增多

**3. Round-Robin (RR)**是preemptive的, 所有ready的都在queue中, 每个任务都有一个time quantum, 用完了没结束也换, 并且放到队尾. 解决长期饥饿但别的不占优

**2. Priority Scheduling:** 每个task都有priority, 高priority的先执行\*

Priority分为两种**static priority**和**dynamic priority**, static的就是fixed的, dynamic在每个新任务到来时会变化

**Multi-Level Queue Scheduling:** Each queue can use a different algorithm. Priorities can be adjusted dynamically.

**Thread Scheduling:** Linux >= 2.6之后我们就只关心threads, the scheduler determines which threads get CPU

time.

## Scheduling for modern servers

- Symmetric multiprocessing (SMP)
  - All processes/threads **share a common ready queue**
    - → **Race Condition** 共享一个就绪队列
  - Scheduling:
    - Each processor/core scheduler examines the common ready queue and selects a process to execute
  - Affinity Scheduling:
    - Soft affinity: **attempt** to keep the same process/thread on the same core 线程可申请到靠近它所需数据的处理器。
    - vs. Hard affinity: use `pthread_attr_setaffinity_np()` to manually bind a thread to a specific core in your pThread program
    - Especially important when we are reaching NUMA world

## File Management

**Virtual File System(VFS)**: An abstraction layer on top of concrete file systems

不同的文件系统有不同的读取规则, 比如NTFS(windows), Ext4(Linux), Fat32(U盘), ISO9660(光碟), 这些在kernel space中, 用C lib中的读取是会调用对应的System Call, VFS则会根据文件启用不同的部分应对对应的文件系统

### Library call VS System call

`fopen()`, invokes `open()` and returns a pointer to **FILE**(a structure define in **stdio.h**)

**File**包含一个Buffered I/O, 读到哪的指针, 1个int(file descriptor, `fd`)来描述.

### Buffered I/O

Why need? Reduces the number of system calls

Modes	Read-related call e.g., <code>getchar()</code>	Write-related call e.g., <code>putchar()</code>
Fully-buffered <code>_IOFBF</code>	Data is <b>read in one bulk</b> and is stored in the buffer.  Invoke the <code>read()</code> system call when the buffer becomes <b>empty</b> .	Data is written to the buffer.  Invoke the <code>write()</code> system call when the buffer becomes <b>full</b> , or before the <b>process terminates</b> .
Line-buffered <code>_IOLBF</code>	Data is read into the buffer until the <b>newline character</b> is encountered.	Data is written to the buffer. When a <b>newline character</b> is encounter, <code>write()</code> system call is invoked.
Un-buffered <code>_IONBF</code>	Directly translate every library call into a <code>read()</code> system call.	Directly translate every library call into a <code>write()</code> system call.

### Change the buffer

```
int setvbuf( FILE *stream, char *buf, int mode, size_t size );
```

buf存的是缓冲区地址, size是大小, 如果buf为NULL意味着不用缓冲区(buffer)

`stdin` and `stdout` are line-buffered by default. `stderr` is un-buffered by default.

**EOF**: What is? 不存在于system call, 是定义在stdio.h中的, `fread()`会记住是否到达末尾, 到达了就返回-1(EOF), 如果没到reads data from the buffer or system calls.

**Linux: Everything is file** e.g. : Regular File, Directory, Block special file, Character special file(mouse)

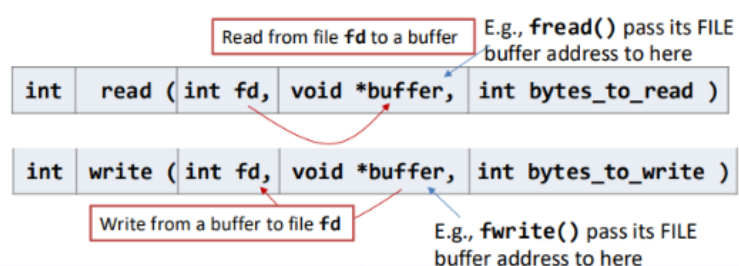
A "File" contains **attributes** and **data**

Common Attributes		FAT32	NTFS	Ext2/3/4	Common Attributes	Way to change them?	
						Command?	Syscall?
Name		✓	✓	✓	Name	mv	rename()
Size		✓	✓	✓	Size	edit it using vi and then save...	write(), truncate(), etc.
Permission	The design of FAT32 does not include any security ingredients.		✓	✓	Permission	chmod	chmod()
Owner			✓	✓	Owner	chown	chown()
Access, creation, modification time		✓	✓	✓	Access, creation, modification time	touch	utime()

**stat**指令能看attributes, 对应的system call有 `stat()`, `fstat()`, `lstat()`

A **directory** is a file consisting of **directory entries**(`dirent`), `dirent` is a struct

```
struct dirent {
    ino_t d_ino; /* unique inode number*/
    off_t d_off; /* offset to the next dirent*/
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type; /* type of file; not supported
                           by all file system types */
    char d_name[256]; /* filename */
};
//可以直接struct dirent * entry
dir = opendir(path); //open
while( (entry = readdir(dir)) != NULL ) // read
    printf("%ld %s\n", (long) entry->d_ino, entry->d_name);
closedir(dir); //close
```



Library calls that eventually invoke the <code>read()</code> system call	Library calls that eventually invoke the <code>write()</code> system call
<code>scanf()</code> , <code>fscanf()</code>	<code>printf()</code> , <code>fprintf()</code>
<code>getchar()</code> , <code>fgetc()</code>	<code>putchar()</code> , <code>fputc()</code>
<code>gets()</code> , <code>fgets()</code>	<code>puts()</code> , <code>fputs()</code>
<code>fread()</code>	<code>fwrite()</code>

**read()** 的流程:

S1: 看是否 the end of the file is reached or not. Comparing size and file seek.

S2: Reading data

S3: File data is stored in a fixed size cache in the kernel.

S4: Write data to the userspace designated buffer.

**write()** 的流程:

S1: Copy data from user-space buffer to kernel buffer.

S2: 根据data length, (1) change in file size, if any, and (2) change in the file seek.

S3: The call returns.

S4: The buffered data will be flushed to the disk from time to time.

**The kernel buffer cache implies:**

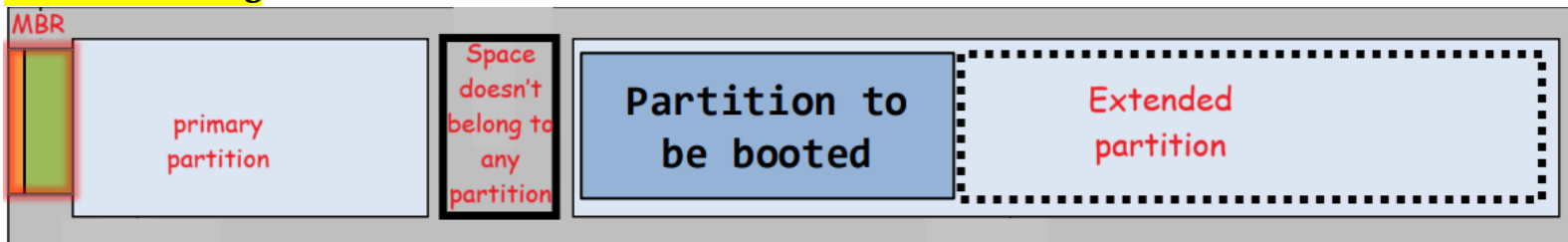
1. Improving reading & writing performance

2. Why not to press reset button: Sudden reset loses cached data not yet written to disk, potentially corrupting file systems.

3. Why to "eject" USB drives: Ejecting ensures all cached data is written to the device, preventing data loss or corruption.



## Disk and booting



### Why do we need to have partitions?

1. Multi-booting, 一个hard disk上可以有多个启动程序(windows, macOS)
2. Data management. 可以分很多个logic drive, 每个存不同的东西(C, D, E盘)
3. Backup and Maintenance. Partitions 是独立的 & 可支持不同的file systems

### HDD(Hard disk)

disk由盘片(platter)组成, 每个有2个surface, 中央有个spindle. 每个surface被划分成了很多个track, track被切成了很多个小部分(sector). 每个sector包含了相等数量的数据位, 中间有gap(标识sector的格式化位).

Disk用read/write head 来读写, 其连接到actuator arm

**CHS**寻址方法, Cylinder\Heads\Sector, 注意每个platter有两面, head数量为platter两倍.

**LBA**寻址方法, 把disk视作连续的blocks, 每个block有对应的编号, 并且字节数固定

### SSD

### Booting(启动)

Legacy boot流程: 1. BIOS(store in EPROM) locates the first bootable device (SSD/USB) 2. Executes its boot code (in MBR) 3. Boot code executes the latter boot loader **现在开始用UEFI**

**MBR(Master boot record)**, stores **Boot code**(橘色部分, 前446bytes) & **partition table**(最多4个, 绿色部分, 后64字节), 结束标记magic number **0xaa55** 2 bytes, 共512 bytes 如今在替换为 **GUID Partition Table (GPT)**

**Boot code**: execute a **boot loader** in a bootable partition(Windows的C:\boot.ini或Linux的GRUB)

**Boot loader**: locate one **kernel image** and boot (bring it to memory & execute) it.

**Kernel image**: is just a file (Linux: /boot/vmlinuz C:\windows\System32\ntoskrnl.exe), when it is found the kernel start. It initializes all kernel subsystems. (initialize memory layout, initialize drivers, etc.)

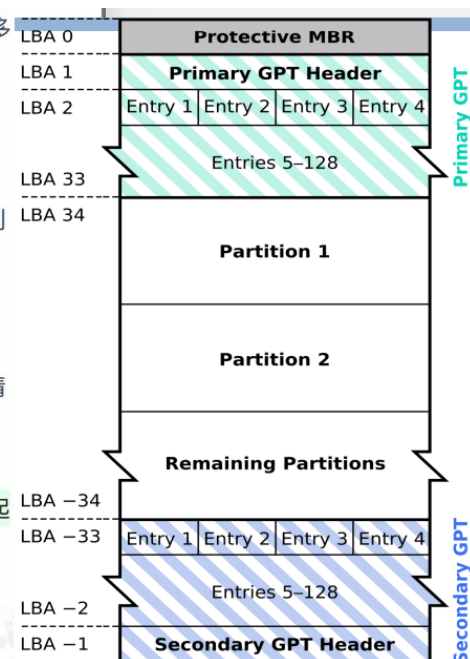
**Partition table**: 里面每个项16个字节, 分为Bootable flag(1, 是否bootable 0x80), 起始CHS(3), Partition type(1, ext4\FAT), 结束CHS(3), 起始LBA(4), Sizes in sectors(4) 这决定了每个分区最大2T

Partition分为Primary和extended, extended只有一个, extended可以划分成很多个logical partition

### GUID Partition Table (GPT)

**GPT (GUID分区表)** : 与MBR相比, GPT是一个更现代的分区方案, 它支持大于2TB的磁盘, 并且可以创建多达128个分区(在某些操作系统中可以更多)。GPT位于磁盘的开始和结束部分, 提供冗余, 以防主GPT损坏。它使用 **全局唯一标识符 (GUID)** 来标识分区, 这意味着每个分区的标识符在全世界范围内都是唯一的。大多数现代操作系统(例如Windows 10、Linux、macOS)都支持GPT。但是, 使用GPT可能需要UEFI固件, 而不是传统的BIOS。

- **保护MBR**: 位于磁盘的最开始, 即第一个扇区 (LBA 0)。保护MBR的目的是使不支持GPT的旧系统识别磁盘, 避免这些系统意外修改GPT磁盘。保护MBR占用1个扇区。
- **GPT头部**: 磁盘的第二个扇区 (LBA 1)。包含有关GPT分区表的元数据, 如其位置、大小和CRC校验码。GPT头部通常也只占用1个扇区。
- **分区条目数组** (即GPT分区表): 跟在GPT头部之后, 这是一系列分区条目, 每个条目128字节。默认情况下, GPT为分区条目数组预留了128个条目, 每个条目128字节, 因此默认情况下分区表占用  $128 \times 128$  字节 = 16KB。对于512字节扇区的磁盘, 这将是32个扇区。
  - 每个分区条目128字节包含以下字段: **分区类型GUID (16字节)**、**唯一分区GUID (16字节)**、**起始LBA (8字节)**、**结束LBA (8字节)**、**属性标志 (8字节)**、**分区名称 (72字节)**
  - 这种分区表属性决定了 每个分区最大容量是 (  $2^{64} + 512$  ) Bytes。
- **GPT备份**: 在磁盘的最后面有一组GPT的备份



**属性标志 = Attribute Flags**: 描述分区特性的标志位, 如是否可启动、是否为隐藏分区等。分区名称 = **Partition Name**, 分区的人类可读名称, 通常使用Unicode字符存储。分区类型GUID = **Partition Type GUID**(Fat/Ext4)

# Mounting 没看

**File System**: A way that **lays out** how data is organized on a storage device

Layout method

## 1. Contiguous allocation

每个Partition的开头有一个Root Directory, 里面存了Filename, Starting address, Size. File deletion is easy, updating root directory. Bad for file creation. 然而即便剩空间大但依然可能放不下新的(**External Fragmentation**)  $\Rightarrow$  Defragmentation process, 拼到一起紧凑一点, 但文件变大时就很麻烦(**Growth problem**)

Application: ISO 9660, CD-ROM

**2. Linked allocation** S1: Chop the storage device into equal-sized blocks. S2: Fill the empty space in a block-by-block manner.

For each block, leave 4 bytes as the “pointer”, 最后一个写-1(NULL), Root Directory 写的是, file name, 1st Block Address, Size.

解决External fragmentation和File Growth problem

存在的问题 **1. Internal Fragmentation**  $\Rightarrow$  Last Block 可能没 fully filled. **2. Poor random access performance**, 如果我要访问File的19th block中的内容, 我需要从头一个一个读

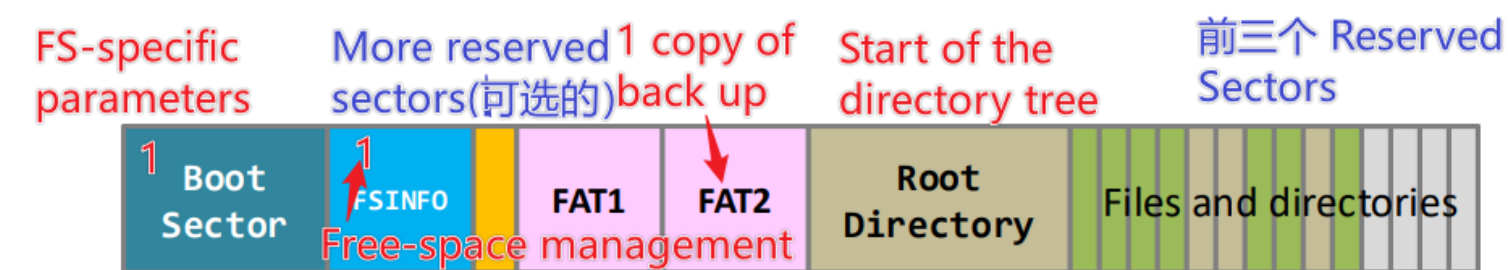
Application: **FAT (File Allocation Table)** 用在: CF cards, SD cards, USB drives

在之前的Linked allocation方法中, 每个block前4byte存后面的位置, FAT则是集中起来存到一个table里, 记录了每个Block的下一个Block的地址. 保存(部分)FAT

在kernel cache中。

Start from floppy disk and DOS, Dos中每个block被称为cluster, FAT xx表示xx-bit cluster address也就是说总共 $2^{xx}$ 个blocks, FAT32只有28, MS reserves 4bits

File System Size计算方法: Cluster Size  $\times$  Cluster address



Root Directory从Cluster #2开始做directory traversal

**Directory Entry**: 每个占32 bytes, 用来描述当前directory下包含哪些文件和sub-directory. 字节0, 文件名的第一个字符(0x00或0xE5表示未分配), 1-10表示文件剩余的部分+扩展名(8+3, 8个字符文件名+3字符扩展名). 字节 11: 文件属性(只读\隐藏) 字节12: 保留字节 字节13-19: Creation and access time information. 字节20-21: High 2 bytes of the first cluster number (0 for FAT16 and FAT12). 22-25 Written time information. 26-27 Low 2 bytes of first cluster number. 28-31 File size(最大4G-1 Bytes, 主要用来决定最后一个Block读多少)

**找文件流程**: 在directory里面找First Cluster, 然后从FAT里一直读, 知道读到最后一个. **如果要写**, 读取FSINFO, 这里面存了下一个空闲的Cluster的位置, 写完更新FSINFO. 如果要**如果要删**, 更新FSINFO和FATS(改为0), 把directory entry的1st bytes 改为0x00

**取消删除算法**: Scan directory structure for entries with first byte 0xE5, restore original filename, extract file size (bytes 28-31) and first cluster number (bytes 20-21 and 26-27). For small files (single cluster), directly read data from first cluster; for large files (multiple clusters), rebuild cluster chain. When rebuilding, check FAT table status of first cluster; if cleared, assume contiguous allocation and read consecutive clusters until reaching file size or verify data validity using file signatures/magic numbers analysis.

## 3. Inode allocation

每个File directory都有一个独特的inode

EXT2/3/4: 由很多个group组成

For Ext2 & Ext3: Block size: 1,024, 2,048, or 4,096 bytes. Block address size: 4 bytes **Max file size** = Block size × Block address size

COMPONENT	DESCRIPTION
Superblock	Stores FS specific data. E.g., the total number of blocks, 、
GDT (Group Descriptor Table)	It stores:- The locations of the block bitmap, the inode bitmap, and the inode table.- Free block count, free inode count, etc...
Block Bitmap	A bit string that represents if a block is allocated or not.
Inode Bitmap	A bit string that represents if an inode (index-node) is allocated or not.
Inode Table	An array of inodes ordered by the inode #.
Data Blocks	An array of blocks that stored files.

Superblock	GDT	Reserved GDT	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks
------------	-----	--------------	--------------	--------------	-------------	-------------

Why having groups?

(1) Performance: spatial locality. Group inodes and data blocks of related files together (2) Reliability: superblock and GDT are replicated in some block groups

Inode Structure (128 bytes long)			
Bytes	Value	...	...
0-1	File type and permission	40-87	12 direct data block pointers
2-3	User ID	88-91	Single indirect block pointer
4-7	Lower 32 bits of file sizes in bytes	92-95	Double indirect block pointer
8-23	Time information	96-99	Triple Indirect block pointer
24-25	Group ID	...	...
26-27	Link count (will discuss later)	108-111	Upper 32 bits of file sizes in bytes

directory entry in directory block 0-3 Inode number of that file/directory 4-5 Length of this entry 6-6 Length of the filename 7-7 File Type 8+ Name in ASCII (max 255 character)

**File Deletion** Ext2直接把这个entry并入上一个entry的长度 Ext 3/4: the inode's data block pointers are zeroed out

**Hard and Soft Links:**

**what is a hard link** A hard link is a directory entry pointing to the inode of an existing file. That file can accessed through two different pathnames 可以理解为复制的时候其实是在新的写一个链接到那个Inode, 这样实际上相当于没拷贝, 但我能读取. 我们对于每个Inode现在需要一个link count, 删除时(unlink())-1, =0时 deallocated 这个 block

**Symbolic link:** create a new inode, 但是在40-99中存的是original file的path, 如果超出60bytes用1个normal inode + 1个direct data block去存 (其实这相当于一个快捷方式**Shortcut**), 如果删了源文件就会变成**dangling link**

**File system consistency:** It is about how to detect and how to recover inconsistency in a file system.

为什么存在inconsistency: 以删除file为例子

**1.** Removing its directory entry. **2.** Releasing the Inode entry. **3.** Returning all used disk blocks to the pool of free disk blocks (updating GDT).

If power-down between Steps 1 & 2 → Orphan Inode(未删除但无法通过目录访问)

If power-down between Steps 2 & 3 → Leak Storage(删除了空间浪费)



## I/O

**Block devices** 以固定大小的block传输数据(SSD/Hard disk)

**Character devices:** 以单个bit stream的形式传输数据 (如键盘、串行端口)

**Character device controller's tasks:** 1. Transform **Raw serial bit stream** to **bytes for controller's registers**

2. Checksum error handling 3. Built-in registers and data buffer for communicating with the CPU

**I/O device controller** Built-in registers and data buffer for communicating with the CPU

1. **Direct I/O:** CPU places data into (and read data from) register/data buffer directly through instructions like

`IN REG, PORT` //read the value at PORT of the device to CPU register REG 2. **Memory-mapped I/O** – Map the

device's registers and data buffer to the memory space. Both kernel (driver) and user-space 可做到

**Device Drivers:** Follow the standard programming interface offered by the OS

**I/O Communication protocol:** 有三种协议

- **Polling** – CPU puts one byte to device's DATA register – CPU keeps polling the device's READY register in order to put the next byte CPU需要不断主动检查, 可能浪费处理能力

- **Interrupt** – CPU waits for interrupt and does something else in between CPU效率更高, 不需要等待

- **Direct Memory Access (DMA)** – DMA controller on **system bus** – Offloading the per-byte polling/interrupt job from CPU to DMA controller 进一步减轻CPU负担, 提高系统整体效率

**DMA:**

1. 传输过程中, CPU可以执行其他任务

2. 避免了数据经过CPU的额外开销

3. 减轻了CPU负担

4. 可以一次设置大量数据的传输, 减少了中断处理的频率

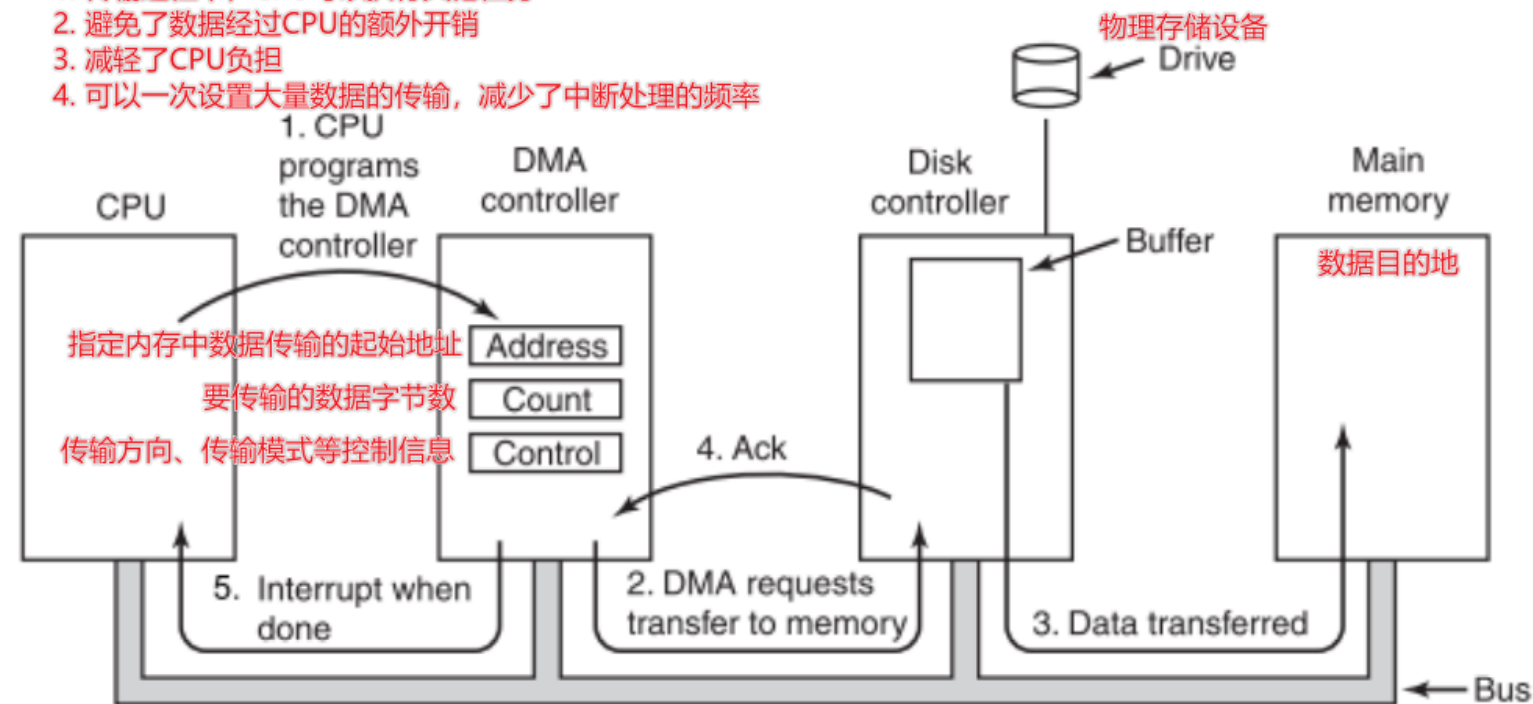


Figure 5-4. Operation of a DMA transfer.

**Print Spooling:** 打印机显然不应该并发, 并且还要解决一个user process open printer but don't use ⇒ Create a root level printer daemon process, and a spooling directory(临时储存区) 流程: 文件放到spooling directory, printer daemon从目录中获取文件发送到打印机 优势: 1. documents formatted for printing are stored in a queue at the speed of the computer, then retrieved and printed at the speed of the printer. 2. Multiple processes can write documents to the spool without waiting, and can then perform other tasks, while the "spooler" process operates the printer

**Memory-mapped I/O**

通过设置mmap(内存映射)将文件链接到进程的地址空间 Write to an address = write to a file **Advantages:** 1.

Reduced data copying 2. Simplified programming model 3. Leverages page cache 4. Supports random access 5.

Potential performance improvement

How many copies in buffer when reading files using C `stdio.h` library? 2, from disk to kernel buffer (page cache); From kernel buffer to user process buffer

**Virtual Memory**: 每个process有自己的Virtual Memory, 不同process之间可能有相同的Virtual Memory address 但指向不同Physical Address

**MMU(memory management unit)** 通常on-chip, 少量off-chip.

**How to work?** S1: CPU把虚拟地址给MMU, MMU根据Page table(在memory里)转换为物理地址 S2: 把物理地址发给Memory Controller, 再把内容传回CPU

**Page table** Every process has its own lookup table in the kernel space.

**How Large?** Number of Address  $\times$  Size of an Address =  $2^{32-12} \times 4bytes$ (for 32-bit), 虽然是一个转换的过程但实际上只存物理地址就好了

Page Table有**permission/valid-invalid bit**(0不在memory)/**Frame Number**

32位虚拟地址前20位是Virtual Page address会被Page Table翻译为Physical Frame address, 后面12位不动是Offset.

Physical Memory是被分成了固定大小的块(frame,  $2^{12} = 4KB$ ), 虚拟地址空间则是分为了跟多个Page. 一个page的虚拟地址对应一个**physical frame**.

**Page is the basic unit of memory allocation**, 所以存在internal fragmentation

**Demand Paging**: 当malloc的时候只是声称内存已分配, 只分配虚拟地址空间页面, 扩大虚拟堆空间, 访问的时候才进入内存

malloc流程: 1. 在page table中分配一个位置, 设置为invalid, Frame #设置为NIL 2. memset时如果发现是invalid的会触发**Page Fault** 3. 异常发生后OS Kernel分配memory指定Frame # 变valid 4. 但如果Memory满了, 我们需要

**Swap area**(disk中)来帮助

Swap area流程: 1. 选一个**victim virtual page**复制到Swap area 2. 腾出来的memory位置给新的page来映射 3. Swap area中存PID 和 Virtual Page #, 原本的Virtual page Bit 变为0(invalid), 下次访问的时候触发page fault

**OOM(out of memory)**: swap area和 memory都满了

**1st stage**: fill free Memory **2nd stage**: Swapping out other processes' frame to disk **3rd stage**: Swapping it own frame out (Disk activity flies high! 持续Page Fault) **Final Stage**: swap area full, Kill this OOM process. 然而在这之后其余process运行时将持续发生page faults(Thrashing)  $\Rightarrow$  Disk activity flies high again

**Swap area** is a space reserved in a permanent storage device: Linux是一个单独分区Swap partition, Windows 是一个hidden file `pagefile.sys` in one of the drives

**fork() implementation**: 从Userspace memory的视角来看子进程和父进程是独立的, 它copy了一份stack\heap....但这种copy是heavyweight的 $\Rightarrow$ **Copy-on-write(COW)** **COW technique**: fork后child process和parent process不立即复制memory page frames, 而是共享其会被标注为只读。当需要修改(写入)时, 出现page fault, 真的做一个copy

**Page replacement algorithms**: goal: minimize further page faults

**1. Optimal**, 如果知道full reference string, 直接选最少的 **2. First in First out** **3. Least recently used(LRU)** 方法1: 每个frame有age counter, 页面被应用的时候变为0, 其他的+1, 时间复杂度 $\Theta(n)$ , 空间需要n个int方法2: Doubly linked list, 被引用就放到头, 每次去掉尾, 时间 $O(1)$ , 空间2n个指针 LRU is approximate but hard to implement  $\Rightarrow$  **4. Clock Algorithm**: Circle linked list, 有一used bit, 被reference修改为1, 需要victim时像始终指针一样扫描, 是0就用 1则变为0

**Page table is huge  $\Rightarrow$  \$4MB**: Use Multi-level page table with unused pages not stored. 对于一个虚拟地址的前20bits, 高10bits对应一级页表中的1024项, 每一项对应一个一级页表, 后10bits对应查出来的二级页表的1024项. 相当于这个拆分:  $4MB = 1024 * 1024 * 4$ , 这样能分散的储存, 如果某个区域未分, 就不需要对应的二级页表.

**Paging Hardware support**: 出现Context switch时, Page Table也要换

TLB(Translation Lookaside Buffer): Cache recent translation of virtual page to physical frame, TLB在CPU和MMU之间, CPU首先看TLB有没有存, 没存让MMU再去Memory中找