OS Includes:

a program ⇒ "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.1.8GHz = 1.8 x 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. => Multiple instruction cycles run in parallel. 在每个时钟周期内同时执行多条指令

Processsor Design CISC & RISC

时钟周期: CISC 多周期执行指令，RISC 单周期执行 -.*内存访问.*：CISC 任何指令可访问内存，RISC 仅 LOAD/STORE 指令访问 -.*流水线.*：CISC 不使用或较少使用流水线，RISC 高度流水线化 -.*指令执行.*：CISC 由微程序解释，RISC 由硬件直接执行 -.*指令格式.*：CISC 可变长度，RISC 固定长度 -.*指令集.*：CISC 指令和寻址模式多，RISC 指令和模式少 -.*复杂性分布.*：CISC 复杂性在微程序，RISC 复杂性在编译器 -.*寄存器.*：CISC 单

一寄存器组，RISC 多寄存器组
RISC 优势：执行速度快、功耗低、设计简单，适合移动设备和嵌入式系统
CISC 优势：代码密度高、兼容性好、对编译器要求低，适合复杂计算任务
桌面/服务器：CISC (x86/x64) 主导，因历史兼容性和软件生态
移动/嵌入式：RISC (ARM) 主导，因低功耗和高效率
Von Neumann 优势: 设计简单，成本低 内存分配灵活 适合通用计算任务 [程序和数据在同一存储器]（RISC: ARM7,CISC: Pentium)
Harvard 优势: 更高的执行效率 减少指令和数据访问冲突 更适合实时系统和特定应用 可支持不同宽度的数据和指令总线 (RISC: ARM9,CISC: SHARC(DSP))
Sequential access is faster than random: 可以利用缓存(caching)和预取(pre-fetching)机制 （random hurts CPU's speculative exectution）
CPU has a 1-bit register to check if the currently in **user-mode** or **kernel-mode**

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
exec之后会exit(0)，后续的不会被执行，will never return to original code
1. It will **replace** user-space info: Program Code, Memory, Register values.
2. kernel info is **preserved**: PID, process relationships.
fork() + exec*() + wait() = system() [**exec*()** 函数族用新的程序替换子进程的内存空间]
Environment variables: main() 函数最后会有一个**envp是字符串数组存环境变量 SHELL: The path to the shell that you're using. HOME: The full path to your home directory. PWD: The full path to the directory that you're currently on. USER: login name. EDITOR: Default text editor PRINTER: 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()
Can a process execute more than one program? Yes, keeps on calling the exec system call family. （代码和数据会更新，但pid和其他的kernel-space info保持不变）

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)

可中断阻塞 (Interruptible)：进程等待某个条件，但可以被信号打断；对系统影响较小，进程可被管理和控制

可以通过发送信号（如SIGINT、SIGTERM）使进程从等待状态返回；允许用户或系统管理员在必要时终止进程

常见情况：1.read()从终端或网络读取数据；2.`sleep()`、`wait()`函数调用；3. 等待信号量或互斥锁

e.g. Ctrl+C中断正在等待用户输入的程序

不可中断阻塞 (Uninterruptible)：进程处于不能被信号打断的等待状态；如果条件长时间不满足，可能导致系统资源占用，甚至需要重启系统

即使发送SIGKILL信号也无法终止进程；通常涉及关键的系统级操作，中断可能导致数据不一致；必须等待操作完成或系统条件满足才能继续

常见情况：1.直接磁盘I/O操作（不通过缓冲）；2.某些系统调用（如`sync`）；3.NFS服务器无响应；4.设备驱动程序中的关键操作

e.g.系统正在将数据写入磁盘，此时即使尝试强制关机，也必须等待写入完成

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

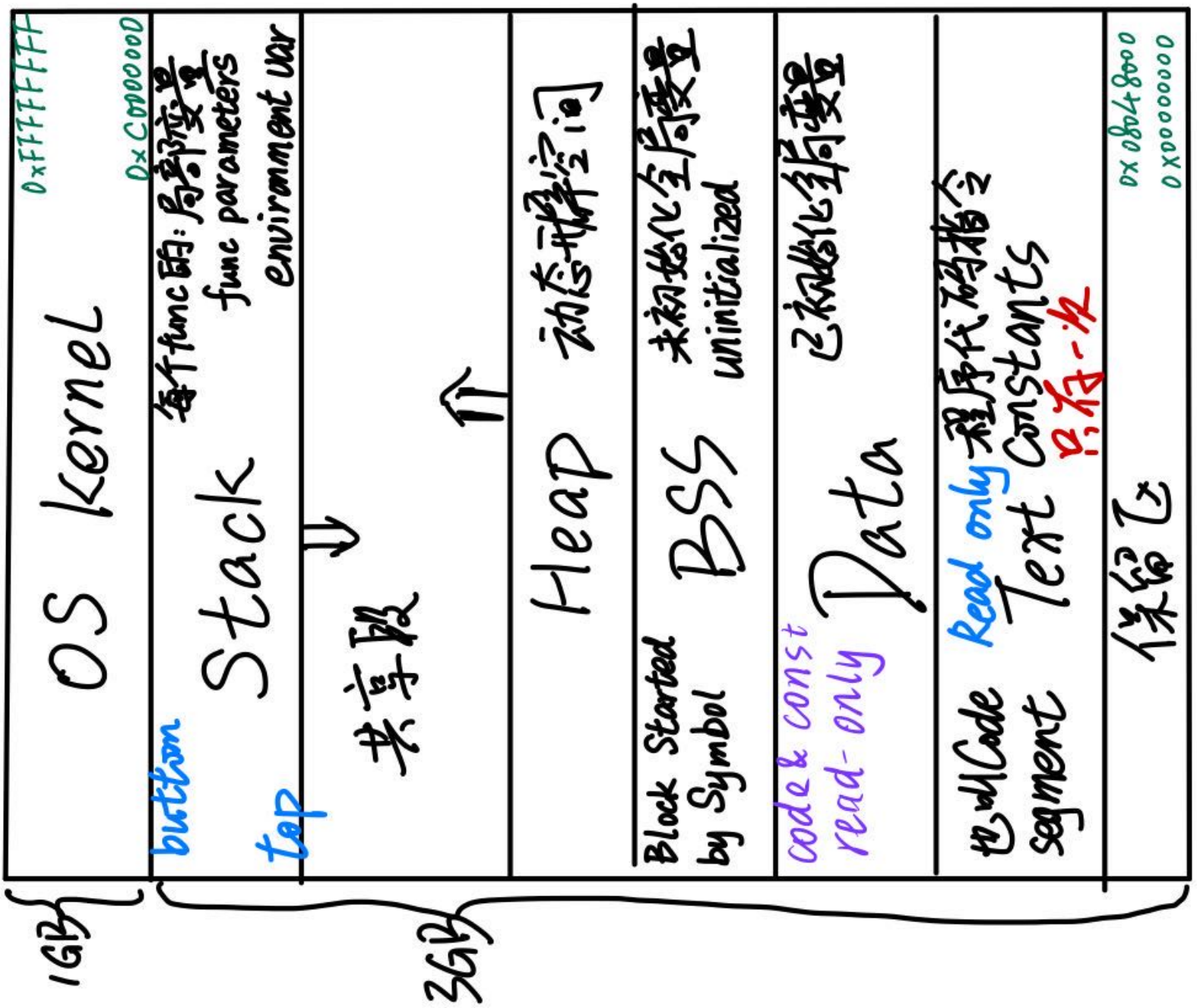
2. Backup all current context of that process to the kernel-space's PCB.
3. Load the Context of that process from the main memory to CPU.
问题: **Context Switch is expensive (上下文切换是昂贵的)**
Direct costs in kernel: Save and restore registers, Switch process address space.
Indirect costs: cache misses, increase 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Defalut 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 = 4GB



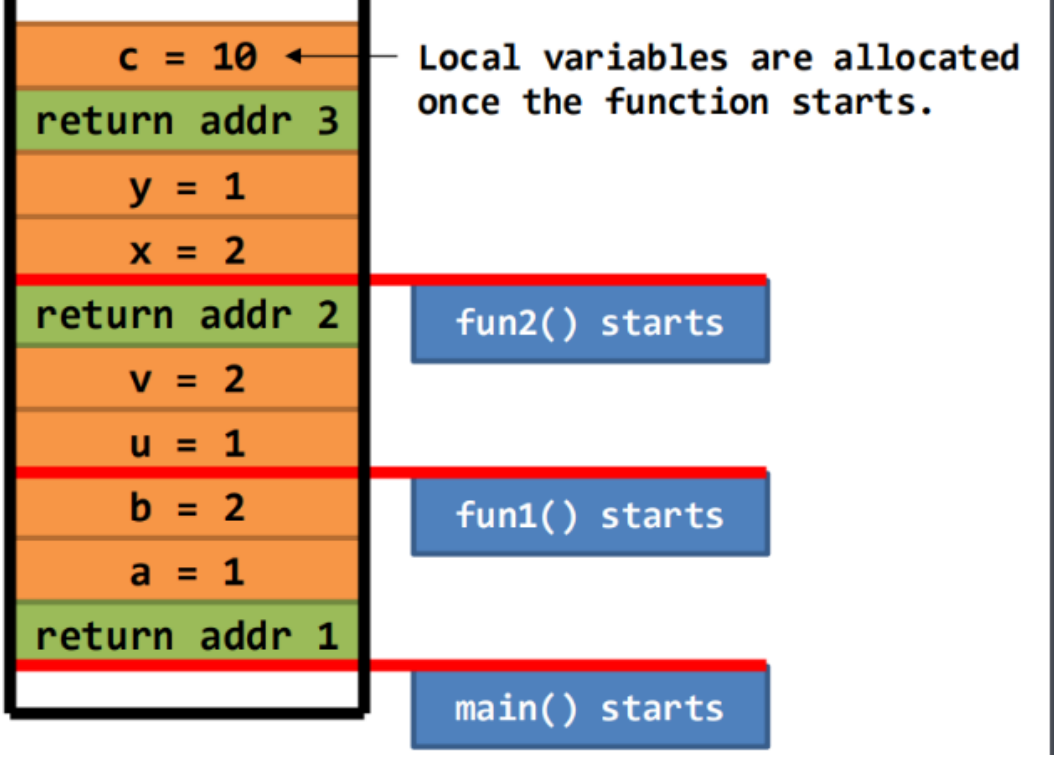
Stack in-depth

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




```c
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}
int fun1(int u, int v) {
    return fun2(v, u);
}
int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

Heap （heap比stack大的多）每次malloc都会创造一个空间, 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: 写入read-only段或读、写unallocated段报错（试图访问无权访问的内存)

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	Costly process switching	Cheap thread switching
Costly creation	Cheap creation	One process multiple threads	
Process can't access the memory area of other processes	Threads share the same memory area	Different programs	Same program (different thread functions)

Kernel is a multi-thread program, it create kernel threads & OS threads
note: Each core simulates two (virtual) cores (Virtual cores alternating between them on a physical cycle-by-cycle basis)

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., **`exit()`, `yield()`**. -**particular kinds of interrupts (e.g., periodic clock interrupt, a new process steps in) are detected**.
Pros	Good for systems that emphasize **interactiveness**. - 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 的任务中选最快完成的, 但存在长期饥饿问题, 一个任务很早就ready, 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，如果这个任务的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，the responsiveness of the process is great但别的不占优

2. Priority Scheduling: 每个task都有priority, 高priority的先执行*(high priority—usually short-lived,but important)

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.

对称多处理(SMP)架构

1. 所有进程/线程共享一个公共就绪队列(Common ready queue) →(Race Condition)

2. Scheduling: 处理器/核心调度器检查公共就绪队列并选择一个进程执行

3. Affinity Scheduling: **软亲和性(Soft affinity)**: 尝试将同一进程/线程保持在同一核心 **硬亲和性(Hard affinity)**：使用`pthread_attr_setaffinity_np()`函数手动将线程绑定到特定核心 这在NUMA(非统一内存访问)架构中尤为重要

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

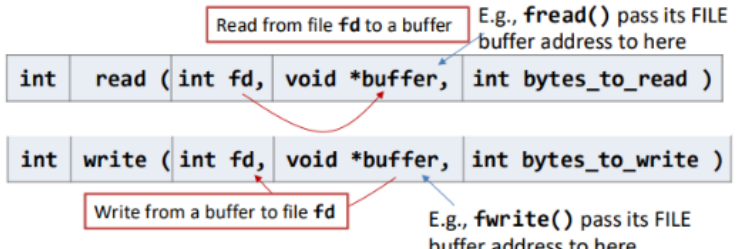
Common Attributes	FAT32	NTFS	Ext2/3/4
Name	✓	✓	✓
Size	✓	✓	✓
Permission	The design of FAT32 does not include any security ingredients.	✓	✓
Owner		✓	✓
Access, creation, modification time	✓	✓	✓

Common Attributes	Way to change them?	
	Command?	Syscall?
Name	mv	rename()
Size	edit it using vi and then save…	write(), truncate(), etc.
Permission	chmod	chmod()
Owner	chown	chown()
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stuct 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
```



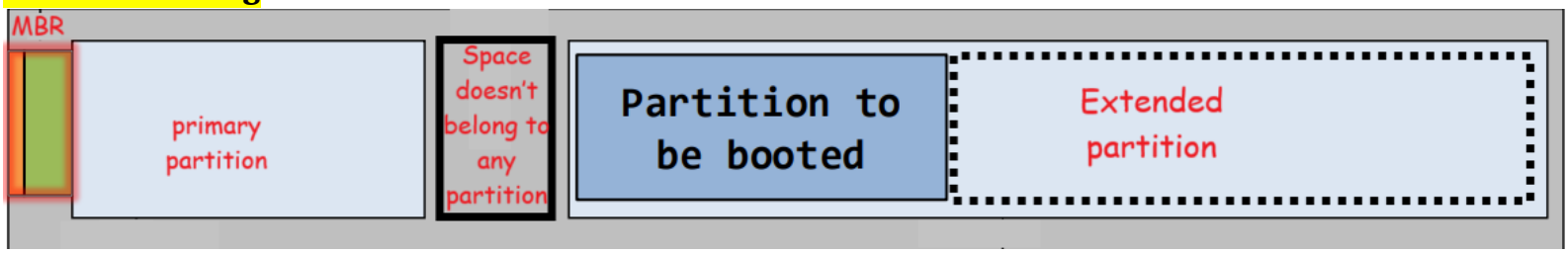
Library calls that eventually invoke the read() system call	Library calls that eventually invoke the write() system call
scanf(), fscanf()	printf(), fprintf()
getchar(), fgetc()	putchar(), fputc()
gets(), fgets()	puts(), fputs()
fread()	fwrite()

`read()`的流程:

S1: 看是否 the end of the file is reached or not. Comparing size and file seek. （比较文件大小和当前文件指针位

置）
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. （用户空间和内核空间是隔离的）
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. （在缓冲区已满时也会flush)
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, maxOS)
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 hand 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. e.g. 如果一个磁盘有4个盘片（8个表面），每个磁道有63个扇区，要访问C=2, H=5, S=10的位置：物理位置 = (2 × 8 + 5) × 63 + 10 - 1 = 21 × 63 + 9 = 1332
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

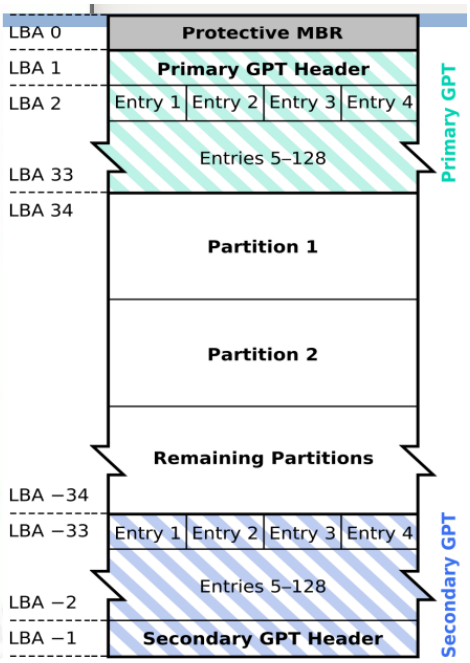
Booting(启动)
Legacy boot流程: 1. BIOS(store in EPROM) locates the first bootable device (SSD/USB) 2. Executes its boot code (in MBR) 3. Boot code executes the fatter boot loader 现在开始用UEFI
MBR(Master boot record), stores **Boot code**(橘色部分, 前446bytes) & **partition table**(最多4个, 绿色部分, 后64字节), 结束标记magic number `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 × 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)

挂载（**Mounting**） 是操作系统将一个存储设备（如硬盘分区、CD-ROM、USB驱动器等）或文件系统与目录树中的特定访问点（称为"挂载点")关联起来的过程。这使得存储设备上的文件和目录可以通过这个挂载点被访问，就像它们是本地文件系统的一部分一样。

挂载操作不会删除挂载点原有内容，只是暂时使其不可见：原挂载点内容在存储介质上保持不变；卸载后，原内容会重新可见；这种机制允许灵活地组织和访问不同存储设备上的数据

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)** ⇒ 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

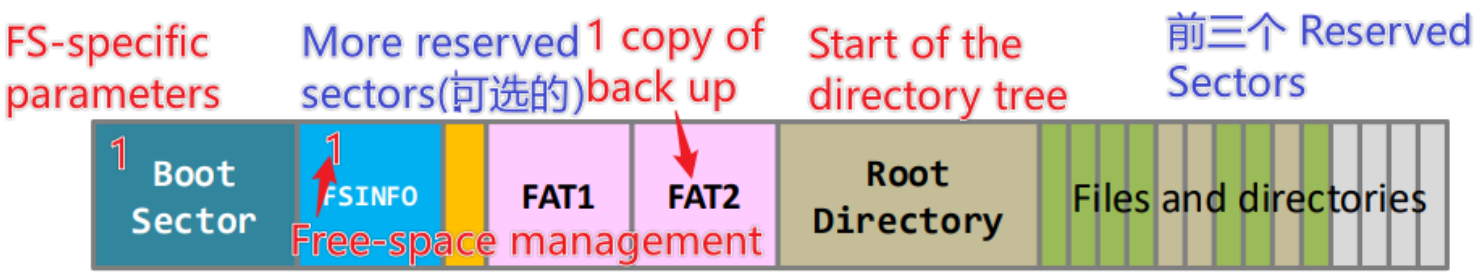
存在的问题 **1.** Internal Fragmentation(a file is not always a multiple of block size) ⇒ Last Block 可能没 fully filled. **2.** Poor random access performance, 如果我要访问File的19th block中的内容, 我需要从头一个一个读，直到从第18个找到19个

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address. 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×Cluster address



Root Directory（储存根目录的文件和子目录项）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) 是little endian储存, 所以实际顺序是31→28

找文件流程: 在directory里面找First Cluster, 然后从FAT里一直读, 直到读到最后一个. 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 directories for deleted entries (first byte 0xE5), restore original filename, extract file size (bytes 28-31, little-endian) and first cluster number (bytes 20-21 and 26-27). For single cluster files, read data directly from first cluster; for multi-cluster files, rebuild the 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.

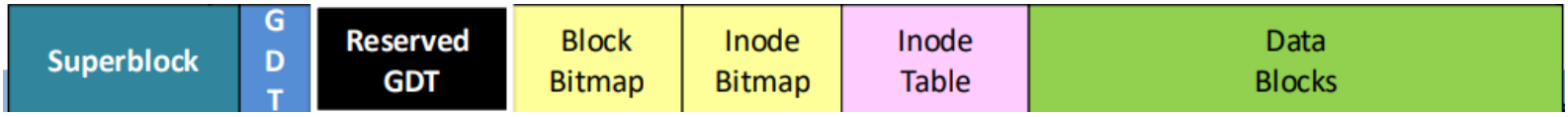
3. Inode allocation

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** = $(direct + (ith\ indirect \div address\ length)^{i}) \times block\ size$

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.



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
2-3	User ID
4-7	Lower 32 bits of file sizes in bytes
8-23	Time information
24-25	Group ID
26-27	Link count (will discuss later)
...	...
40-87	12 direct data block pointers
88-91	Single indirect block pointer
92-95	Double indirect block pointer
96-99	Triple Indirect block pointer
...	...
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length; Ext 3/4: the inode's data block pointers are zeroed out

Hard and Soft Links:

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**(悬空链接)

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(无法通过目录访问但仍占用资源)

If power-down between Steps 2 & 3 ➔ Leak Storage(数据块被标记为已使用但实际未被分配)

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;高速设备可能导致频繁中断

• **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:

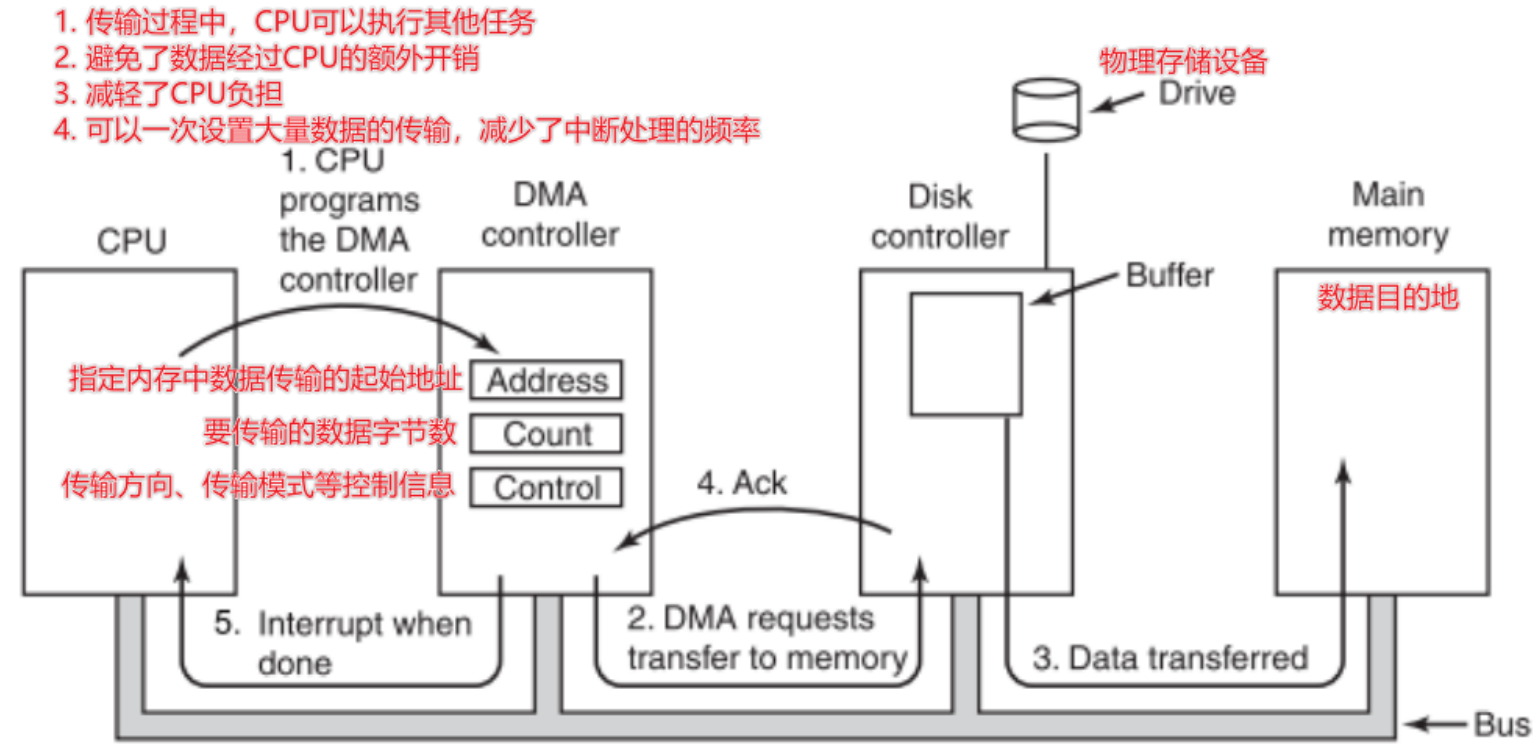


Figure 5-4. Operation of a DMA transfer.

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 on user-process

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

adv: consider 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 × 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很多个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

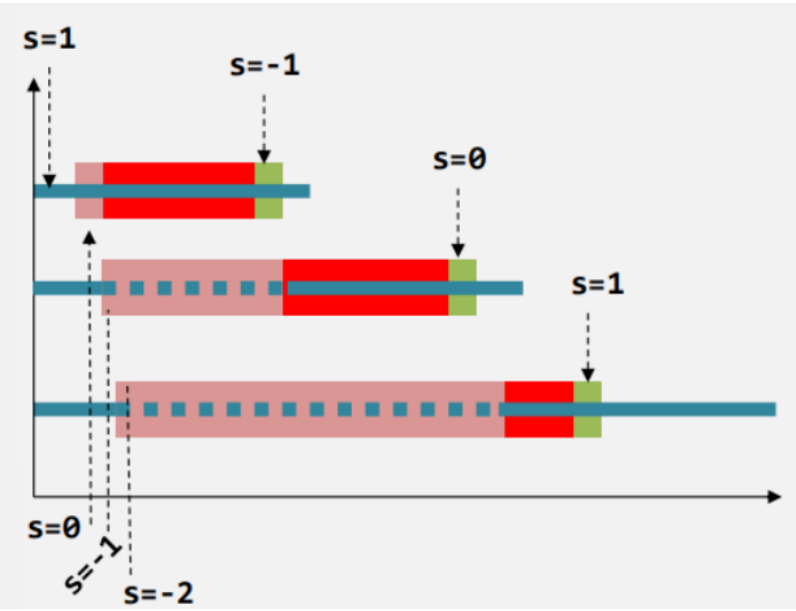
Demand Paging: 当malloc的时候只是声称内存已分配, 只分配虚拟地址空间页面, 扩大虚拟堆空间, 访问的时候才

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Virtaul 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 files 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(Thrasing) ⇒ Disk activity flies high again 系统花费大量时间在页面换入换出上，而几乎没有时间执行实际工作
Swap area is a space reserved in a permanent storage devise: Linux是一个单独分区Swap partition, Windows 是一个hides 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⇒**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Θ(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 ⇒ 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 ⇒ $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, 每一项对应一个二级页表, 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

Synchronization

concurrent access may yield the horrible ***data race***: may happen whenever “shared object” + “multiple updates” + “concurrently”
Solution: **Mutual exclusion(互斥)**: Not to access the “shared object” at the same time
A **critical section** is the code segment that accesses shared objects. (Note that one critical section can access more than one shared objects.)
two solutions: ***Locking***(=Mutual Exclusion) and ***Lock-free***(check if the final result)
lock-based solution: **Mutual Exclusion** and **Bounded Waiting**
Disabling interrupt when the process is inside the critical section. [When a process is in its critical section, no context switch]
is usable for kernel level for old CPUs but now for Multi-core complication – It is possible that another core modifying the shared object in the memory. Modern multi-core CPU bundles with atomic instructions to make sure one core’s modification is atomic
Spin-based Lock(自旋锁)
for process 0， 当turn != 0时waiting，否则进入CS，结束后转换turn=1。
Busy waiting wastes CPU resources ； But very OK for short critical section；But this simple implementation imposes a “strict alternation” order （Sometimes you give me my turn but I’m not ready to enter CS yet then

you have to wait long）
当进程1已经完成cs的部分，正在进行剩余部分时，进程0无法进入cs
Peterson's solution to implement a spin-lock（one more extra shared var: ***interested***）
在lock函数，for process a，设定interested[a] = TRUE, turn = process（`turn`变量的含义是：如果两个进程同时想进入临界区，谁应该让步); 当 turn == process a && interested[other] == TRUE 时waiting。在unlock中，for process a，设定interested[a] = FALSE。
= Busy waiting + shared variable turn for mutual exclusion + shared variable interest to resolve strict alternation
However, suffer from **priority inversion problem**: 低优先级持有锁，高优先级被阻塞，higher priority takes CPU for busy waiting，导致性能下降
cannot work for >2 processes 但是可以推广
Sleep-based lock: Semaphore(信号量)
Semaphore is an extra shared struct : Include
1.an integer that counts the # of resources available (Can do more than solving mutual exclusion) and 2. a wait-list




```
void sem-wait(semaphore *s){
  disable_interrupt();
  *s = *s - 1; //s.v--
  if ( *s < 0 ) {
    enable_interrupt(); //允许其他进程的响应和调度
    sleep(s); //add2waitlist
    disable_interrupt();}
  enable_interrupt();}
void sem-post(semaphore *s){
  disable_interrupt();
  *s = *s + 1; //s.v++
  if ( *s <= 0 ) wakeup(s);
  enable_interrupt();}
```

`sem_t *sem_open(const char *name, oflag, mode_t mode, unsigned int value);` oflag打开标志 `O_CREAT`创建 `O_EXCL`独占 `mode`权限模式(0666) `value`初始值
`int sem_wait(sem_t *sem);` 信号量>0, -1就返回, =0阻塞直到可用 `int sem_post(sem_t *sem);`信号量+1, 如有等待则唤醒 `sem_close(sem);`关闭当前进程对信号量的引用`int sem_unlink(const char *name);` 标记信号被删除, 所有进程都关闭他就实际删除
Unnamed Semaphore used only by threads in the **same process** or threads in different processes but have **mapped the same memory** into their address spaces
`int sem_init(sem_t *sem, int pshared, unsigned int value);` `pshared` = 0表示当前process使用, 非0则多进程共享 `int sem_destroy(sem_t *sem);`
Shared Memory 共享内存的文件应该编译时用 `-lrt` 链接
`int shm_open(const char *name, int oflag, mode_t mode);` 创建或打开共享内存对像`oflag`为(`O_RDONLY`只读`O_RDWR` 读写 `O_CREAT`如不存在则创建,| 连接), 有问题返回-1 `mode`权限模式（当O_CREAT标志使用时），如0644
`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);` 将文件或设备映射到内存中 `addr`映射起始地址, NULL/0系统自己选择 `length`映射自己长度 `prot`内存保护标志(`PROT_READ` 可读 `PROT_WRITE` 可写 `PROT_EXEC` 可执行`PROT_NONE` 不可访问) `flags`(MAP_SHARED 共享映射，修改对其他进程

可见; MAP_PRIVATE 私有映射，修改不会写回; MAP_ANONYMOUS 不基于文件的映射→与fd=-1一起用) `fd` 文件表描述符(`shm_open` 返回那个) `offset` 文件偏移量通常为0

`int munmap(void *addr, size_t length);` 取消映射 `int shm_unlink(const char *name);` 删除共享内存对象

IPC problems(Inter-process communication)

Producer Consumer Problem (The Bounded-Buffer Problem): [Single-Object Synchronization]
producer 会产生 item 存放到 buffer 中，而 consumer 可以将数据从 buffer 中取出 item (e.g. pipe)
Dining Philosopher Problem: [Multi-Object Synchronization] 五个哲学家，ta 们围坐在一张圆桌旁，每个哲学家面前都有一碗米饭，而 ta 们两两之间分别有一根筷子,可以思考或者拿筷子
Reader Writer Problem: reader 和 writer 的冲突；Writer 和 writer 的冲突；

Producer-consumer problem

#1 producer想要在缓冲区中放入一个新项目，但是缓冲区已满。producer应该等待。consumer应该通知producer它已经退出队列
#2 consumer想要从缓冲区中消费一个项目，但是缓冲区是空。consumer应该等待。producer应该在进入队列后通知consumer
ls → pipe → less
note: When "ls" process goes back to the running state, it immediately writes a byte to the pipe so as to complete the execution of the write() system call.
solving by semaphore
`mutex`: 互斥信号量，初始值为1，用于保护缓冲区的互斥访问； `avail`: 表示缓冲区中的可用空间，初始值为缓冲区大小N； `fill`: 表示缓冲区中已填充的项目数，初始值为0;
for producer: item = produce_item(); sem_wait(&avail); sem_wait(&mutex); insert_item(item); sem_post(&mutex); sem_post(&fill);
for consumer: sem_wait(&fill); sem_wait(&mutex); item = remove_item(); sem_post(&mutex); sem_post(&avail);
Q1: Necessary to use both "avail" and "fill"? ANS: Yes. If consumer gets CPU first, it removes item from NULL
Q2: Can we swap Lines 5 & 6 of the producer? ANS: No.This scenario is called a deadlock [Consumer waits for Producer's mutex at line 5 (waits for Producer (line 8) to unlock the mutex) ;Producer waits for Consumer's avail at line 6 •(it waits for Consumer (line 8) to release avail)] 顺序应该是先申请资源再获得锁
Livelock: 在活锁情况下，线程或进程并没有完全阻塞或停止 - 它们仍在执行操作并能从等待状态返回，但却无法推进完成实际工作。

Pthread's sleeplock

pthread_mutex_lock(&mutex); //- If the mutex is not locked, lock the mutex. - If the mutex is locked, block the calling thread.
pthread_mutex_unlock(&mutex); //- If the mutex is locked, then unlock the mutex. If there are threads blocked because of this mutex, one of those threads will resume. - If the mutex is unlocked, do nothing.
Thread Function:获取互斥锁;{检查共享变量`shared`是否小于最大值`MAXIMUM`;如果小于最大值，则增加共享变量的值;如果达到或超过最大值，则释放锁并退出循环;}释放互斥锁;随机休眠0-1秒

信号量(Semaphore)

If protecting shared resources (i.e., requires mutual exclusion) ➔ use semaphore or pthread_mutex
When doing something more (e.g., IPC problems like producer-consumer) ➔ use semaphore or Pthread's condition variables
Spinlock using **TAS[test_and_set()]** (by Maurice Herlihy): 当锁未被持有时， `lock`值为`false`;`test_and_set(&lock)`读取`false`并将`lock`设为`true`，然后返回`false`;`while`条件为`false`，循环结束，线程获得锁; 当锁已被持有时， `test_and_set`返回`true`，线程继续自旋

问题：不足以实现超过两个线程的无等待同步；ABA问题无法解决

CAS[compare_and_swap()]: lock - a shared var

锁获取：线程尝试原子地将锁值从0(未锁定)更改为1(已锁定);如果成功(CAS返回true)，线程进入临界区;如果不成功(锁已被另一个线程持有)，线程在while循环中自旋，不断尝试直到成功

锁释放：完成临界区后，线程简单地将锁设回0;这允许另一个自旋的线程获取锁

初始化：锁初始化为0(未锁定状态)：`Initialize _Atomic lock = 0`

Lock-free:线程不阻塞等待资源，而是直接尝试操作，如果发现冲突则重试。

读取：线程读取共享数据的当前状态;**计算**：基于读取的状态计算新状态;**更新**：使用原子操作（如CAS)尝试更新; 验证：检查更新是否成功: 1成功：操作完成; 2失败：重新开始整个过程

Atomic Instruction: `<stdatomic.h>` 在定义变量前+`_Atomic`使其为原子化的

`_Bool atmoic_compare_exchange_weak(volatile A *object, C *expected, C desired);` 比较object指向的值是不是等于expected, 如果是用desired替换object的值, 不是就用object替换expected, A为atomic type, C为non-atomic counterpart. volatile 类似const是type的一个property , 表示访问之前可能会发生变化, 即使没有比修修改. 函数返回值是比较的结果

`atomic_load(x)`给x做一个本地的保存

lock-based: lock-holder sleeping; lock-holder diess; deadlock **VS** lock-free: ABA problem; difficult to code **VS** wait-free: guarantee progress for every thread; operation-X() must finish in a finite number of steps; hard to achieve

ABA problem Solution: adds a **counter alongside** each pointer(之前stack的例子中lfstack_t加一个tag, 每次+1). This ensures that when comparing values, both the pointer and its associated counter must match, effectively detecting any intermediate changes.

memory consistency: Strong x86 (i.e., Intel and AMD)➔ cache coherence ➔core 2 reads “new”; Weak ARM (Advanced RISC Machine) ➔ almost all mobile devices ➔ no cache coherence ➔ core 2 reads “old”

Dining philosopher: requirements:mutual exclusion; deadlock-free 先释放互斥锁，再等待条件

```c
void captain(int i) {
if (state[i] == HUNGRY && state[L]
!= EATING && state[R] != EATING) {
        state[i] = EATING;
        signal(p[i]);}}
```

```c
mutex_t mutex;  // 互斥锁
semaphore p[N]; // 每个哲学家一个信号
State state[N]; // 每个哲学家的状态
#define L (i + N - 1) % N // 左邻居
#define R (i + 1) % N     // 右邻居
```

```c
void pickup_chopsticks(int i) {
    lock(mutex);
    state[i] = HUNGRY;
    captain(i);
    unlock(mutex);
    if (state[i] != EATING)
        wait(p[i]);}
```

```c
void putdown_chopsticks(int i) {
    lock(mutex);
    state[i] = THINKING;
    captain(LEFT);
    captain(RIGHT);
    unlock(mutex);}
```

```c
/* 1. */pthread_t tid;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond1, cond2;
int con; // share
/* 2.*/pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond1, NULL);
pthread_cond_init(&cond2, NULL);
/* 3. */pthread_create(&tid, NULL, funcname, NULL);
pthread_join(tid, NULL); // 等待线程结束
// 在函数内部开始建立锁
/* 4. */pthread_mutex_lock(&mutex);
/* 有时需要等待某个条件满足 */
while(/* 自己不能的条件 */) {pthread_cond_wait(&cond1, &mutex); }
// 向自己不利的条件变化一下
/* 5.*/pthread_cond_signal(&cond1);
pthread_mutex_unlock(&mutex);
```

```c
int pipefds[2];//1是写入
pipe(pipefds);//不能等于-1
write(pipefds[1], "CSCI3150", 9);
read(pipefds[0], buf, 9);
//如果要fork 用完赶快关掉
close(pipefds[0]);
```