1. Intro of OS

1.1 Bootstrapping 引导程序

Bootstrapping又被称为引导过程,是计算机启动时加载操作系统到内存中的过程.

当开启计算机时，CPU首先在一个预定的位置（如x86架构中的BIOS或UEFI固件）寻找启动指令。这个过程称为自检（POST），之后，控制权(系统执行指令的能力和权限)被交给引导加载程序（Bootloader）。在Windows系统中，这个引导加载程序通常位于硬盘的主引导记录（MBR）或者近年来使用的GUID分区表（GPT）的等效区域。

引导加载程序有责任:

1. 识别并初始化系统硬件。
2. 加载操作系统内核到内存中并执行它。

这个过程称为“引导”或“启动”，因为它好比是计算机通过其自身的引导带（bootstrap）来“提升”自己进入一个可操作的状态。这个术语来源于“自力更生”的表述，意指一个系统能够不依赖外部输入自主地启动。
在内核被加载并执行之后，它接管系统，初始化其它系统级别的软件，最终提供用户接口，如命令行或图形用户界面。这样，计算机便准备好接收用户输入，并执行程序了。

1.2 Context Switching 状态切换

Context switching 是指操作系统内核在多个进程（或线程）之间切换执行权的过程。这是多任务操作系统进行任务管理的基本功能之一，允许单个处理器在多个任务之间迅速切换，从而给用户一种同时执行多个程序的错觉。

具体来说，在进行 context switching 时，操作系统会执行以下步骤:

1. **保存状态**：操作系统会保存当前正在执行的进程（或线程）的状态信息。这通常包括程序计数器、寄存器内容、系统调用状态、内存映射等。
2. **加载状态**：操作系统随后加载另一个进程（或线程）的状态信息到这些硬件组件中。这个过程包括更新程序计数器以指向新任务的代码位置，恢复寄存器的内容，以及设置内存访问权限等。
3. **执行新任务**：加载新状态后，处理器开始执行选中的新任务。

1.3 Kernels

1.3.1 Monolithic Kernels 整体内核

Monolithic Kernels（整体内核）是一种操作系统内核的设计方式，它将大部分的系统服务和驱动程序集成到内核空间中。这种设计与微内核（Microkernel）或层次内核（Layered Kernel）形成对比。

整体内核的特点包括：

- **集中式管理**：几乎所有的系统管理任务，如进程管理、内存管理、文件系统以及网络堆栈等，都在一个大的内核空间中执行。
- **性能**：因为服务和驱动程序在内核空间运行，它们可以直接访问硬件和内存，这通常会带来更好的性能，尤其是在系统调用和设备操作时。
- **简化的通信**：内核中的不同服务和模块之间可以直接进行函数调用，而无需复杂的消息传递机制。
- **安全与稳定性风险**：如果内核的任何部分发生故障，整个系统可能会受到影响。这意味着整体内核可能比微内核更容易受到单点故障的影响。
- **可移植性和维护性**：整体内核因为其复杂性和庞大的代码基础，可能在可移植性和维护性方面面临挑战。

整体内核的主要特征：

- **操作系统的主要部分在内核空间运行**：这包括设备驱动程序、文件系统、进程间通信（IPC）等。这些都是操作系统的核心组件，它们在内核空间运行，这是一个有高权限的执行环境。
- **内核空间**：内核空间是指保留给操作系统内核的内存区域，并且在这个空间中运行的代码可以执行特权操作，如直接访问硬件或管理系统资源。在大多数现代操作系统中，内核空间与用户空间相隔离，后者是应用程序运行的环境。
- **内核的模块化**：尽管是整体内核，但现代操作系统（如Linux）允许某些内核组件以模块化的形式存在，可以在运行时动态加载和卸载。在Linux中，`modprobe`工具用于管理这些内核模块：加载新的模块以增加功能，或者卸载模块以释放资源或因为不再需要某个特定的硬件支持。

通过动态加载和卸载内核模块，系统管理员可以根据需要调整系统的硬件支持和功能，而无需重启系统。这提供了一种灵活性，使得整体内核的系统能够更加适应不断变化的硬件和软件环境。

1.3.2 Microkernels 微内核

在微内核设计中，只有核心的功能部分运行在内核空间，而其他部分则可以以系统服务的形式运行在用户空间。

具体来说：

- **核心内核在内核空间**：微内核中，只有最关键的组成部分，如调度器（负责决定哪个进程获得CPU时间）、进程管理（负责创建和终止进程）、内存管理（负责分配和回收内存资源）等运行在内核空间。这部分内核是始终加载的，因为它们对于操作系统的运行至关重要。
- **其他内核部分在用户空间**：与整体内核不同，微内核中的其他组件，如文件系统和设备驱动程序，可能作为系统服务在用户空间中运行。这意味着它们虽然是内核功能的一部分，但它们的执行环境与普通的用户级应用程序相同。

这种设计的优势在于：

- **安全性和稳定性**：由于非核心组件在用户空间运行，它们即使失败也不太可能导致整个系统崩溃。这增加了系统的整体稳定性。
- **灵活性**：在用户空间运行的系统服务可以像普通应用程序一样启动和停止，不需要重新启动内核来更改这些服务。

1.4 User Mode & Kernel Mode

- 大多数程序运行在user mode
- 当调用库文件或调用system call时，切换至kernel mode
  `getpid(), printf(), exit()`
  - 调用完成后，切换回user mode
- 如何切换模式：
- syscall instruction
  - software interrupt

2. Process Management

- **程序（program）** 是一组指令，它被写入计算机系统的存储器中，以便在需要时执行。程序是一种静态的实体，它包含了处理任务所需的所有指令和数据，但它本身并没有具体的执行状态。
- **进程（process）** 是执行中的程序的实例。当一个程序被加载到计算机的内存并开始执行时，它成为一个进程。进程是动态的，它有自己的执行状态和运行的上下文。每个进程都有自己的内存空间、程序计数器（记录下一个将被执行的指令的地址）和一组寄存器等。进程可以独立地执行，并与其他进程并行运行。每个进程都有独立的资源分配和管理。

2.1 Interrupts

IRQ(Interrupt Request)中断请求信号是硬件设备用来通知CPU其需要处理某些事件的一种机制。当设备需要CPU注意时，其会发送一个IRQ信号给CPU。（改变信号的沿）

- 在CPU的执行周期中，写回（write back）是最后一个阶段，在这个阶段结束时，CPU会检查IRQ
- 如果检测到IRQ线被激活，CPU会中断当前正在执行的代码，转而执行处理该设备的代码。这个过程包括以下步骤：

  1. **保存程序计数器（PC）**：当前的程序计数器（PC），它指示CPU当前执行到程序的哪个位置，会被推入（push）栈中以便之后能回到中断前的执行点。

  2. **查询中断向量表**：CPU会查看一个特殊的表——中断向量表，以找到对应的中断服务例程（Interrupt Service Routine，简称ISR）的地址。ISR是一段专门用来处理特定设备请求的小代码。

  3. **执行中断处理程序**：找到ISR的地址后，这个地址被加载到PC中，然后CPU开始执行这个中断处理程序

     - **中断确认（IA）信号**：当CPU开始处理一个设备的

中断请求时，它会发出一个中断确认（Interrupt Acknowledge，简称IA）信号告知该设备它的请求已经被接受。这样设备就知道它的信号已经被CPU注意到，并将开始被处理。

4. **恢复执行**：中断处理程序执行完毕后，之前保存的PC值会从栈中弹出（pop），重新加载到PC中，CPU随后会从被中断的地方继续之前的程序执行。

- **取消IRQ信号**：有时候，CPU处理完中断请求后，会直接取消（de-assert）IRQ线，而不是使用单独的IA线。这个行为是为了清除中断请求，让系统知道该请求已经得到处理，以便设备知道不需要继续发出中断请求。

- **中断在多进程运行中的作用：**
- **中断处理程序切换进程**：当定时器中断发生时，中断处理程序会执行，它可能会选择停止当前运行的进程并切换到另一个进程。这是实现时间共享和多任务处理的关键机制
  - **定时器**：硬件设备如定时器（timer）会周期性地（例如，每毫秒）中断CPU。这种中断被用于操作系统的时间管理和进程调度。

2.2 Process Management

- 进程运行的过程中的状态被分为三种：
- **Running**

- 在被CPU执行中

  - **Ready**
  - 排在队列中，未运行
    - 由**调度器Scheduler**选择下一个运行的进程
  - **Blocked**
  - 进程在等待其他的进程完成而没有准备好运行

2.2.1 Switching between processes

- **Process Context 进程上下文：** 当CPU在运行一个进程的时候，CPU需要维护此进程的多种信息，这被称作进程上下文，包含：
- CPU register value：这些是当前任务的中间计算结果、程序计数器、指令寄存器等。
    - Stack Pointers：指向进程栈顶部的指针，进程栈存储了执行路径、局部变量等。
    - CPU status word register：包含标志位，指示上一个操作是否产生了溢出、是否结果为零、以及中断是否被允许等状态信息。
- 当旧的进程被block时，**所有进程上下文必须被保存**，新的进程的进程上下文会被导入

2.3 Process Creation

- `fork()`：使用此命令新建一个**进程**，新建的进程被称为”子进程“，原有进程被成为”父进程“
- 子进程**获得了父进程的进程上下文副本**，但后续的操作是各自独立的

2.4 Process Control Blocks

当一个进程被创建时，操作系统会为其创建一个称为“进程控制块”（Process Control Block，简称PCB）的数据结构，用以维护该进程的相关信息。以下是PCB包含的关键信息：

1. **进程标识符（Process ID，PID）**：
    - 每个进程有一个唯一的标识符，用于区分系统中的不同进程。
2. **栈指针（Stack Pointer）**：

- 指向进程的栈顶，栈用于存储函数参数、返回地址以及局部变量。

3. **打开文件（Open Files）**：
   - 进程打开的文件列表，通常包括文件描述符和相关的文件状态信息。
4. **挂起信号（Pending Signals）**：
   - 待处理的信号集，信号是进程间通信的一种方式。
5. **CPU使用情况（CPU usage）**：
   - 该进程使用CPU的统计信息，如累计CPU时间。
     PCB存储在一个称为“进程表”（Process Table）的结构中，系统中的每个进程都有一个对应的PCB。
     当进程终止时：

- 操作系统通常会释放大多数资源，如关闭打开的文件等，并将这些资源返回给系统。
- 然而，PCB会保留在内存中，这允许子进程将结果返回给父进程。
- 父进程通过调用“wait”函数来检索子进程的结果。在这之后，子进程的PCB被释放。
  如果父进程从未调用“wait”：
- 子进程的PCB会保留在内存中，这样的子进程被称为“僵尸进程”（Zombie Process）。
- 如果僵尸进程过多，进程表可能会耗尽空间，导致无法创建新的进程。

3. Process Scheduling

- 进程可以是CPU bound（密集计算型），或I/O bound（密集读取型）

3.1 Type of Multitaskers

批处理（Batch Processing）：任务逐一执行，直至完成。在一个任务完成之前不会启动另一个任务
协作多任务处理（Co-operative Multitasking）：在这种环境中，当前正在运行的进程不会被操作系统的调度器强制挂起。进程必须自愿放弃CPU时间，使得其他进程有机会运行。
抢占式多任务处理（Pre-emptive Multitasking）：在这种模式下，操作系统的调度器可以强制挂起当前运行的进程，以便其他进程可以使用CPU。这样可以确保所有进程都能获得执行的机会。
实时多任务处理（Real-Time Multitasking）：这种模式要求进程必须在固定的截止时间前完成。如果进程未能在截止时间前完成：

- **硬实时系统**（Hard Real-Time Systems）：系统死机
- **软实时系统**（Soft Real-Time Systems）：通常只是造成不便，系统性能会降低

3.1.1 Scheduling policies for Multitaskers

- **For all types of multitaskers:**
- **固定优先级**（Fixed Priority）：根据设置的优先级顺序来运行任务。优先等级为**0**的进程会具有最高优先。
- **批处理**（Batch Processing）:任务逐一执行，直至完成。在一个任务完成之前不会启动另一个任务

- **先来先服务**（First-come First-served，FCFS）：按照任务到达的顺序来运行任务。
    - **最短作业优先**（Shortest Job First，SJF）：优先执行预计运行时间最短的任务，可能会导致**饥饿Starvation**，即预计运行时间长的任务始终被多个运行时间短的任务打断
- **协作多任务处理**（Co-operative Multitasking）：在这种环境中，当前正在运行的进程不会被操作系统的调度器强制挂起。进程必须自愿放弃CPU时间，使得其他进程有机会运行。
- **循环轮询与自愿调度**（Round Robin with Voluntary Scheduling，VC）：任务轮流获得CPU时间，但每个任务在使用CPU时必须自愿放弃CPU时间(使用 `yield()` 挂起进程)，从而允许下一个任务运行
- **抢占式多任务处理**（Pre-emptive Multitasking）：在这种模式下，操作系统的调度器可以强制挂起当前运行的进程，以便其他进程可以使用CPU。这样可以确保所有进程都能获得执行的机会。
- **带时间切片的循环轮询**（Round Robin with Timer，RR）：为每个进程分配时间切片，在时间切片结束时调度器将其挂起，并将CPU分配给下一个进程
    - **最短剩余时间**（Shortest Remaining Time，SRT）：CPU被分配给(预估)剩余时间最短的进程，这是对SJF的改进
- **实时多任务处理**（Real-Time Multitasking）：这种模式要求进程必须在固定的截止时间前完成。如果进程未能在截止时间前完成，对于硬实时系统，系统会死机；对于软实时系统，系统性能会降低。
- **Earliest Deadline First Scheduling (EDF)**：它给即将达到截止时间的任务分配最高优先级。EDF允许任务的周期变化，不需要固定周期，只要任务能在它们的截止时间之前完成。这是一种动态调度策略，任务的优先级在运行时根据截止时间进行调整。

- **Rate Monotonic Scheduling (RMS)**：它分配优先级基于任务的请求率（即周期的倒数）。周期性任务中，周期短的任务（意味着请求率高的）被赋予更高的优先级。RMS适用于硬实时系统，其中任务的周期是固定的，并且在每个周期内任务必须完成

3.2 Scheduling in Linux

- 优先级在Linux中是动态的
- 通过 `nice -n 19 tar cvzf archieve.tgz` 创建一个优先级为19的任务
  - 低于19的任务会优先执行
  - 普通用户可以创建0-19的任务
  - 超级用户(sudoer)可以创建-20到19的任务
- Linux maintains three types of processes:
- Real-time FIFO:
  - RT-FIFO processes cannot be pre-empted except by a higher priority RT-FIFO process.
  - Real-time Round-Robin:
  - Like RT-FIFO but processes are pre-empted after a time slice.
  - Linux only has “soft real-time” scheduling.
  - Cannot guarantee deadlines, unlike RMS and EDF we saw earlier.
    - Priority levels 0 to 99
  - Non-real time processes
  - Priority levels 100 to 139
- **Calculate priorities: Chapter 13.6**

4. Inter-Process Communication

4.1 Race Condition & Critical Section

- **Race Condition 竞态访问**：两个或更多的进程或线程在访问共享资源时，它们的执行顺序导致不可预知的或错误的结果的情况。它通常发生在并发环境中，尤其是当多个操作必须以正确的顺序执行时，否则可能导致数据冲突。
  举例来说，假设有两个线程，它们都试图同时更新同一个变量。如果它们的操作没有适当地同步，一个线程的更新可能会覆盖另一个线程的更新，结果是变量中的数据不是任何一个线程预期的值。
- **Critical Section 临界区**：我们将一段**可以访问共享资源的代码范围**定义为临界区 Critical Section。在这个范围内的代码会**访问共享资源**，如果多个进程的代码同时访问共享资源，则会出现race condition，这是我们需要避免的。
  所以我们先定义出一个代码区域或范围，其是会访问共享资源以出现潜在的race condition，以便进行下一步操作。

Mutual Exclusion

- **相互排斥（Mutual Exclusion）** 是指在并发编程中确保当一个线程进入临界区时，**其他线程或进程必须等待**，直到该临界区的线程退出，才能访问共享资源的一种原则或机制。
- To prevent race conditions, 4 rules must be followed:
  - No two processes can in their critical section **at the same time**

- No assumptions may be made about speeds or numbers of CPUs
  - **Note**: we can relax this assumption for *most* embedded system since they have single CPU
  - May apply to systems using multi-core microcontrollers
- No process outside of its critical section can block other processes
- No process should **wait forever** to enter its critical section

Implementing Mutual Exclusion

1. **禁用中断（Disabling interrupts）**：

```
- 原理：在单处理器系统中，通过禁用中断，当前运行的代码可以防止上下文切换，从而避免进入临界区的竞争条件。
- 优点：简单易实现；在临界区内的代码不会被打断。
- 缺点：只适用于单处理器系统；增加了系统调用和中断响应的延迟；如果临界区内代码执行时间过长，可能影响系统响应性能。
```

2. **锁变量（Lock variables）**：

```
- 原理：使用一个共享变量作为锁，任何线程在进入临界区之前必须检查并设置这个变量的状态。起始值为1，进程a进入临界区时，锁=0。进程b等待直到锁=1
- 优点：实现简单。
- 缺点：可能引起忙等（busy waiting），浪费CPU资源；不满足原子操作，仍可能发生竞争条件。
```

3. **严格轮换（Strict alternation）**：

- 原理：严格按顺序轮换，每个线程轮流进入临界区。
- 优点：简单，且保证了公平性。
- 缺点：不是很灵活，因为即使一个线程不需要进入临界区，它也必须等待其轮次来临才能让其他线程进入。

4. Peterson's solution：

- 原理：是一种软件解决方案，结合了锁变量和严格轮换的概念，使得两个线程可以安全地交替进入临界区。
- 优点：不需要特殊的硬件支持；实现了真正的相互排斥。
- 缺点：仍然使用忙等；只适用于两个线程。

5. 测试并设置锁（Test-and-set lock）：

- 原理：使用一个原子操作的测试并设置（test-and-set）指令来实现锁。
- 优点：是原子操作，因此在多处理器系统中也可以安全使用。
- 缺点：可以导致忙等，尤其在锁争用较高的时候。

6. 睡眠/唤醒（Sleep/Wakeup）：

- 原理：使用操作系统提供的睡眠和唤醒调用来控制线程的执行，线程在不能进入临界区时会睡眠，在可以进入时被唤醒。
- 优点：避免了忙等，CPU可以切换到其他任务。
- 缺点：睡眠和唤醒操作的实现可能会导致额外的复杂性，如需要防止信号丢失或错误唤醒的情况。

Semaphores

- A semaphore is a special lock variable that counts the number of wake-ups saved for future use
- A value of '0' indicates that no wake-ups have been saved
- Semaphore中有两个原子操作：
- **等待（P）**：线程在尝试进入临界区之前调用这个操作。如果信号量的值大于零，信号量的值减一，线程进入临界区。如果信号量的值已经是零，这意味着没有可用的资源，线程将被阻塞，直到信号量的值变为正。
    - **发信号（V）**：线程在离开临界区时调用这个操作。信号量的值增加一，如果有线程正在等待这个信号量，则其中一个将被唤醒。
- 当semaphore中的计数功能被禁用的时候，就变成了互斥锁
- 1 = unlocked
    - 0 = locked
- 仍可能出现deadlock

Deadlock & Busy Wait

- **死锁（Deadlock）**
- **定义**：死锁是指两个或多个进程在执行过程中因为竞争资源而造成的一种僵局。在死锁的情况下，每个进程都在等待其他进程释放资源，但这些资源又被这些进程本身所持有，从而导致所有进程都无法继续执行。
    - **条件**：死锁的发生通常需要满足四个条件：互斥条件、持有并等待条件、不可剥夺条件和循环等待条件。
    - **后果**：死锁会导致系统资源的浪费和系统吞吐量的下降。在严重的情况下，可能需要重新启动系统或采取其他措施来打破死锁。

- **处理死锁的策略：**
  1. **检测和恢复**：
     - 允许死锁发生，但需要有机制来检测它，并一旦检测到就采取措施恢复系统，比如中断并重启涉及的进程。
  2. **避免（动态）**：
     - 在运行时进行检查，以阻止可能导致死锁的资源分配。这涉及到对资源分配请求进行评估，以确保它们不会引起系统的不安全状态。
  3. **预防（静态）**：
     - 通过限制请求和分配资源的方式，从根本上排除死锁的可能性。
- **死锁通常需要以下三个条件同时满足：**

```
1. **互斥**：资源不能共享，必须由一个进程独占。
2. **保持并等待**：进程至少持有一个资源，并且正在等待获取额外的资源。
3. **循环等待**：存在一个进程链，每个进程都在等待下一个进程持有的资源。
```

Eliminate mutual exclusion解决互斥

- Not possible in most cases. 在大多数情况下，这是不可能的，因为某些资源（如打印机）本质上就是不可共享的。
- Spooling makes I/O device sharable. 通过技术如假脱机（Spooling）可以使某些I/O设备变得可共享。

Eliminate hold-and-wait 解决循环并等待

- 要求进程一次性请求其需要的**所有资源**
- 要求进程一次性请求其需要的**所有资源**

- 如果当前请求无法立即满足，则释放**所有已持有的资源**

Eliminate circular wait 解决循环等待

- 对系统中的所有资源进行排序
- 要求每个进程必须按照资源编号的升序来请求资源

Reusable & Consumable Resources

可重用资源（Reusable Resources）：

- **定义**：可重用资源是指在系统中数量固定，可以被多个进程共享使用的资源。这类资源使用完毕后不会消失，可以被释放并重新分配给其他进程。
- **特点**：
  - **数量固定**：如内存、设备、文件和数据库表等，它们的总数在运行时不会改变。
  - **非共享**：在任一时刻，每个单元要么是空闲的，要么被某个进程独占。
  - **请求-获取-释放**：进程使用这些资源时通常遵循请求资源、获取资源、最终释放资源的周期。

可消耗资源（Consumable Resources）：

- **定义**：可消耗资源是指它们的数量会随着系统的运行而变化，通常是由进程创建，并被其他进程消耗。
- **特点**：
  - **数量变化**：如消息或信号等，它们可以在运行时被创建和消耗，因此它们的总数是可变的。
  - **创建和消费**：一个进程可以释放（或说是创建）资源，无需先获取资源。而另一个进程可能会请求和获取（消费）这些资源。

死锁的关联：

- **可重用资源死锁**：如果多个进程各自持有一部分资源，并请求更多的资源时，可能导致循环等待的情况，这是死锁的经典场景。
- **可消耗资源死锁**：尽管可消耗资源不像可重用资源那样直观地与死锁关联，但如果进程间的信号通信不当，也可能导致死锁。例如，一个进程等待从另一个进程接收信号，而后者也在等待某种资源或信号，这可能形成死锁。
- **忙等待（Busy Waiting）**
- **定义**：忙等待是指一个进程在等待某个条件（通常是等待资源或信号）成立时，持续占用CPU进行循环检查，而不是释放CPU并被挂起。
    - **特点**：在忙等待中，进程不会失去对CPU的控制，而是持续消耗CPU资源来检查某个条件是否成立。
    - **用途**：忙等待通常用于实现低延迟的同步机制，如自旋锁（Spinlock）。但它也可能导致CPU资源的浪费，尤其是在等待时间较长的情况下。
- 区别
- **资源利用**：死锁会导致进程完全停止执行，涉及的资源无法被利用；忙等待会导致CPU持续被占用，但可能不涉及其他资源的浪费。
    - **进程状态**：在死锁中，进程处于阻塞状态，等待其他进程释放资源；在忙等待中，进程处于运行状态，持续检查某个条件是否满足。

Problem with Semaphores: Priority Inversion

优先级反转（Priority Inversion）是操作系统中的一个经典问题，发生在一个高优先级任务被迫等待一个低优先级任务释放资源的情况。这通常是因为有一个中等优先级的任务阻止了低优先级任务的执行，而高优先级任务又在等待低优先级任务持有的资源。

Monitor & Conditional Variables

- **监视器Monitor：** 这是一种同步构造，其封装了资源共享的访问，提供了一种安全地允许多个进程访问同一资源的方式。其他任何试图访问监视器的任何方法methods的进程都会被阻塞，直到当前的method执行完成。
- **条件变量**：监视器使用条件变量来挂起和唤醒线程。这些条件变量是监视器对象的一部分，允许线程在某些条件不满足时等待（wait），并在条件可能已变为真时被唤醒（signal）。

Barriers

屏障(Barrier) 是一种特殊形式的同步机制，用于协调一组进程或线程，而不是单个进程或线程。它通常用于并行编程和多线程应用中，确保在某个执行点上所有的进程或线程都达到了屏障点，然后才能一起继续执行。换句话说，它是一种集体等待点，直到所有成员都准备好了，才能跨过这个点。
（Checkpoint）
如何工作：
当一个进程或线程到达屏障点时，它会在那里等待，直到所有其他进程或线程也都到达这一点。一旦最后一个进程到达屏障点，所有在屏障点等待的进程或线程就可以继续执行。
使用场景：
屏障在以下情况下特别有用：

- **并行计算**：在数据处理或计算密集型任务中，可能需要将任务分割成多个部分并行处理。屏障可以确保各个部分在继续下一步之前都完成了当前步骤。
- **同步启动**：确保所有线程或进程都已准备好，然后同时开始执行。
- **迭代算法**：在每个迭代步骤结束时同步，比如在使用迭代方法求解数值问题时。

Memory Management

Physical Memory Organisation

- Physical memory is: The actual matrix of capacitors (DRAM) or flip-flops (SRAM) that stores data and instructions. Arranged as an array of bytes
- **Word**: 物理内存通常以byte管理，然而CPU传输数据的基本单位会大于1byte
- CPU传输数据的基本单位为word
    - For 8-bit machine, 1 word = 1 byte,
      For 16-bit machine, 1 word = 2 bytes,
      For 32-bit machine, 1 word = 4 bytes
      For 64-bit machine, 1 word = 8 bytes

Endianness (Big Endian & Little Endian)

- **Big Endian:** higher order bytes at lower address
  在大端字节序中，最高有效字节（MSB）存储在最低的内存地址上，而最低有效字节（LSB）存储在最高的内存地址上。这类似于我们阅读数字的方式，从左到右读，高位在前。
- **Little Endian:** lower order bytes at lower
  在小端字节序中，最低有效字节（LSB）存储在最低的内存地址上，而最高有效字节（MSB）存储在最高的内存地址上。这类似于我们阅读数字的方式倒过来，低位在前。
  x86: Little Endian, TPC/IP: Big

Alignment

- 数据可以从多个word中读取，这种从多word读取数据的情况一般发生于大的数据或者未**对齐**
- 相比于对齐的数据，读取未对齐的数据会造成传输额外的word

Memory Management

Logical & Physical Addresses

- **Logical addresses:**
- These are addresses as "seen" by executing processes code
  - 也被称为虚拟地址（Virtual Address），是由执行程序产生的地址。
  - 对于程序中的任何内存请求，都会生成逻辑地址。
  - 逻辑地址是**相对地址**，它是从**程序开始的地方**计算的。
  - 它由程序的代码通过内存管理单元（Memory Management Unit，MMU）在运行时转换为物理地址。
  - 逻辑地址使得程序员无需关心内存中的实际物理位置，可以使用一致的地址空间来编写程序。
- **Physical addresses:**
- These are addresses that are *actually* sent to memory to retrieve data or instructions
  - 物理地址是数据实际存储在物理内存中的地址。
  - 它是由计算机的硬件和操作系统管理的。

    - 当MMU转换逻辑地址时，它会使用页表或其他数据结构来找到对应的物理地址。
    - 物理地址直接指向内存中的一个实际位置。

Multiple Program System

多程序系统是一种操作系统环境，它允许多个程序同时驻留在内存中，以便CPU可以在它们之间切换来实现多任务处理。这种系统的设计旨在提高资源利用率和系统吞吐量，因为当一个程序等待某些事件（如I/O操作）完成时，CPU可以切换到另一个程序继续工作，从而减少CPU空闲时间。
在多程序系统中，操作系统负责内存管理，确保每个程序有独立的地址空间，防止程序之间相互干扰。这通常通过使用逻辑地址和物理地址的转换来实现，即程序编写时使用逻辑地址，运行时操作系统和硬件负责将逻辑地址映射到物理内存地址。

- Having multiple processes complicates memory management:
- **Conflicting addresses**: >1 program expects to load at the same place in memory
    - **Access violations**: program overwrites the code/data of another program/OS
    - The ideal situation would be to give each program a section of memory to work with
    - Basically each program will have its own address space

Base and Limit Registers

- 使用base 和 limit 寄存器可以解决多程序系统带来的**内存地址冲突address conflict**和**同时访问access violations**问题

- Base Register:
- 存储分配给程序的物理内存块的起始物理地址。当程序中的逻辑地址生成时，系统会将逻辑地址加上基址寄存器的值来得到物理地址。
- Limit Register:
- 存储程序分配的内存块的大小。它用来检查生成的物理地址是否在分配的内存块内。
- 每个程序都会被分配base 和 limit register。这个分配是固定的。任何想要访问超出或低于此限额的操作会造成**segmentation faults**
- 使用此分区方法的潜在问题是**内存碎片化 Fragmentation**，分为内部和外部碎片化
- **内部碎片化 Internal Fragmentation**指程序被分配到过大的内存空间，剩余的内存无法被其他程序使用，造成浪费
    - **外部碎片化 Externel Fragmentation**指剩余内存被分为很多个小片，剩余内存的总空间足够分配给其他程序，但是单个内存切片不能满足程序需求

Manage Memory within Processes

- 我们见识到了从操作系统层面管理不同程序的内存，现在我们需要一个能够管理程序内部内存的方法
- 在程序内部，stack用作创建local variables和存储local addresses
    - Heap被用来创建动态变量，即heap的分配是动态的
- E.g. In UNIX, process space is divided into:
- **Text segments**: Read-Only, contains code. May have > 1 text segments

- **Initialised Data**: Global data initialised from executable file. `char *msg[] = "Hello World!"`
- **BSS Segment**: Contains uninitialised globals
- **Stack**: Contains statically allocated local variables and arguments to functions, as well as return addresses
- **Heap**: Contains dynamically allocated memory

Manage Free Memory

- 由于程序不断索要和释放内存空间，使得空余内存空间不是一整块，我们需要知道空余内存在哪里：
- 两种方法，**Bit Maps**和**Free/Allocated List**
- 在两种方法中，我们都将内存空间分为固定大小的块，被称为**allocation units**

Bit Map

位图Bit Map是一种数据结构，用于跟踪内存块的使用情况。在这个结构中，内存被分成固定大小的块，每个块由位图中的一个位（bit）表示。如果一个位设置为0，它表示相应的内存块是空闲的；如果设置为1，则表示内存块被占用。

- **优点**：
  - 位图在表示大量内存块时非常高效，因为每个内存块只需要一个位来表示其状态。
  - 位操作通常非常快速，尤其是在现代计算机系统上。
- **缺点**：

- 位图的缺点是它不够灵活，每个块的大小是固定的，这可能导致内存碎片问题，特别是如果块大小和实际请求不匹配时。
- 对于非常大的内存区域，位图自身可能也会占用相当多的内存空间。

Free/Allocated List

- 使用单链表或双向链表实现
  已分配链表是另一种内存管理技术，它使用链表数据结构来跟踪空闲内存块。链表中的每个节点代表一个空闲内存块，包含块的起始地址和大小。
- **优点**：
  - 已分配链表允许更加灵活的内存分配，因为它可以精确地跟踪每个空闲块的大小，所以可以根据需要分配不同大小的内存块。
  - 它可以通过合并相邻的空闲块来减少内存碎片。
- **缺点**：
  - 搜索合适的空闲块可能比较慢，尤其是当空闲列表很长时，可能需要遍历整个列表来找到足够大的空闲块。
  - 空闲列表可能需要额外的存储空间来维护链表结构。

Allocation Policies

- Bit Map和已分配链表只告诉我们空闲内存的位置，而没有说如何分配他们
- 我们有三种简单的分配方式：
- First Fit：遍历链表或位图，找到**第一个有足够空间**的连续内存地址并分配
  - Best Fit：遍历链表或位图，找到一个**有足够空间且为最小的连续内存地址**并分配

    - Worst Fit: 找到一个**最大的连续内存地址并分配**，这可以减少细微的孔tiny useless holes

- 我们可以对空闲的内存块进行**从小到大或从大到小排序**，以减少搜索时间。但是排序后对相邻内存的合并就会更困难
- 我们还有一种较为复杂的分配方式：**Buddy Sort**
- 这个方法更高效，(better than $O(n)$)
    - Buddy Sort通过将内存地址不断地二分，然后将**有足够空间且为最小的连续内存地址**分配给程序
    - 例如总共有1024bytes内存，程序索要127 bytes
        - Buddy Sort二份内存为两个512，然后将一个512二分为两个256，再将一个256二分为两个128
        - 分配128给程序
    - 如果程序释放此128内存，则会判断是否可以和**相邻的同大小的内存合并**
    - 对于这个例子，释放128的内存使得内存空间变为一个1024的整块

File Systems

- 文件系统掩盖了物理存储设备的复杂性，并提供了一个高效的存储管理方式，并且对文件进行访问权保护。

Files and Directories

- **Logical file organisation**: Most common: **Byte Stream**
  在文件系统中，**字节流（Byte Stream）** 指的是文件数据的一种表示方式，它将文件内容表示为一个连续的、顺序的字节序列。使用字节流的概念，文件系统不需要关心文件数据的具体含义或结构，而是把文件当作一个由单个字节组成的长串数据。
- 文件可以按记录（record）组织，每个记录存储一组相关数据。记录可以是固定大小的，也可以是可变大小的：
  - **固定大小记录**：每个记录占用相同数量的字节，这简化了记录的查找和访问，因为每个记录的起始位置可以轻易计算出来。
  - **可变大小记录**：记录根据存储的数据量有不同的大小。这种方式更灵活，但访问特定记录可能需要额外的信息或遍历整个文件。
  - 文件可以隐式地址化Implicitly address或显式地址化Explicitly address:
  - **隐式地址化（Sequential Access）**：文件可以按照顺序访问，每次访问下一个记录。这种方式简单，适用于需要顺序处理文件的场景。
    - **显式地址化**：可以通过以下方式显式访问记录：
      - **按位置（Record Number）**：可以直接根据记录的编号访问，这通常适用于固定大小记录。
      - **按键（Key）**：特别是在记录有特定键（如数据库中的主键）的情况下，可以通过键来查找和访问记录。

Directory Management

- 目录管理的主要问题：

```
1. **数据的结构（Shape）**：
    - 文件系统的数据结构可以采取多种形式，如树形结构、图形结构等。这影响了文件和目录的组织方式，以及用户如何导航和管理文件。
2. **关于文件的信息（What）**：
    - 决定存储哪些元数据来描述文件，例如文件大小、创建和修改日期、权限、所有者信息等。
3. **信息存储位置（Where）**：
    - 确定这些信息存储在文件系统的哪个部分。通常，这些信息存储在文件的元数据区域，可以是文件头部、专门的目录节点（inode）或其他数据结构。
4. **如何组织条目以提高效率（How）**：
    - 如何在目录中组织文件和目录条目，以便可以快速地插入、删除和搜索文件。
```

File Directories

- 文件系统结构一般为**树形结构（Tree-structured）**：
- 树形结构是一种常见的文件系统结构，它模拟了现实世界的文件夹和文件组织方式。
    - 文件系统的每一项（可以是文件或目录）都位于一个树状的层次结构中，每个项有**一个父目录**（除了根目录），并且可以有**多个子项**。
- **树形结构（Tree-structured）的特点有**：
- **简单的搜索、插入和删除操作**：
    - 树结构可以通过路径名来定位任何项，这使得搜索（查找文件）、插入（添加新文件）和删除（移除文件）操作相对简单。
    - **非对称共享**：

    - 在纯粹的树形结构中，每个文件或目录只有一个父节点。这意味着共享（如创建快捷方式或链接）是非对称的，文件通常只能在一个地方拥有**“真实”位置**。

- 需要在文件路径中保留什么信息?

```
1. 所有描述性信息：包括大小，创建日期等
    - 这会使目录结构变得特别大，并且使搜索变得困难
2. 只保留文件名和指针
    - 这会使在每个描述符中都要有磁盘读写
```

- 目录可以有固定的大小，也可以设计为可扩展的：
- **固定大小**的目录简化了管理，但可能限制了目录能包含的最大文件数。
    - **可扩展大小**的目录更加灵活，可以随着文件的添加而增长，但可能需要更复杂的管理策略以处理目录大小的变化。

Basic File System

打开/关闭文件：

1. **打开文件**：
    - 打开文件的过程包括检索文件的相关信息，并为文件访问设置必要的数据结构。这些信息通常存储在文件描述符或索引节点（`i-node`）中。
    - 为了有效地访问文件，操作系统会在打开文件时获取文件描述符，并可能在内存中管理一个打开文件表，以跟踪哪些文件当前是打开状态。
2. **关闭文件**：
    - 关闭文件意味着完成对文件的访问，并更新系统中的相关数据结构，如关闭文件表中的项，并可能更新文件的最后访问时间。

    - 关闭文件操作确保所有的数据都正确写入存储，并释放相关的系统资源。

文件描述符 file descriptor (i-node in UNIX) 包含以下信息：

- Owner id
- File type
- Protection information
- Mapping to physical disk blocks
- Time of creation, last use, last modification
- Reference counter

打开文件表（Open File Table） 是操作系统用来管理所有当前打开文件的数据结构。每当一个进程打开一个文件时，操作系统会在这个全局表中创建一个条目，以跟踪文件的重要信息和状态。打开文件表是操作系统用来实现文件访问控制和文件共享的机制之一。
打开文件表通常包含以下信息：

- **文件描述符（File Descriptor）** 或 **文件句柄（File Handle）**：这是一个唯一标识符，关联到打开文件表中特定的条目。
- **文件位置指针**：它记录了文件内的当前位置，用于读写操作的下一次开始位置。
- **文件打开模式**：例如，只读、只写或读写模式。
- **文件的访问权限**：这些权限决定了哪些操作是允许的。
- **文件锁定状态**：指示文件或其部分是否被锁定，以及锁定的类型。
- **引用计数**：记录有多少进程正在使用这个文件。每当有新的进程打开同一文件时，引用计数会增加；当进程关闭文件时，计数减少。
- **指向文件控制块或i-node的指针**：这包含了文件所有的元数据，如文件大小、所有者、权限以及文件数据在磁盘上的位置等。

当程序执行打开文件操作时，操作系统首先检查文件的权限，然后在打开文件表中创建或更新一个条目，并将文件描述符返回给程序。程序随后使用这个文件描述符来进行读写等操作。
当文件关闭时，操作系统会更新打开文件表，并减少引用计数。当引用计数降至零时，操作系统会清理该条目，释放资源。

Organising Data On a Disk 在磁盘中管理数据

- 磁盘中的第一个分区应该为**Boot**，这个分区包含了启动操作系统的代码，这个分区在PC中也叫作MBR(Mater Boot Record)，这个分区也包含了分区表
- 接下来的分区为文件系统File Systems，在PC中，最多可以分为4个**主分区Primary Partition**和多个**逻辑分区Logical Partition**

Organising Files

- 在计算机存储中，扇区（Sector）和逻辑块（Logical Block）是数据存储和访问的两个基本单位，它们之间有着密切的关系：
- **扇区 Sector:**
  - 扇区是硬盘驱动器（HDD）或固态驱动器（SSD）上的基本物理存储单位。
  - 在传统的硬盘上，一个扇区通常存储512字节数据；现代硬盘通常使用4096字节（4KB）作为一个扇区的大小。
  - 扇区是磁盘读写操作的最小物理单位。
  - **逻辑块（Logical Block）或文件系统块（Filesystem Block）：**
  - **逻辑块(数据块)** 是文件系统管理数据的逻辑单位。

    - Logical blocks need not to be same as physical sector size - may be some multiple of sectors
    - 文件系统将一组扇区逻辑地组合成一个块，块的大小通常是扇区大小的整数倍。
    - 在文件系统中，块是分配空间和管理文件的基本单位。
- 我们需要一种数据结构去记录哪个逻辑块为哪个文件的一部分
- 经典的数据结构：list，trees和arrays

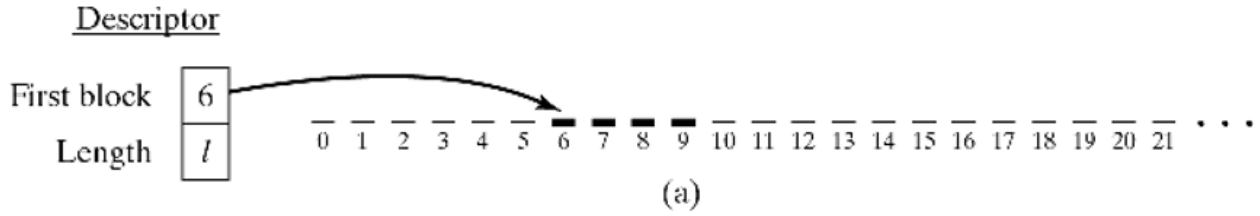
Physical Organisation Methods

Contiguous Organisation

连续组织（Contiguous Organization）是文件系统中文件存储的一种方法。在这种组织中，文件的所有数据块在磁盘上物理位置上是连续的。这意味着文件的起始块和结束块之间的所有块都是连续存放的，没有空隙。
连续组织的特点包括：

- **简单的访问**：因为文件的数据块是连续的，所以读取文件时可以一次性读取整个文件，这使得文件的访问速度非常快，特别是对于读取大文件时。
- **高效的磁盘利用**：连续的数据块减少了寻道时间，因为磁头不需要在不同的磁道之间移动来读取同一个文件的不同部分。
  连续组织的缺点：
- **碎片问题**：随着文件的创建和删除，磁盘上会出现碎片。新文件可能找不到足够大的连续空间，即使磁盘上总的空闲空间是足够的。
- **文件扩展问题**：如果一个文件需要扩展，但其后面的磁盘空间已被占用，该文件就需要移动到一个有足够连续空间的新位置。

- **预分配问题**：在创建文件时，必须预先知道文件的最大大小，这样才能分配连续的空间，这在许多情况下是不现实的。

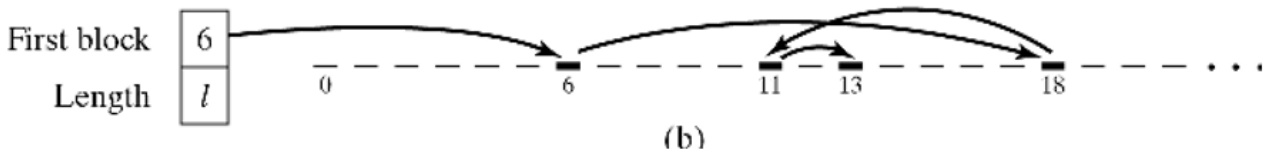


Linked Organisation

链式组织（Linked Organization）是文件系统中文件存储的一种方法，其中文件的各个部分分散存储在磁盘上的不同区域，每部分通常被称为一个数据块或簇。在链式组织中，每个数据块都包含了指向文件下一个数据块的指针，形成了一个链条。这意味着，文件系统只需知道文件的起始块，就可以通过跟踪指针从一个数据块移动到下一个数据块，从而访问整个文件。
链式组织的特点：

- **非连续存储**：文件的数据块可以散布在磁盘的任何地方，不需要物理上连续。
- **动态扩展**：文件容易扩展，只需要在链的末尾添加新的数据块并更新前一个数据块中的指针。
- **无需预分配空间**：不需要在文件创建时就预知文件的最终大小，文件可以随着数据的添加而增长。
  链式组织的缺点：
- **访问速度较慢**：由于数据块可能分散在磁盘的各个部分，因此读取文件时可能需要多次寻道操作，这可能会导致访问速度较慢。
- **指针占用空间**：每个数据块都需要额外的空间来存储指向下一个数据块的指针。

- **可靠性问题**：如果链中的某个指针损坏，可能会导致整个文件的剩余部分无法访问。



(b)

Physical Organisation Methods: FAT

- FAT是**链式分配Linked Organisation**的一种形式，但与传统的链式分配不同，它不是将指针存储在数据块中，而是将整个链表存储在一个称为FAT的特殊表中。
- 在这个系统中，文件的每个数据块在FAT表中都有一个对应的条目。这个条目包含了文件的下一个数据块的编号或特殊代码。
- **FAT表存储在内存中：**
- 虽然FAT表在磁盘上有物理副本，但为了快速访问，整个FAT表在系统启动时会被读取到RAM（随机访问存储器）中。
    - 将FAT存储在RAM中可以显著提高访问文件数据块的速度。
- **FAT表的内容：**
- **块编号（Block Number）**：如果文件占用了多个数据块，FAT表中的每个条目会指向文件的下一个数据块。
    - **文件结束（EOF）代码**：表示文件结束的特殊代码，相当于链表中的NULL指针，标志着文件的最后一个数据块。
    - **空闲（FREE）代码**：表明一个数据块当前未被使用。
    - **坏块（BAD）标记**：表示数据块不可用，可能是因为磁盘错误。
- **FAT的混合机制：**

- FAT结合了位图Bit Map（用于跟踪空闲数据块）和链式分配Linked Organisation（用于管理文件的数据块列表）的特点。
    - 它通过数组方式管理空间，每个数据块在FAT中都有一个对应的条目。

FAT Directory

- **目录作为特殊文件：**
- 在DOS中，目录被视为一种特殊类型的文件（类型为目录），它包含了一系列的目录项。
    - 每个目录项是一个32字节的结构，使用小端字节序（Little Endian）格式存储信息。
- **FAT目录项结构：**
- **文件名和扩展名**：目录项的前11字节用于存储文件名（8字节）和扩展名（3字节）。
    - **文件属性**：其中一部分字节用于存储文件属性，这些属性可能包括：
        - 只读（Read-Only）
        - 隐藏（Hidden）
        - 系统文件（System）
        - 目录标志（Directory Flag），用来标识该条目是一个文件还是一个目录
        - 存档（Archive），通常用于备份和文件变更跟踪
        - 卷标（Volume Label），标识磁盘卷的名称
    - **时间和日期**：文件创建、上次访问以及上次修改的时间和日期信息。
    - **第一个块（Cluster）号**：文件数据开始的地方，即第一个数据块或簇的编号。
- **根目录的特殊性：**

- 根目录在FAT文件系统中有特殊的地位，因为它的位置是已知的（即在磁盘的固定位置）。
    - 在FAT16文件系统中，根目录的大小是固定的，限制为512个条目。这意味着根目录可以直接存储512个文件或目录的信息，而不需要任何扩展。
    - 其他目录则不受此限制，它们的大小可以动态变化，并且它们的条目可以通过目录类型的属性来区分。

FAT Organisation

四种类型的FAT条目：

1. **块编号**：

> - 如果一个条目是块编号，那么它指向文件的下一个数据块。这些编号形成了一个链表，指示文件在磁盘上的存储顺序。

2. **EOF（文件结束）字节**：

> - 特殊的EOF标记表示文件数据块链表的结束。当FAT中的一个条目被标记为EOF时，它表示当前块是文件的最后一个数据块。

3. **FREE（空闲）字节**：

> - 如果一个条目被标记为FREE，那么相应的数据块当前没有被使用，可用于存储新数据。

4. **BAD（坏块）字节**：

> - 如果一个条目被标记为BAD，那么相应的数据块由于磁盘错误而不可用。

文件和块的链接：

- 文件的数据块在FAT中通过块编号链接起来。例如，文件A可能由FAT中的第3个块开始，第3个块的FAT条目指向第5个块，第5个块的条目又指向第8个块，形成链表3→5→8。
  空闲块和坏块：
- 空闲块通过FAT条目中的FREE值来表示。文件系统在需要分配新块时会查找这些FREE标记的条目。
- 坏块通过FAT条目中的BAD值来标识。这些块不会被文件系统用来存储数据，因为它们可能会导致数据损坏。

FAT Operation

Delete 操作：
当在MS-DOS中执行删除操作时（例如使用 `del` 命令），文件系统通常执行以下步骤：

1. **标记FAT条目**：在文件分配表（FAT）中，文件的第一个簇（数据块）的条目被标记为“空闲”（FREE）。这意味着文件系统不再认为这些簇属于任何文件，因此它们可以被用来存储新数据。
2. **更新目录项**：文件的目录项中的文件名首字符通常被替换为一个特殊的删除字符（如0xE5），这样文件就不会在目录列表中显示出来。
   Undelete 操作：
   由于MS-DOS不会立即清除文件内容，而是只标记FAT中的簇为“空闲”并修改目录项，因此在新数据写入磁盘覆盖这些簇之前，文件通常可以被恢复。恢复文件的基本步骤包括：
3. **识别删除的文件**：通过扫描目录结构找到被标记为删除的文件项。
4. **检查FAT**：检查与该文件关联的簇在FAT中是否仍然是连续的，且没有被其他文件覆盖。如果这些簇未被覆盖，文件内容还在磁盘上。
5. **恢复目录项和FAT**：将目录项中的文件名首字符从删除字符恢复为原始字符，同时在FAT中恢复文件簇链的链接。

FAT16

FAT16 文件系统结构：

- 在FAT16文件系统中，每个FAT条目是16位的，这意味着FAT表中每个条目可以指向65,536（即2^16）个不同的**簇cluster(数据块block)**。
- **扇区大小sector size** 通常设置为512字节，这是硬盘存储的基本单位。
  处理大于32MB磁盘的问题：
- 由于每个FAT条目是16位的，FAT16理论上可以直接管理的最大磁盘大小是32MB（即64K个簇乘以每个扇区512字节）。要管理更大的磁盘，需要增加每个簇包含的扇区数，即增加簇的大小。
  簇大小：
- 簇是文件存储的逻辑块，它由多个扇区组成。MS-DOS将这个逻辑块称为簇。
- 簇的最大大小通常为32KB，这意味着每个簇可以包含64个扇区（因为每个扇区512字节）。
  文件系统和文件大小限制：
- 如果每个簇为32KB，FAT16文件系统的最大容量为2GB（即64K个簇乘以每簇32KB）。
- 单个文件的最大大小略小于2GB，因为文件大小也受到簇大小和FAT条目数量的限制。
  簇大小与内部碎片：
- 如果簇的大小较大，会导致较大的内部碎片，因为即使文件只比一个簇小一字节，也需要分配整个簇来存储它。
  FAT16 的簇大小扩展：
- 尽管32KB是FAT16文件系统的常规最大簇大小，但在某些操作系统实现中，簇大小可以增加到64KB。这将文件系统的最大限制扩大到4GB，但会进一步增加内部碎片，并可能会导致与某些系统的兼容性问题。

VFAT

VFAT（Virtual File Allocation Table）是FAT文件系统（如FAT16）的扩展，它添加了对长文件名的支持。长文件名在Windows 95及以后的版本中得到了广泛使用。VFAT允许文件名最多达到255个字符，而FAT16只支持最多8个字符的文件名和最多3个字符的扩展名，即所谓的“8.3”命名约定。

如何实现长文件名支持：

- **与FAT16的兼容性**：VFAT通过在目录项中存储额外的信息来支持长文件名，同时保留了一个标准的FAT16短文件名条目。这样做确保了旧软件和操作系统仍然可以访问VFAT格式化的磁盘，即使它们不识别长文件名。
- **创建短文件名**：为了保持与FAT16的兼容性，VFAT自动为每个长文件名创建一个符合“8.3”命名约定的短文件名。这个短文件名是从长文件名派生的，可能包含一些特殊字符和数字来确保唯一性。
- **存储长文件名**：长文件名被分割成多个部分，并存储在一系列的目录项中，这些目录项在FAT16中使用非法属性标记，以防止FAT16系统将这些目录项当作常规文件或目录项处理。
- **多个目录项**：一个长文件名可能需要多个连续的目录项来完整存储。这些目录项中的每一个都包含文件名的一部分，并以特定方式链接起来，以便操作系统可以将它们重组为完整的文件名。

面临的挑战：

- **两种文件名的管理**：VFAT需要管理两种文件名，即短文件名和长文件名。这可能会在文件系统操作中引入复杂性。
- **别名问题**：因为每个文件有两个名字，可能导致别名问题，即两个不同的长文件名可能映射到相同的短文件名。
- **旧软件兼容性**：只认识短文件名的旧软件可能会遇到问题，因为它们可能无法正确处理长文件名。

FAT 32

FAT32是FAT文件系统系列中的一个版本，它扩展了FAT表的大小，增加了文件系统能支持的最大容量，并对一些限制进行了改进。

FAT32的特性：

1. **FAT表大小**：
    - FAT32使用28位来索引簇（尽管总位数是32位，但实际只使用28位），这意味着FAT表可以跟踪更多的簇。这允许文件系统支持更大的存储设备。
2. **文件系统最大大小**：
    - 随着簇编号的扩展，FAT32文件系统理论上可以支持最大到127GB（甚至更大，取决于簇的大小）的存储设备。
3. **内部碎片减少**：
    - 由于可以支持更多的簇，FAT32允许使用更小的簇大小，从而减少了内部碎片——即未使用的存储空间，这通常发生在文件不够大以填满最后一个分配给它的簇。
4. **FAT表大小增加**：
    - 更多的簇意味着FAT表本身的大小会增加，因为需要更多的条目来映射磁盘上的簇。
5. **根目录大小限制**：
    - FAT32移除了FAT16中根目录大小的限制。在FAT32中，根目录被当作一个普通目录处理，可以动态增长，其大小受可用空间的限制。
6. **最大文件大小**：
    - FAT32支持的最大单个文件大小理论上可以达到2^32-1字节（即4GB减去1字节）。这对于大文件的存储提供了更大的灵活性。

Disk Fragmentation (From Hardware)

磁盘碎片（Disk Fragmentation）是文件存储在硬盘上时出现的一种现象，它会影响硬盘访问速度和整体性能。

连续块（无碎片）的优势：

- **连续块的快速访问**：如果一个文件的数据块在磁盘上是连续的，从几何和处理的角度来看，磁盘访问速度会更快。这是因为磁盘读写头不需要在不同的位置之间移动，可以一次性顺序读取所有数据块。
- **同一柱面上的块**：如果数据块位于硬盘的同一柱面上，那么磁盘的寻道时间会大大减少，从而提高数据访问速度。

  碎片产生的过程：
- 在文件系统刚安装完毕时，数据可能会被选项地分配在磁盘上，这意味着文件会被存储在连续的数据块中。这时的文件系统（如新安装的FS）可能表现出优异的性能。
- 随着时间的推移，用户可能会删除一些文件和数据块，同时也会插入新的文件和数据块。这些操作会导致磁盘上的数据块分布变得更加随机和分散。
- 经过一系列的删除和写入操作后，文件的数据块可能不再连续，变得“分散”在磁盘上，这就是磁盘碎片。

  磁盘碎片的影响：
- 碎片的产生意味着逻辑上连续的数据块在物理磁盘上实际上是“相隔甚远”的。这与内存碎片不同，内存碎片是由于分配和释放内存时产生的未使用空间的小块造成的。
- 磁盘碎片化导致硬盘读写头需要在磁盘上移动到多个不同的位置来读取一个文件，增加了寻道时间和旋转延迟，从而降低了数据访问速度。

  如何处理磁盘碎片：
- 在FAT中，碎片化在较大的簇（数据块）中影响较小
- 但这会引起更大的内部碎片化
- MSDOS的解决方案：

- 在整个文件系统上运行去碎片化程序，即移动所有的数据块，使他变成连续的
    - 这样在运行完成之后就会有一整个大的空闲数据块
    - 这可能会运行很长时间

Indexed Organisation

索引组织（Index Organization）是文件系统中用于管理文件存储的一种方法。在这种组织中，每个文件都有一个索引结构（如索引节点或inode），索引中包含了文件所有数据块的指针。这种方法允许文件的数据块在磁盘上非连续存放，同时仍然能够有效地访问文件的全部内容。
索引组织的特点：

1. **索引结构**：
    - 文件的索引结构通常存储在一个固定的位置，例如inode表中。
    - 索引中的每个条目对应文件的一个数据块或一组数据块。
    - 索引可以直接指向数据块，或者指向其他索引结构，这取决于文件的大小和文件系统的设计。
2. **直接和间接指针**：
    - 直接指针直接指向文件的数据块。
    - 大型文件可能还会使用间接指针，这些指针指向其他包含指针的块，这些指针再指向实际的数据块。
3. **支持大文件**：
    - 通过使用多级索引（如单级间接、双级间接、三级间接指针），索引组织可以支持非常大的文件。
4. **随机访问**：

    - 由于文件的每个数据块都可以通过索引快速定位，因此支持高效的随机访问。
      索引组织的优势：

- **灵活性**：文件的数据块可以存储在磁盘的任意位置，不需要连续的空间。
- **效率**：对于小文件，通常只需要访问索引和少数几个数据块；对于大文件，虽然需要通过多个索引层次，但通常仍然比链式组织更高效。
- **可扩展性**：索引结构允许文件动态增长。
  索引组织的缺点：
- **复杂性**：管理索引结构比链式组织或连续组织更复杂。
- **开销**：索引本身需要占用一定的磁盘空间，尤其是对于非常大的文件系统。

Varieties of indexed organisation 索引组织的变体

索引组织的变体涉及了不同的技术来管理文件在磁盘上的存储，尤其是大文件和数据库系统。这些变体的目的是在提高存储效率和减少访问延迟之间找到平衡。
多级索引层次结构（Multi-level Index Hierarchy）：

- 在多级索引结构中，一个主索引（Primary Index）不直接指向文件的数据块，而是指向次级索引（Secondary Indices），这些次级索引然后指向实际的数据块或进一步的索引层次。
- **优点**：这种方法允许文件系统管理非常大的文件，因为通过添加更多的索引层次可以极大地增加文件系统的最大文件大小。
- **缺点**：随着索引层次的增加，读取文件的某个部分可能需要更多的磁盘访问，因为每一级索引都可能涉及到一个磁盘I/O操作。
  增量索引（Incremental Indexing）：
- 在增量索引方法中，顶级索引有固定数量的条目，这些条目直接指向文件的数据块。

- 当顶级索引的条目不足以包含所有数据块的指针时，文件系统会分配额外的索引层次来存储更多的指针。
- **优点**：这种方法在文件较小时保持了效率，只有当文件增长到需要额外索引层次时，才增加额外的开销。
- **缺点**：与多级索引层次结构类似，增量索引也可能导致随着文件的增长，磁盘访问的次数增加。

UNIX File Systems

System V文件系统（S5FS）的主要组成：

1. **索引节点（inodes）**：
    - 每个文件和目录在UNIX文件系统中都有一个对应的inode。inode包含了文件的元数据，如所有者、权限、大小、时间戳（创建、访问和修改时间）以及指向文件实际数据的指针。
    - inode编号是唯一的，它在文件系统中标识一个特定的文件。
2. **目录（directories）**：
    - 目录在UNIX文件系统中是特殊类型的文件，它们包含文件名和对应的inode编号。
    - 目录项将文件名映射到inode，这使得文件系统可以根据名称找到文件的inode，进而访问文件数据。
    - 目录可以包含硬链接（hard link）和符号链接（symbolic link）。硬链接直接关联到文件的inode，而符号链接则包含了指向另一个文件路径的文本指针。
3. **文件分配**：

- UNIX文件系统使用多级索引树来分配文件数据。每个文件的inode包含直接指针（直接指向数据块的指针），以及一级、二级和三级间接指针（指向包含其他指针的块的指针）。这种结构允许文件系统有效地管理大文件和大量的小文件。

System V文件系统（s5fs）中的索引节点（inodes）是文件系统核心概念之一。它们代表了文件系统中的每个文件，并且承载了文件的大部分元数据。
inode的特性：

1. **文件的实际对象**：
   - inode代表了文件系统中的实际文件对象。每个文件在文件系统中都有一个唯一的inode。
2. **一对多映射**：
   - 由于硬链接的存在，多个文件名（位于不同的目录项中）可以映射到同一个inode。这意味着多个文件名可以引用同一个文件内容和属性。
3. **包含的元数据**：
   - inode包含了关于文件的所有元数据，除了文件名。这些元数据包括文件大小、所有者、权限、时间戳（文件创建、最后访问和修改时间）等。
   - inode还包含**指向文件数据所在磁盘块的指针**。这些指针构成了文件的“目录表”（**Table of Content, TOC**）。
4. **引用计数**：
   - inode包含一个引用计数，即指向该inode的硬链接数。当引用计数降至零时，意味着没有任何目录项引用这个inode，文件可以被删除。这当然也受限于文件是否被打开——即所有指向该文件对象的文件描述符都已关闭。
5. **每个文件的TOC**：

- 与MS-DOS的FAT（全局文件分配表）不同，s5fs中的每个inode都有自己的TOC，这表示文件数据在磁盘上的位置。这提供了更灵活的文件数据管理方式，允许文件系统更有效地处理大文件和小文件。

TOC

System V文件系统（s5fs）中的“目录表”（Table of Content, TOC）是inode结构的一部分，它详细指出了文件的各个部分在磁盘上的位置。TOC使用了直接和间接的指针来定位文件的数据块。
TOC索引块的类型：

1. **直接块指针**：
   - 用于小文件的存储，因为直接块指针直接指向包含文件数据的磁盘块。
   - 对于直接块，没有额外的磁盘开销，文件访问效率高。
2. **单级间接块**：
   - 用于大于直接块容量但小于双级间接块容量的文件。
   - 磁盘开销是一个块的大小，因为需要一个**额外的块来存储指向实际数据块的指针**。
   - 文件访问速度比直接块慢，因为需要额外的步骤来解引用间接块。
3. **双级和三级间接块**：
   - 用于大于单级间接块容量的文件。
   - 磁盘开销更大，因为需要多个间接块来存储额外级别的指针。
   - 随机文件访问需要查找多个间接块，这使得访问速度比单级间接块慢。
   - 文件的随机访问会更慢，因为需要遍历更多的间接指针。

文件大小与索引块关系：

- 文件的大小决定了使用哪种类型的索引块：

- **直接块**适用于最小的文件。
- **单级间接块**用于中等大小的文件。
- **双级间接块**和**三级间接块**用于更大的文件。

S5FS Parameter

S5FS文件系统的参数包括：

1. **直接块的数量**：
   - 文件系统为每个文件分配的直接块的数量，直接块用于存储文件的实际数据。s5fs可能使用10个直接块，而ext2使用12个直接块。
2. **间接层次的数量**：
   - 文件系统支持的最大间接索引层次，通常是2或3。每增加一个层次，都会增加可支持的文件最大大小。
3. **逻辑块大小**：
   - 确定磁盘I/O效率的一个关键参数。逻辑块可以由多个连续的物理块组成。逻辑块的大小还决定了索引块中可以包含的指针数量以及最大间接层次。
   - 例如，ext2文件系统在创建时可以选择1KB、2KB或4KB作为逻辑块的大小。
4. **块指针大小**：
   - 影响索引能力，确定了文件系统可寻址的最大磁盘块。块指针的大小必须足够容纳逻辑块的最大数量。

文件大小与索引结构的关系：

- **小文件**只使用TOC中的直接块，直接块的数量可以根据文件系统的设计而变化。
- **较大文件**需要使用第一个间接块，这类似于单级页表。

- **更大的文件**会使用双级间接块，双级间接块指向间接块，如果文件足够大，可能还会使用三级间接块。

文件系统设计的优化：

- 文件系统设计允许存在“逻辑零填充的空洞”—如果文件的一部分未被实际数据占用，相关的块指针可以设置为NULL。当读取这些逻辑块时，系统将返回零值，这避免了实际分配空间给未使用的数据。

各种文件组织方法的存储方式和占用空间

- **Contiguous Organisation:**
- 需要存储的信息：
  1. **起始地址**（Start Address）
  2. **长度或大小**（Length or Size）：
     - 文件或程序占用的空间大小。这可以是字节数、磁盘块数或内存页数。
     - 起始地址和长度各需要一个disk block number所占用的空间，即如果disk block number = 2 bytes，则总共需要额外的4bytes来存储这些信息
  - 存储这些信息的地方：

```
1. **文件系统的目录项**（对于磁盘上的文件）：
    - 在文件系统中，每个文件的目录项可能包含其在磁盘上的起始地址和长度。这些信息用于定位文件的实际数据。
2. **进程控制块（PCB）**（对于内存中的程序）：
    - 在操作系统中，每个运行中的程序（进程）通常会有一个进程控制块（PCB），它包含了进程的关键信息，包括其在内存中的起始地址和占用的空间大小。
```

  - 每次写入一个文件只需要读写磁盘一次

- **Linked Organisation**:
- 需要存储的信息：
    1. **指向下一个数据块的指针**：
        - 每个数据块中都包含一个指针（或索引），指向文件的下一个数据块的位置。这是实现链式组织的核心。
    2. **文件的起始块地址**
    3. **文件的最终块地址**
    - 这三个数据各需要一个disk block number所占用的空间
  - 存储这些信息的地方
  - **文件系统的目录项**：
    - 文件的目录项或元数据区域会**存储文件起始块的地址**。这允许文件系统找到文件的第一个数据块，并从那里开始遍历整个文件。
    - **数据块本身**：
    - 在链式组织中，**每个数据块通常直接包含指向下一个数据块的指针**。这些指针构成了文件的链式结构。
  - 写入一个文件时，每写完一个数据块，就要额外进行一次磁盘读写，即数据块的数量就是磁盘读写次数（最坏情况下）
- **FAT16:**
- 需要存储的信息：
    - **FAT表**，大小为number of data block * 2, 或者为storage size / cluster (data block) size
    - **文件的起始块地址**
  - 存储位置
  - 引导扇区之后的扇区中，或根目录区域之前
  - 写入一个文件只需读取磁盘一次
- **Indexed Organisation:**

- 需要存储的信息：
  - **索引块：索引块直接占用一整个data block**。索引块包含一系列指针，每个指针指向文件的一个数据块。这些指针构成了文件数据的索引。
    - 该大小取决于指针的大小,即disk block number所占用的空间，如果disk block number=2bytes，disk block = 512，则每个索引块能够存储256个指针。
    - 需要注意的是，如果存在多个索引块，则索引块的末尾需要存储下一个索引块的地址
  - **文件的索引块地址**
    - 文件的索引块地址存储在文件中
    - 每次文件操作需要读取每个索引块，每次读取一个索引块造成一次磁盘读写。以及每个存储文件的数据块都需要一次磁盘操作(最坏情况下)

`fseek`, `fwrite`, `fread`

1. `fseek` 函数：
   - `fseek` 是用于文件定位的函数，通常用于移动文件指针的位置。
   - 它的原型如下：`int fseek(FILE *stream, long offset, int origin)`。
   - 参数 `stream` 是一个指向文件的指针，`offset` 是偏移量，`origin` 是起始位置。
   - `origin` 参数可以取以下值之一：
     - `SEEK_SET`：从文件开头开始偏移 `offset` 个字节。
     - `SEEK_CUR`：从当前文件指针位置开始偏移 `offset` 个字节。
     - `SEEK_END`：从文件末尾开始倒退 `offset` 个字节。

- 函数返回值是操作是否成功的标志，通常为 `0` 表示成功。

2. `fwrite` 函数：
   - `fwrite` 用于将数据块写入文件。
   - 它的原型如下：`size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream)`。
   - 参数 `ptr` 是一个指向要写入的数据的指针，`size` 是每个数据项的大小，`count` 是要写入的数据项的数量，`stream` 是文件指针。
   - 函数返回成功写入的数据项数量。
3. `fread` 函数：
   - `fread` 用于从文件中读取数据块。
   - 它的原型如下：`size_t fread(void *ptr, size_t size, size_t count, FILE *stream)`。
   - 参数 `ptr` 是一个指向存储读取数据的缓冲区的指针，`size` 是每个数据项的大小，`count` 是要读取的数据项的数量，`stream` 是文件指针。
   - 函数返回成功读取的数据项数量。