# CS111 操作系统原理（Operating Systems Principles）学习笔记

> **数据来源**：斯坦福大学 CS111（Spring 2026），讲师 Mendel Rosenblum。 本笔记基于课程公开网站（https://web.stanford.edu/class/cs111/ 及归档站点 cs111.1266）上的公告、课程大纲（Syllabus）、FAQ、考试信息页、全部 9 个作业页面，以及全部 28 讲公开讲义 PDF（Lecture1—Lecture28.pdf）整理编写。 推荐教材：《Operating Systems: Principles and Practice》（2nd Edition），Thomas Anderson & Michael Dahlin。 本文档为个人学习笔记，内容版权归斯坦福大学及其作者所有，仅用于个人学习目的。

---

# 课程概览（Course Overview）

## 1. 课程目标与范围

CS111 是斯坦福大学的操作系统入门课，目标是让学生理解现代操作系统提供的基本设施，并学会"充分利用操作系统与硬件"。课程按主题分为**三大板块**，最后以若干小主题收尾：

| 板块 | 内容 | 对应作业 |
|---|---|---|
| **并发（Concurrency）** | 进程与线程、上下文切换、同步、调度、死锁 | Assign1—Assign4 |
| **内存管理（Memory Management）** | 链接、动态存储分配、动态地址翻译、虚拟内存、请求调页 | Assign5—Assign6 |
| **文件系统（File Systems）** | 存储设备、磁盘管理与调度、目录、保护、崩溃恢复 | Assign7—Assign8 |
| **补充主题** | 闪存（Flash Memory）、虚拟机（Virtual Machines） | — |

课程的核心思想（Lecture 28 总结）： – **虚拟化（Virtualization）**：把一样东西变成另一样东西，或变成许多个——CPU→线程、存储→文件、主存→地址空间。 – **并发管理（Managing Concurrency）**：同步是系统中最难的部分之一。 – **原子性（Atomicity）**：让一组操作看起来像一个不可分割的操作——同步、文件系统一致性都依赖它。 – **局部性（Locality）**：过去往往能预测未来——调度、TLB、分页、文件缓存都建立在这一假设上。 – **分层（Layering）**：用高层抽象隐藏底层复杂细节。

## 2. 课程结构

- **授课**：周一/周三/周五 11:30AM—12:20PM，Nvidia Auditorium（课程录像通过 Canvas 提供）。
- **Section（习题课）**：每周 50 分钟，由 CA 带领 10—15 名学生做练习；需在课程网站报名。

- **作业**：9 个个人作业，在 myth 集群（ssh 远程 Linux 工作站）上完成，使用 gcc/g++、make、gdb、valgrind 等工具；作业通常周四 11:59PM 截止。
- **评分**：作业 35% + Section 参与 5% + 课堂参与 5% + 期中 20% + 期末 35%。
- **考试**：
- 期中：5 月 7 日（周四）19:00–21:00，Cemex Auditorium，允许带 2 张双面笔记纸。
- 期末：6 月 10 日（周三）8:30–11:30，Nvidia Auditorium，允许带 3 张双面笔记纸。
- 均为闭卷纸笔考试，按座位就座（Academic Integrity Working Group 监考试点）。

## 3. 课程作业总览（全部有公开页面）

| 作业 | 主题 | 要点 |
|---|---|---|
| Assign0 | Welcome to CS111! | 熟悉 myth 环境、gdb、sanitycheck/submit 工具；复习 C/C++、STL；读代码 + 写少量代码。无迟交。 |
| Assign1 | Lambdas, Threads, and Processes | 用 λ 表达式创建匿名函数；创建线程（同进程内）与进程（fork）；实验原子操作对并发行为的影响。 |
| Assign2 | Synchronization | 用 **monitor 模式**实现两个同步问题：Caltrain 乘客上车、派对宾客分组；识别竞态与死锁。 |
| Assign3 | Thread Dispatcher | 在**用户态**用 C++ 实现线程机制：在单个系统线程上调度任意多个用户级线程（各自独立栈、定时器中断实现 round-robin）。 |
| Assign4 | Locks & Condition Variables + Trust | 在 Assign3 基础上实现 Mutex 和 Condition（单核系统）；并回答关于信任的伦理问题。 |
| Assign5 | Memory-Mapped Encrypted Files | 把加密文件映射进虚拟地址空间：**请求调页**——捕获 page fault、按需读页、记录脏页、关闭时回写；用 mprotect 模拟硬件特性。 |
| Assign6 | Page Replacement with the Clock Algorithm | 扩展 Assign5，让映射文件可以大于物理内存：实现 Clock 页面置换算法。 |
| Assign7 | Reading Unix V6 Filesystems | 用 C 语言实现 Unix V6（1975 年）文件系统读取器：块层 → inode 层 → 文件层。 |
| Assign8 | Journaling File System | 用 FUSE 挂载 V6 文件系统：为元数据更新加**预写日志（write-ahead log）**、用位图替代链表管理空闲块、崩溃后重放日志恢复一致性。无迟交。 |

---

# Lecture 1: Welcome to CS111! / 操作系统导论

# 概述

本讲回答"什么是操作系统"，并采用**历史视角**：操作系统是为了解决真实问题而演化出来的。通过回顾 1940 年代至今的硬件与软件变迁，理解内核（kernel）为何存在、它管理哪些资源，以及为什么操作系统是计算机科学中最有趣、最综合的领域之一。

# 核心概念与系统机制图解

## 操作系统（Operating System）

- **定义**：管理共享资源（CPU、内存、I/O 设备、文件、网络）并保护参与者互不干扰的软件集合；实现这些功能的内核代码运行在**特权模式**下，占据内存中的保留区域，对整个系统有完全控制权。
- **直观解释**：操作系统是"资源大管家"——它决定谁在什么时候能用 CPU、能用哪块内存、能读写哪个文件，同时保证一个坏应用不能拖垮整个机器。
- **为什么值得学**：操作系统是"魔法"，学完后你能理解计算机真正的工作方式；它整合了并发、虚拟内存等有趣概念，并引出哲学问题（如"公平比整体满意度更重要吗？"）。

## 操作系统演化简史（三阶段）

```
1940s–1960s: 硬件贵、人便宜 → 目标:最大化机器利用率
  ├─ 简单批处理监控器(batch monitor): 读入一叠卡片作业, 顺序执行
  ├─ 1960s: 数据通道与中断 → I/O 与计算重叠
  └─ 1962 IBM 7094: 内存重定位 + 内存保护 → 多任务(multitasking)与内核(kernel)出现
1960s–1980s: 硬件便宜、人贵 → 目标:最大化人的效率
  ├─ 分时系统(timesharing): 交互式使用, 文件系统, 响应时间与抖动问题
  └─ 个人电脑: 一人一机
1990s–今天: 网络与无处不在的计算
  ├─ 1993 WWW → 网络: 机器间共享与通信
  ├─ 手机、电视、智能设备
  └─ 现代系统: 极小(设备)到极大(数据中心/云)
```

- **批处理监控器的问题**（推动演化的动力）：
- 一次只能跑一个作业——作业等待 I/O 时 CPU 闲置，利用率差；
- 没有保护——坏作业能破坏监控器本身；
- 短作业被长作业堵住——理想情况应能重排作业。

## 现代操作系统做什么（Lecture 1 清单）

- CPU：**并发**——让多个任务共享处理器；
- 内存：在进程间共享内存；
- I/O 设备：高效管理设备操作；

- 文件：跨用户共享存储；
- 网络：让多台计算机协作；
- 安全：保护参与者互不干扰。

### 计算机系统分层结构

```
+---------------------------------------+
|    Application   Application   App    |
+---------------------------------------+
|   Operating System Kernel             |
|   (进程管理 / 内存管理 / 文件管理)      |
+---------------------------------------+
|   Hardware: CPU, DRAM, Disks          |
+---------------------------------------+
```

## 代码示例与系统调用解说

操作系统课的代码从 `hello world` 开始——但本讲的"代码"其实是**课程地图**。以 Assign0 的典型流程为例：

```
# 登录 myth 集群
ssh myth.stanford.edu
# 克隆作业起始代码
git clone /afs/ir/class/cs111/repos/assign0/$USER assign0
# 构建
make
# 运行测试
./sanitycheck
# 调试
gdb ./myprogram
# 提交
./submit
```

**【代码做了什么？】** 建立从"源码 → 编译 → 测试 → 调试 → 提交"的完整作业流水线；`make` 调用 gcc/g++，`gdb` 提供断点/单步调试，`valgrind` 检测内存错误。

**【系统机制透视】** 这条流水线本身就是操作系统的缩影：`ssh` 创建一个远程进程并通过网络与它通信；`git clone` 走文件系统与网络协议栈；`make` 依赖进程创建（fork/exec）与文件系统；`gdb` 用 `ptrace` 系统调用观察和控制另一个进程——你在第一周就已经在使用进程、文件、网络和虚拟内存四大抽象了。

## 关键要点

1. 操作系统是历史演化的产物：先解决机器利用率（批处理），再解决人的效率（分时），再解决连接（网络）。

2. 内核 = 运行在特权模式、管理全部硬件资源的代码；用户应用代码受到限制。
3. 现代 OS 的核心职责可归纳为：CPU 并发、内存共享、I/O 管理、文件共享、网络安全。
4. 课程三大板块：并发（4 个作业）、内存管理（2 个作业）、文件系统（2 个作业）。
5. 虚拟化（一个物理资源伪装成多个逻辑资源）是本课程贯穿始终的主题。

## 常见陷阱与注意事项

- **把"操作系统"与"内核"混为一谈**：内核是 OS 的核心部分；广义的 OS 还包括 shell、系统工具、库等。
- **以为 OS 只是"调度器"**：现代 OS 同时是资源管理器、保护者、抽象提供者。
- **忽视历史**：不理解批处理/分时的痛点，就很难理解为何需要虚拟内存和并发抽象。

## 思考题

1. **问题**：为什么说"内存保护 + 重定位"是内核出现的先决条件?
2. **答案**：没有保护，一个作业就能破坏其他作业或监控器代码，无法安全地让多个作业同时驻留内存；没有重定位，多个程序无法加载到各自的内存区域。二者齐备后，多任务才成为可能，而管理多任务的特权代码就是内核。
3. **问题**：虚拟化思想在"线程"和"文件"两个抽象中分别体现在哪里?
4. **答案**：线程把一个物理 CPU 虚拟化成多个可并发执行的逻辑执行流；文件把磁盘上物理的块集合虚拟化成命名的字节序列，用户无需关心数据在磁盘上的物理布局。

---

# Lecture 2—3: 线程、进程与调度（Threads, Processes, and Dispatching）

## 概述

本讲介绍操作系统最核心的两个执行抽象：**线程**（thread，最小的执行单元）与**进程**（process，一个或多个线程及其执行状态）。随后深入调度机制（dispatching）：OS 如何在有限的核（core）上运行远多于核数的线程，包括进程控制块（PCB）、线程状态机、上下文切换（context switch），以及陷阱（trap）与中断（interrupt）如何驱动调度器运行。

# 核心概念与系统机制图解

## 线程（Thread）

- **定义**：在一颗核上顺序执行的一段代码；它按顺序执行一串指令，因此"一次只发生一件事"，易于推理。
- **直观解释**：线程是"房子里的住户"。同一进程内的住户共享公共空间（全局变量、堆）和设施（打开的文件），但各自有自己的私人物品（寄存器、调用栈）。
- **为什么需要多线程**：利用多核并行；简化应用结构（如 Web 服务器为每个连接开一个线程）。
- **执行状态（Execution State）**：一切能影响或被线程影响的东西——代码、数据、寄存器、调用栈、打开的文件、网络连接、时间等。

## 进程（Process）

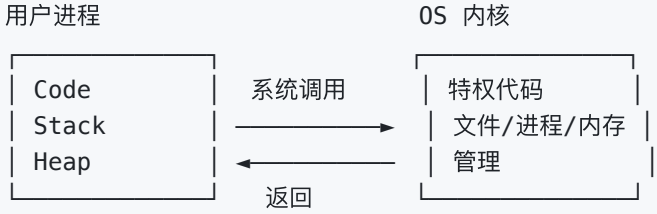
- **定义**：一个或多个线程 + 它们的执行状态；是操作系统资源分配（内存、文件句柄等）的基本单位。
- **直观解释**：进程是一座"房子"，拥有自己的地址空间（房间布局）、财产（数据）和住户（线程）。房子之间相互隔离——一个房子的问题通常不影响另一个房子。

**线程间共享 vs 私有状态**（Lecture 2 课堂问题）：

| 状态 | 进程内线程间共享? |
| --- | --- |
| 代码（Code） | ✅ 共享 |
| 变量（Variables，全局/堆） | ✅ 共享 |
| 寄存器（Registers） | ❌ 每个线程私有 |
| 调用栈（Call Stack） | ❌ 每个线程私有 |
| 打开的文件 / 网络连接 | ✅ 共享 |
| 时钟（Time of day） | ✅ 共享 |

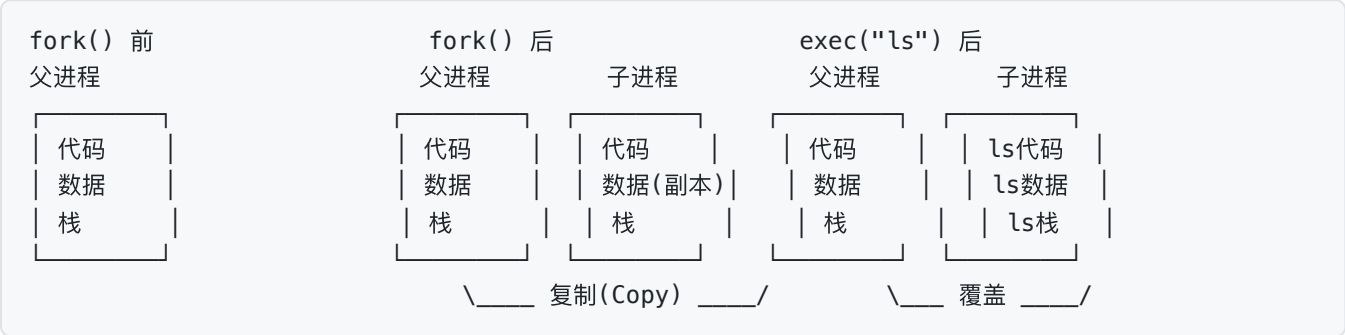
## 系统调用（System Call）

- **定义**：用户进程请求内核执行特权操作的机制；调用序列因 OS 而异，但本质都是"陷入内核"。
- **机制**：进程执行特殊指令（如 x86-64 的 `syscall`），CPU 切换到内核模式，跳到内核中预先注册的处理代码，执行完再返回用户模式。



## 进程创建：fork / exec / wait（Linux）

- `fork()`：复制当前进程，产生一个几乎完全相同的子进程（子进程从 fork 返回处继续执行，返回 0；父进程得到子进程 PID）。
- `execvp()`：用新程序的代码和数据**覆盖**当前进程——不创建新进程，只是换内容。
- `waitpid()`：等待子进程结束，回收其退出状态。
- **为什么"先复制再覆盖"**：优点——在 exec 之前可以修改进程状态（改环境变量、重定向文件描述符，例如 `ls > ls.out`）；缺点——大部分复制的工作被浪费。Windows 用 `CreateProcess`（10 个参数）一步到位，Linux 只需 fork（0 参数）+ exec（2 参数）。




## 线程创建

- 系统调用级：Linux `clone`、macOS `thread_create_running`、Windows `NtCreateThreadEx`；需要提供：起始程序计数器（要调用的函数）、线程栈区域与初始栈指针、参数传递方式。
- 语言库包装：C++ `std::thread t(func); t.join();`、C `pthread_create(...)`、Go `go func()`、Python `threading.Thread(...)`。

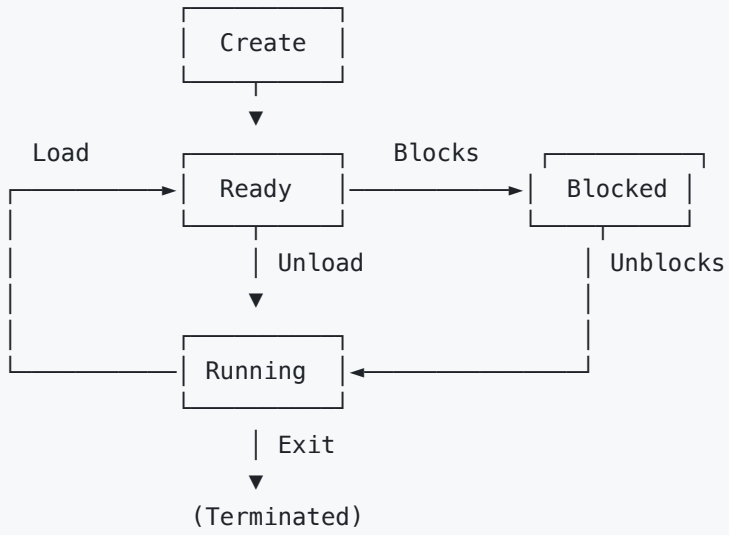
## 核（Core）与多核演化

- 早期一台机器一个 CPU（单核）；把多个处理器放进一颗芯片 → **多核（multicore）**；一颗核同时跑两个线程 → **同时多线程 SMT（Intel 超线程）**。
- 今天的核数：手机约 8 核，笔记本 12–16，台式机 16–24，服务器 128–256 核（256–512 硬件线程）。

## 进程控制块（PCB, Process Control Block）

- **定义**：内核为每个进程维护的数据结构，包含：
- 每个线程保存的执行状态（保存的寄存器等）；
- 调度信息；
- 进程所用内存的信息；
- 打开文件的信息；
- 记账及其他杂项信息。

## 线程状态机



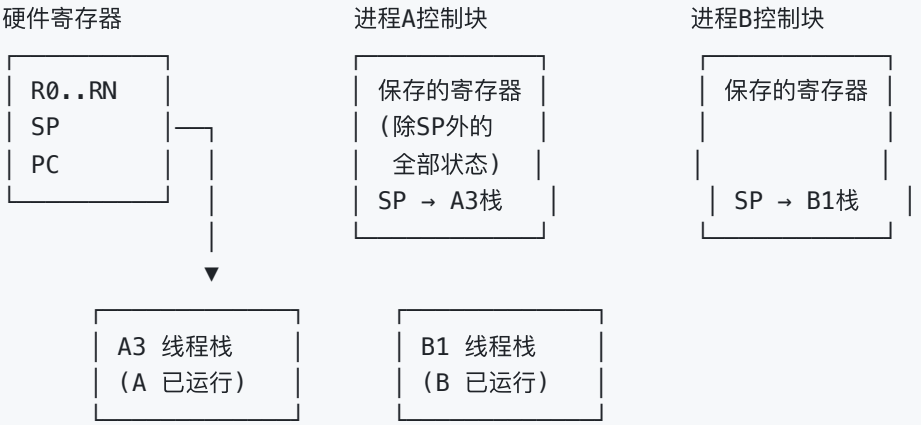

- **Ready**：就绪，等待被调度到核上；**Running**：正在核上执行；**Blocked**：阻塞（等待 I/O、锁、事件）。

## 调度器（Dispatcher）与上下文切换

- 单核调度器主循环：

```
while true:
    1. 让某个线程运行一段时间
    2. 保存它的执行状态
    3. 加载另一个线程的状态
end while        ← 第2、3步合称"上下文切换"
```

- **上下文切换机制图解**（以线程 A 切换到 B 为例）：




```
步骤: 1) 调用 context_switch
      2) 把寄存器保存进 A 的 PCB(线程状态)
      3) 把 SP 切换为 B 的栈指针, 从 B 的 PCB 恢复寄存器
      4) 从 B 上次被中断/让出的地址继续执行
```

- **是什么触发调度器运行?**

- 让进程自己回调 OS？→ **协作式多任务**：坏进程会搞死机器。✗
- **陷阱（trap）**：系统调用、非法指令、段错误、页错误等，自动跳入 OS；
- **中断（interrupt）**：键盘按键、磁盘传输完成等外部事件；
- **定时器（timer）**：如果进程没有主动让出 CPU，定时器周期性中断强制切换（抢占式调度的基础）。
- **线程是怎么"出生"的**：创建线程时，内核把 PCB 线程状态和栈初始化为"看起来刚从第一行指令上下文切换而来"的样子。

## 代码示例与系统调用解说

### 示例 A：fork + exec + wait 创建并运行新程序

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();                 // 1. 创建子进程(复制当前进程)

    if (pid < 0) {                      // fork 失败
        perror("fork failed");
        return 1;
    }

    if (pid == 0) {
        // 子进程代码: 用 ls 程序覆盖自己
        char* argv[] = {"ls", "-l", nullptr};
        execvp("ls", argv);             // 2. 覆盖当前进程的代码和数据
        perror("exec failed");          // 只有 exec 失败才会执行到这里
        return 1;
    } else {
        // 父进程代码: 等待子进程结束
        int status;
        waitpid(pid, &status, 0);       // 3. 阻塞直到子进程退出
        printf("Child (pid %d) finished\n", pid);
    }
    return 0;
}
```

**【代码做了什么？】** 1. `fork()` 调用一次、返回两次：父进程得到子进程 PID，子进程得到 0。 2. 子进程调用 `execvp("ls", ...)` 把自己替换成 `ls` 程序；exec 成功则不返回。 3. 父进程 `waitpid` 阻塞等待，子进程结束后回收其退出状态。

**【系统机制透视】** – `fork()` 在内核中做了什么：分配新的 PCB 和地址空间结构，把父进程的页表/内存复制（现代实现是**写时复制 COW**：父子共享物理页并标记只读，任一方向写入时内核才复制该页）给子进程，把子进程加入就绪队列。 – `exec` 做了什么：释放/替换进程的代码段、数据段，载入新可执行文件，重置栈

与堆，**保留**文件描述符与环境变量（这正是"先 fork 再 exec"的意义——可以在 exec 前重定向 I/O）。 – 隔离性：父进程和子进程的地址空间彼此独立，任何一方的修改不会影响另一方。

### 示例 B：std::thread 创建线程

```
#include <iostream>
#include <thread>

void worker(int id) {
    // 这段代码将与主线程并发执行
    std::cout << "Thread " << id << " running\n";
}

int main() {
    std::thread t(worker, 42);   // 创建一个新线程，从 worker(42) 开始
    std::cout << "Main thread continues\n";
    t.join();                    // 等待线程 t 结束(否则进程退出时会崩溃/未定义行为)
    return 0;
}
```

**【代码做了什么？】** `std::thread` 构造器在库内部调用线程创建系统调用（Linux 上是 `clone`），为新线程分配独立栈并从 `worker` 函数开始执行；`join()` 阻塞主线程直到工作线程完成。

**【系统机制透视】** – **与 fork 的本质区别**：`std::thread` 不复制地址空间——新线程与已有线程**共享**进程的全部代码、全局变量、堆和打开文件，只有寄存器和栈是私有的。这就是"轻量级"的含义。 – 新线程的"出生"：内核创建一个线程控制块（TCB），把它的寄存器现场初始化成"刚从函数入口被调度进来"的样子（返回地址指向线程结束后的清理代码，栈指针指向新分配的栈顶）。 – 代价：由于共享地址空间，线程间必须用同步机制（后续讲座的锁/条件变量）避免数据竞争。

## 关键要点

1. 线程是最小的执行单元；进程 = 线程 + 执行状态，是资源分配与隔离的基本单位。
2. 上下文切换 = 保存当前线程状态 + 恢复目标线程状态；由陷阱（trap）或中断（interrupt）触发，定时器是实现抢占式调度的关键硬件。
3. `fork` / `exec` / `wait` 是 Unix 进程管理三件套；`fork` 复制进程、`exec` 换程序、`wait` 回收。
4. 线程切换比进程切换便宜：同一进程内线程共享页表，切换时无需刷新 TLB；进程切换必须换页表。
5. 调度器（dispatcher）是**机制**（怎么切），调度策略（scheduler）是**政策**（切给谁）——Lecture 8 详述。

## 常见陷阱与注意事项

- **僵尸进程（Zombie）**：子进程退出后其 PCB 仍存在，直到父进程 `wait()` 回收；父进程不 wait 会造成内核资源泄漏。
- **孤儿进程（Orphan）**：父进程先退出，子进程被 init（PID 1）收养并负责回收。
- **忘写 `t.join()` / `t.detach()`**：`std::thread` 析构时线程仍可 join 会调用 `std::terminate`。
- **线程安全**：多线程共享全局变量而无同步 → 数据竞争，结果不确定。
- **exec 失败检查**：`exec` 成功后不返回，必须检查失败情况（否则子进程会"悄悄"继续执行父进程代码）。

## 思考题

1. **问题**：为什么线程上下文切换通常比进程上下文切换开销小?
2. **答案**：同一进程内的线程共享地址空间和页表，切换时不需要更换页表、不必刷新 TLB（缓存失效是进程切换的主要开销之一）；进程切换必须载入新页表并使 TLB 全部失效，还要承受缓存冷启动。
3. **问题**：`fork()` 之后父进程修改一个全局变量，子进程能看到吗?
4. **答案**：看不到。`fork()` 创建的是独立副本（写时复制技术下，写入才复制物理页），父进程的修改只影响自己的副本，子进程的页不受影响。
5. **问题**：为什么现代 OS 用定时器中断而不是"请求进程主动让出 CPU"来驱动调度器?
6. **答案**：协作式（cooperative）方案下，一个拒绝让出 CPU 的（恶意或出错的）进程会独占机器、使系统无法响应；定时器中断提供了**抢占（preemption）**能力，保证每个线程最终都能获得 CPU（公平性与可用性）。

---

# Lecture 4: 并发（Concurrency）

## 概述

当多个线程共享状态时，执行顺序会影响结果——这就是并发编程困难的根源。本讲从"独立线程 vs 协作线程"出发，引入**竞态条件（race condition）**、**原子操作（atomic operation）**、**临界区（critical section）**与**互斥（mutual exclusion）**等核心概念，并用经典的"买牛奶问题（Too Much Milk）"演示：仅靠原子读/写无法解决同步问题，从而引出下一讲的锁与条件变量。

## 核心概念与系统机制图解

### 独立线程（Independent Threads）vs 协作线程（Cooperating Threads）

- **独立线程**：不共享状态，彼此既不能影响也不能被影响。优点：**确定性**（结果只取决于输入）、**可复现**、**调度顺序无关**。例如同时运行两个互不相干的程序。
- **协作线程**：共享状态。缺点：**非确定性**、不可复现、结果依赖执行顺序、调度顺序至关重要。例如多线程银行账户、Web 服务器。

### 为什么要允许线程协作?

1. **资源共享**：一块磁盘上很多文件；一个银行账户文件被很多 ATM 访问；
2. **性能**：多核并行执行；I/O 与计算重叠。
3. 结论：并发是**必要的**，不能回避。

### 竞态条件（Race Condition）

- **定义**：多个线程访问共享数据，且至少一个线程在写，而访问顺序影响结果的情形。

```
线程 #1:            线程 #2:
A = 1;               B = 1;          → 顺序无关，无竞态
A = B + 1;           B = 2 * B;      → 顺序相关，有竞态
```

### 原子操作（Atomic Operation）

- **定义**：对其它线程而言"看起来瞬间完成"的操作——不可分割（indivisible），无法被观察到执行到一半。
- **通常是原子的**：单字读取、单字写入（`A = B;` 要么读到旧值要么新值，绝不会是半新半旧的位）。
- **通常不是原子的**：`x++`（读–改–写三步）、结构体拷贝、多步操作。
- **关键结论**：如果没有硬件提供的原子原语，你**无法用软件凭空造出**一个来——必须由硬件提供原子读改写指令（Lecture 6 详述）。

### 临界区与互斥

- **同步（Synchronization）**：用原子操作保证协作线程的正确性。
- **临界区（Critical Section）**：一次只允许一个线程执行的代码段。
- **互斥（Mutual Exclusion）**：强制实现临界区的机制，通常用某种锁实现。

### 买牛奶问题（Too Much Milk）——软件尝试的演化

**问题设定**：两个室友分别检查冰箱，若没牛奶就去买，然后置 `milk = 1`。目标是**安全性**（绝不买太多牛奶）与**活性/可用性**（需要时一定有人买）。

```
尝试1(朴素):
if (milk == 0) {
    buy_milk();
    milk = 1;
}
   两个线程同时检查milk==0
   → 都去买 → 牛奶过多
```

```
尝试2(加便条note, 先查milk后写note):
 if (milk == 0) {
     if (note == 0) {
         note = 1;
       buy_milk();
note = 0;
   }
     }
     两个线程可能同时通过 if(note==0) → 仍过多
```

```
尝试3(轮流):
A: if (note == 0) { ... }
B: if (note == 1) { ... }
   → 安全, 但A永远等B或B永远等A
   (饥饿, 活性失败)
```

```
尝试4(各自便条+忙等):
 A: noteA = 1;
    if (noteB == 0) { if (milk==0) buy; }
 noteA = 0;
 B: noteB = 1;
   while (noteA == 1) {}   // 忙等
   if (milk == 0) buy_milk();
   noteB = 0;
   → 正确! 但不对称 + 忙等浪费CPU
```

- 对称且正确的软件解法是 **Peterson 算法**，但本课的重点是：**这些尝试都太复杂、太微妙**——我们需要更好的原语：**锁** 和 **条件变量**（下一讲）。

## 代码示例与系统调用解说

### 示例：无同步的计数器（演示竞态）

```
#include <iostream>
#include <thread>
#include <vector>

int counter = 0;   // 全局共享变量

void increment(int n) {
    for (int i = 0; i < n; i++) {
        counter++;   // 读-改-写, 非原子!
    }
}

int main() {
    std::vector<std::thread> threads;
    for (int t = 0; t < 4; t++) {
        threads.emplace_back(increment, 100000);
    }
    for (auto& t : threads) t.join();
    std::cout << "counter = " << counter << " (期望 400000)\n";
    return 0;
}
```

**【代码做了什么？】** 4 个线程各自对共享全局变量 `counter` 执行 10 万次 `counter++`，最后打印结果。

**【系统机制透视】** – `counter++` 在机器层面是三条指令：`load counter; add 1; store counter`。两个线程可能交错执行（如 T1 load 后、store 前，T2 也 load 到同一旧值），导致两次增量只生效一次。 – 运行结果通常小于 400000 且每次不同——这正是"协作线程非确定性"的直接体现。 – 修复方向（预告）：用 `std::mutex` 保护临界区，或用 `std::atomic<int>`（硬件原子指令）。

## 关键要点

1. 协作线程共享状态 → 执行顺序影响结果 → 竞态。
2. 原子操作 = 对其它线程"不可分割"的操作；单字读写是原子的，多步操作不是。
3. 硬件必须提供原子原语；软件只能在其之上构建更高层的同步构造。
4. 正确同步需同时满足**安全性**（坏事绝不发生）与**活性**（好事最终发生）。
5. 买牛奶问题说明：手工软件同步极易出错——这是引入锁与条件变量的动机。

## 常见陷阱与注意事项

- **假设 `x++` 是原子的**——它不是。
- **只检查一个条件就进入临界区**：检查与修改之间必须是一个原子步骤，否则两个线程都能通过检查。
- **忙等待（busy waiting）**：空转循环浪费 CPU；应阻塞（后续用条件变量解决）。
- **只保证安全性而忽略活性**（如轮流方案导致的饥饿）。

## 思考题

1. **问题**：`A = B;`（单字赋值）是原子的吗？`A = B + 1;` 呢？
2. **答案**：`A = B;` 是单字读+单字写，通常原子；`A = B + 1;` 至少是"读 B → 计算 → 写 A"，中间步骤可被其它线程交错，不是原子的。
3. **问题**：为什么说"如果硬件没有原子操作，软件无法创建一个"？
4. **答案**：任何软件"原子化"方案本质上都要靠"读–判断–写"序列，而这些步骤本身需要原子性来防止交错；因此原子性必须由硬件指令（如 x86 的 `xchg`、`lock` 前缀）在最底层提供。

---

# Lecture 5: 锁与条件变量（Locks and Condition Variables）

## 概述

买牛奶问题证明：我们需要高层同步原语。本讲引入**互斥锁（mutex/lock）**提供互斥、**条件变量（condition variable）**提供阻塞等待，并通过"生产者–消费者管道"这一经典问题，逐步演示从错误版本到正确版本（monitor 模式）的完整推导过程。

## 核心概念与系统机制图解

### 锁（Lock / std::mutex）

- **定义**：提供互斥的同步对象。`lock()`：阻塞直到锁空闲，然后标记为当前线程持有；`unlock()`：把锁标记为空闲并唤醒一个等待线程；`try_lock()`：不阻塞，拿不到就返回错误。
- **直观解释**：锁是"厕所门闩"——一个人进去后把门闩上，其他人只能在门外等待；出来时开门让下一个进。
- **买牛奶问题的锁解法**：

```
std::mutex mutex;
mutex.lock();
if (milk == 0) { buy_milk(); milk = 1; }
mutex.unlock();
```

### 条件变量（Condition Variable, std::condition_variable）

- **定义**：一种让线程在**持锁**状态下原子地"释放锁 + 睡眠等待某个条件"的原语。
- **直观解释**：条件变量是"候诊室的叫号"——患者（等待线程）在候诊室睡觉，护士（通知线程）叫到号时唤醒他；被唤醒后要重新排队（重新拿锁）才能继续。
- **API**：
- `wait(lock)`：原子地释放锁并把调用线程置为睡眠；被唤醒后重新获取锁再返回。
- `notify_one()`：唤醒一个正在睡眠的线程（如果有）。
- `notify_all()`：唤醒所有正在睡眠的线程。
- **重要警示**：被通知的线程**不一定立刻拿到锁**，唤醒时条件可能已不成立 → **必须用 `while` 重新检查条件，而不是 `if`**。

### 生产者–消费者问题（Producer/Consumer，一个有界缓冲的"管道"）

需求：生产者放字符进缓冲区，消费者取字符；缓冲空时消费者等待，缓冲满时生产者等待。

**错误版本 v1**：只用锁，`get()` 在空缓冲时 `count--` 变成 –1、读到未定义字符。 **错误版本 v2/v2.5**：空转等待（`while (count == 0) {}`）——忙等浪费 CPU，或在锁外检查条件导致竞态。 **错误版本 v2.9**：

`while (count == 0) { mutex.unlock(); mutex.lock(); }`——正确但**过度忙等**（惊群式空转）。 **正确版本 v3**：条件变量 + `while` 循环（见代码示例）。

### Monitor 模式（Assign2 的核心）

- **定义**：一种组织共享数据与同步的编程模式，包含：
- 一个共享数据结构；
- 一组操作该数据的方法；
- 一把锁（每个方法进入时加锁、返回前解锁）；
- 一个或多个条件变量用于等待。
- **直观解释**：把"资源 + 门卫 + 排队区"打包成一个对象；外界只能通过带锁的方法访问资源，从结构上杜绝"忘了加锁"。

### 锁的粒度（Lock Granularity）

- **粗粒度（coarse-grained）**：一把大锁保护所有数据——简单、开销低，但并发度低（锁竞争激烈）。
- **细粒度（fine-grained）**：多把小锁保护不同数据——并发度高，但复杂、易出竞态与死锁（下下讲）。
- **最佳实践**：在可接受的竞争水平下**尽量少用锁**；把一把锁与"一组相关的变量"绑定（即 monitor 风格）。例：Linux 内核、Python 全局解释器锁（GIL）。

## 代码示例与系统调用解说

### 示例：生产者/消费者 Pipe（monitor 模式，正确版 v4）

```
#include <condition_variable>
#include <mutex>

const int SIZE = 8;

class Pipe {
public:
    Pipe() : count(0), nextPut(0), nextGet(0) {}

    void put(char c) {
        std::unique_lock<std::mutex> lock(mutex);   // 进入临界区(RAII 加锁)
        while (count == SIZE) {                     // 缓冲满: 等待, 必须用 while!
            charRemoved.wait(lock);                 // 原子地释放锁并睡眠
        }
        count++;
        buffer[nextPut] = c;
        nextPut = (nextPut + 1) % SIZE;
        charAdded.notify_one();                     // 唤醒一个等待的消费者
    }                                               // 出作用域自动解锁

    char get() {
        std::unique_lock<std::mutex> lock(mutex);
        while (count == 0) {                        // 缓冲空: 等待
            charAdded.wait(lock);
        }
        count--;
        char c = buffer[nextGet];
        nextGet = (nextGet + 1) % SIZE;
        charRemoved.notify_one();                   // 唤醒一个等待的生产者
        return c;
    }

private:
    std::mutex mutex;
    std::condition_variable charAdded, charRemoved;
    char buffer[SIZE];
    int count;
    int nextPut;
    int nextGet;
};
```

**【代码做了什么？】** – `put`：加锁 → 若缓冲满则 `charRemoved.wait(lock)` 睡眠 → 放入字符 → `charAdded.notify_one()` → 自动解锁。 – `get`：加锁 → 若缓冲空则 `charAdded.wait(lock)` 睡眠 → 取字符 → `charRemoved.notify_one()` → 返回。 – `std::unique_lock` 是 RAII 锁包装：构造时加锁、析构时解锁，且支持传给 `wait()`（`wait` 需要能原子释放/重获锁）。

**【系统机制透视】** – **`wait()` 的原子性为什么关键**：如果"释放锁"和"睡眠"不是原子的，那么消费者释放锁后、入睡前，生产者可能执行完 `notify_one()`——通知落在消费者入睡之前，消费者将永远错过唤醒（丢失唤醒）。`wait(lock)` 把"释放锁+睡眠"合并为一步，杜绝该窗口。 – **为什么必须 `while` 而不是 `if`**：被唤醒的线程要重新竞争锁；在它拿到锁之前，其它线程可能又消费/生产，使条件再次不成立（如 T1、T2 两

个消费者同时被唤醒，只有一个字符)。`while` 循环保证唤醒后**重新检查条件**。 – **`notify_one` vs `notify_all`**：本问题中任一时刻只需一个消费者/生产者被唤醒，`notify_one` 足够且开销小；但若被唤醒的线程发现条件仍不成立（例如用 `notify_all` 唤醒多个等待者），`while` 会让多余者继续睡眠——正确性由 `while` 兜底。

## 关键要点

1. 锁解决**互斥**（一次一个线程进临界区）；条件变量解决**等待**（在持锁时阻塞直到某事件发生）。
2. 条件变量 `wait` 必须与锁配合，且**必须用 `while` 重新检查条件**（防丢失唤醒与假唤醒）。
3. Monitor 模式 = 共享数据 + 方法 + 锁 + 条件变量，是编写线程安全类的推荐结构。
4. 锁的粒度权衡：并发度 vs 复杂度/开销——尽量少用锁、按"相关变量组"绑定。
5. 被 `notify` 唤醒 ≠ 条件成立：唤醒只是"可能有机会了"。

## 常见陷阱与注意事项

- **用 `if` 代替 `while` 检查条件** → 竞态（消费未定义字符、覆盖未读字符）。
- **忘记在等待前加锁，或在等待时持锁不放** → 死锁或未定义行为。
- **`notify` 时机错误**：必须在修改共享状态**之后**、释放锁**之前（或之后）**通知；通知太早会丢失唤醒。
- **锁粒度不当**：全局一把大锁（性能差）vs 每变量一把小锁（易死锁）。
- **忘记解锁**：用 RAII（`std::lock_guard` / `std::unique_lock`）避免。

## 思考题

1. **问题**：`pthread_mutex_lock()` / `std::mutex::lock()` 在锁被占用时，线程进入什么状态?
2. **答案**：线程被放入该锁的等待队列，状态变为 **BLOCKED（阻塞）**，不再占用 CPU；当持锁线程 `unlock()` 时，内核/运行库把一个等待线程移回就绪队列（READY），它重新竞争锁。
3. **问题**：为什么条件变量的 `wait` 要"原子地释放锁"？如果先释放锁、再睡眠会怎样?
4. **答案**：先释放锁会留下一个窗口：在释放锁与睡眠之间，另一个线程可能已经修改条件并发出了 notify，而等待者还没入睡，于是永远错过唤醒（丢失唤醒）。`wait` 原子完成"释放+睡眠"以消除该窗口。
5. **问题**：monitor 模式为什么能减少同步错误?
6. **答案**：它把共享数据封装在对象内，所有访问都必须经由"先加锁的方法"，从结构上杜绝了"忘记加锁/解锁"；条件变量与共享状态放在一起，使等待条件与状态修改的对应关系一目了然。

---

# Lecture 6: 锁的实现（Implementing Locks）

## 概述

前几讲使用锁，本讲**在操作系统内部实现锁**：单核上通过**关中断（disable interrupts）**获得原子性；多核上必须借助硬件的**原子读改写指令（如 xchg）**。我们从朴素实现出发，逐版本发现竞态窗口，最终得到 Linux 风格的正确实现，并理解"为什么多核上一定程度的忙等不可避免"。

## 核心概念与系统机制图解

### 单核（Uniprocessor）锁：关中断

- **关键事实**：单核上，只有陷阱（trap）或中断（interrupt）能把当前线程切走。因此**关中断 = 获得临界区**。
- **实现**：

```
class Lock {
    bool locked = false;
    ThreadQueue q;          // 等待该锁的线程队列
};

void Lock::lock() {
    intrDisable();                          // 原子地: 关中断
    if (!locked) {
        locked = true;
    } else {
        q.add(currentThread);               // 加入等待队列
        blockThread();                      // 阻塞自己(必须保持中断关闭!)
    }
    intrEnable();                           // 重新开中断
}

void Lock::unlock() {
    intrDisable();
    if (q.empty()) {
        locked = false;                     // 无人等待, 直接释放
    } else {
        unblockThread(q.remove());          // 唤醒队首线程(把锁"移交"给它)
    }
    intrEnable();
}
```

- **为什么 `blockThread()` 必须在关中断状态下调用**：若先开中断再阻塞，开中断与阻塞之间可能来一个中断把另一线程调度进来，而该线程可能调用 `unlock` / `lock`，破坏队列一致性——"加进队列"与"变成阻塞"必须原子。

### 多核（Multiprocessor）锁：原子读改写

- **问题**：关中断只对当前核有效，其它核上的线程照常访问共享锁变量 → 需要**硬件原子指令**。
- **原子交换 `xchg`**：把变量的旧值取回并同时写入新值，一步完成（Intel x86 指令）。

### 多核锁实现演化（Lecture 6 逐版本）

```
v1: 自旋锁(spinlock)——忙等:
    void lock()  { while (locked.exchange(true)) { /* 空转 */ } }
    void unlock(){ locked = false; }
    → 简单但忙等; 而且锁竞争激烈时大量缓存一致性流量

v2: 尝试加原子交换+阻塞:
    if (locked.exchange(true)) { q.add(currentThread); blockThread(); }
    → 竞态: 两个核可能同时操作线程队列 q!

v3: 用自旋锁保护队列(二阶段锁):
    自旋锁保护 locked 标志与线程队列; 队列操作在自旋锁内完成
    → 仍有竞态: 持锁者正在运行, 等待者被阻塞后, 谁来唤醒它?
      (unlock 由另一个核执行时与 blockThread 交错的问题)

v4: 显式设置 BLOCKED 状态 + redispatch:
    q.add(currentThread);
    currentThread->state = BLOCKED;   // 先标记阻塞
    spinlock = false;                 // 释放自旋锁
    redispatch();                     // 再调度其它线程
    → Linux 的做法: 让调度器看到"我已阻塞", 避免调度器把我又调度回来

v5: 加上关中断, 防止本核中断打断上述序列
    → 最终版(见代码示例)
```

- **为什么多核上忙等不可避免**：等待一个由**其它核**持有的锁时，你无法通过"关本核中断"获得原子性，只能自旋；目标是**尽量缩短忙等时间**（只保护极短的临界区，如队列操作），而不是完全消除。

## 代码示例与系统调用解说

### 示例：多核锁的最终实现（v5，Linux 风格）

```
#include <atomic>

class Lock {
public:
    void lock() {
        intrDisable();                          // 1. 关本核中断
        while (spinlock.exchange(true)) { }     // 2. 自旋获取自旋锁(保护内部状态)
        if (!locked) {
            locked = true;
            spinlock = false;                   // 3a. 拿到锁, 释放自旋锁
        } else {
            q.add(currentThread);               // 3b. 加入等待队列
            currentThread->state = BLOCKED;     // 4. 显式标记阻塞
            spinlock = false;                   // 5. 释放自旋锁
            redispatch();                       // 6. 让出 CPU, 调度器运行其它线程
        }
        intrEnable();
    }

    void unlock() {
        intrDisable();
        while (spinlock.exchange(true)) { }
        if (q.empty()) {
            locked = false;                     // 无人等待 → 释放锁
        } else {
            unblockThread(q.remove());          // 有人等待 → 把锁移交给队首线程
        }
        spinlock = false;
        intrEnable();
    }

private:
    bool locked = false;
    ThreadQueue q;                              // 阻塞在该锁上的线程队列
    std::atomic<bool> spinlock;                 // 保护 locked 与 q 的自旋锁
};
```

**【代码做了什么？】** `lock()` 先关中断（防本核中断），再用自旋锁保护内部状态：锁空闲则直接占有；否则入队、标记阻塞、释放自旋锁并让出 CPU。`unlock()` 对称地移交锁或释放锁。

**【系统机制透视】** – **为什么需要两级同步**：`locked` 标志与线程队列 `q` 是共享状态，任何核上的线程都能操作它们，必须用硬件原子指令（`exchange`）保护；而"入队→标记 BLOCKED→redispatch"序列必须对调度器原子可见，否则调度器可能把一个已阻塞的线程又调度到核上运行（它却在等待锁）——这就是 v4 中"先设 BLOCKED 再 redispatch"的意义。 – **`xchg` 的硬件行为**：一条指令同时"读取旧值并写入新值"，总线/缓存一致性协议保证所有核看到一致的顺序——这是软件无法模拟的原子性来源。 – **用户态对应物**：Assign4 在用户态实现单核锁/条件变量时，由于没有中断开关，用"关闭抢占/单线程调度器配合"的技巧；而 C++ 的 `std::mutex` 在用户态通常用 futex（fast user-space mutex）：先试自旋，失败则通过系统调用睡眠，把"忙等+阻塞"结合。

## 关键要点

1. 单核上"关中断"即可构造临界区；多核上必须依赖硬件原子读改写指令。
2. 阻塞线程必须在关中断/持自旋锁状态下完成"入队 + 标记阻塞 + 让出"，避免调度器竞态。
3. 多核系统中忙等不可避免，但应把忙等限制在极短临界区内。
4. 硬件原子原语（如 `xchg`）是构建一切高层同步（锁、信号量、futex）的基石。
5. 一个"看起来正确"的锁实现往往藏有竞态窗口——需要逐版本推敲中断与调度器的交错。

## 常见陷阱与注意事项

- **在关中断时调用可能阻塞的函数** → 中断长时间关闭，系统失去响应（时钟中断进不来）。
- **先释放自旋锁再 redispatch**：顺序颠倒会导致另一核的线程在队列操作进行中修改队列。
- **忘记处理"锁被移交"的情形**：`unlock` 若直接 `locked=false` 而不唤醒，可能出现"锁空闲但无人知道"。
- **自旋锁持锁时间过长**：其它核自旋浪费 CPU，缓存行 bouncing 严重。

## 思考题

1. **问题**：为什么单核锁可以只用关中断实现，而多核不行?
2. **答案**：单核上只有中断/陷阱能切换线程，关中断即独占 CPU；多核上其它核的线程可并发访问共享锁变量，关本核中断管不住别的核，必须用跨核一致的原子指令。
3. **问题**：v4 中 `currentThread->state = BLOCKED;` 为什么必须发生在 `redispatch()` 之前?
4. **答案**：若先 redispatch 再设状态，调度器可能在状态仍为 RUNNING/READY 时把该线程重新调度到核上，而它正等着锁——出现"阻塞线程还在跑"的不一致；先标记 BLOCKED 保证调度器不会再选它。

---

# Lecture 7: 死锁（Deadlock）

## 概述

当系统使用多把锁（细粒度锁、模块化设计）时，线程经常需要同时持有多把锁——这引入了**死锁**风险。本讲定义死锁、给出其四个必要条件，并用"资源–线程图"的环（circularity）直观解释，最后讨论**检测**（数据库常用）与**预防**（消除任一必要条件，实践中常用全局锁序）两种对策。

# 核心概念与系统机制图解

## 死锁（Deadlock）

- **定义**：一组线程全部阻塞，每个线程都在等待另一个线程持有的资源；由于所有线程都阻塞，谁也无法释放资源 → "相互阻塞导致无法进展"。
- **直观解释**：两个人都拿着对方要用的钥匙，谁也不肯先放手 → 双双卡死。

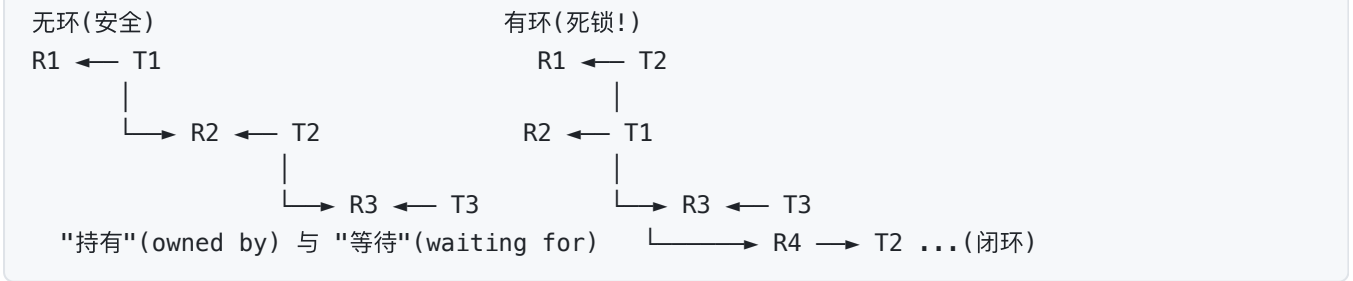
### 经典死锁示例

```
线程 A:                          线程 B:
m1.lock();                       m2.lock();
m2.lock();   ← 等待B释放m2       m1.lock();   ← 等待A释放m1
...                              ...
m2.unlock();                     m1.unlock();
m1.unlock();                     m2.unlock();
→ A 持有 m1 等 m2, B 持有 m2 等 m1 → 环 → 死锁
```

### 死锁的四个必要条件（全部满足才可能死锁）

| # | 条件 | 含义 | 消除途径 |
|---|---|---|---|
| 1 | **有限访问（互斥）** Mutual Exclusion | 资源不能被共享 | 让线程永不需要等待（资源足够多）——对锁不现实 |
| 2 | **不可抢占** No Preemption | 资源一旦给出不能被夺走 | 把资源抢走——对 CPU 可行，对锁不可行 |
| 3 | **多重独立请求（持有并等待）** Hold & Wait | 线程持有一个资源时再请求另一个 | 一次请求全部资源——难以实现且易过度分配 |
| 4 | **循环等待** Circular Wait | 请求–持有图中有环 | **全局锁序：所有线程按同一顺序加锁** ← 最常用 |

### 资源–线程图（可视化环）




### 死锁检测（Detection）

- 确定系统是否死锁，然后**终止其中一个线程**打破死锁。

- 对操作系统通常不实用，但**数据库系统**常用：事务可被中止并重试。

### 死锁预防（Prevention）——重点

- **消除条件 4（循环等待）是最实用的途径**：给每把锁分配一个编号（锁的"等级/rank"），**所有线程都按递增（或递减）顺序加锁**。运行时可校验加锁顺序是否违反。
- **案例：两个进程执行 mv 命令**

```
进程1: mv a/x  b/y    需要锁住目录 a 和 b
进程2: mv b/z  a/q    需要锁住目录 b 和 a
按"参数顺序加锁" → 进程1锁a等b, 进程2锁b等a → 死锁
按"全局固定顺序"(先锁 a 再锁 b) → 进程2也先锁a, 不会与进程1交叉 → 无死锁
```

### 设计洞见

- **死锁是全局设计问题**：它打破模块化——需要所有模块就加锁顺序达成一致；修改系统时很容易重新引入死锁。

## 代码示例与系统调用解说

### 示例：用"全局锁序"避免死锁（锁等级）

```
#include <mutex>
#include <stdexcept>

std::mutex m1, m2;          // 规定: 永远先锁 m1, 再锁 m2 (lock rank 1 < 2)

void transfer_ab(std::mutex& first, std::mutex& second) {
    // 防止调用方以任意顺序传参: 按地址排序, 保证全局一致的加锁顺序
    if (&first < &second) {         // 先锁地址较小的锁
        first.lock();
        second.lock();
    } else {
        second.lock();
        first.lock();
    }
    // ... 转账等临界区操作 ...
    second.unlock();
    first.unlock();
}
```

**【代码做了什么？】** 无论调用者以何种顺序传入两把锁，函数内部都按"地址排序"固定加锁顺序，从结构上消除循环等待。

**【系统机制透视】** – 预防死锁的关键不是"不要同时持有多把锁"，而是"所有线程以**相同全局顺序**获取它们"——这样请求–持有图中不可能出现环。 – 如果两个线程以相反顺序加锁，即使只差一把锁，也可能形成

等待环；锁序规则把"环"这种全局属性变成"局部可检查"的属性（每个加锁点只需检查等级单调递增）。 – 现实中（如 Linux 内核）用锁等级 + 静态/动态检查（lockdep）来发现违反锁序的代码路径。

## 关键要点

1. 死锁四条件：互斥、不可抢占、持有并等待、循环等待；**全部满足**才死锁。
2. 最实用的预防手段：**全局锁序**（所有线程按同一顺序获取锁）。
3. 死锁检测（终止线程）在数据库事务中常用，在操作系统中不常用。
4. 死锁是全局设计问题，破坏模块化；引入新锁时必须与既有锁序保持一致。
5. 死锁不仅限于锁：任何"等待"（内存耗尽、网络消息、分布式系统）都可能形成死锁。

## 常见陷阱与注意事项

- **加锁顺序不一致**：两个线程用不同顺序获取同一组锁 → 最经典的死锁来源。
- **回调/嵌套调用中加锁**：A 持锁调用 B，B 又尝试锁 A 持有的锁。
- **多个互斥锁未分层**：新代码随意加锁而不检查全局顺序。
- **忙等也能死锁**：两个线程互相自旋等待对方释放自旋锁（饥饿与死锁的边界）。

## 思考题

1. **问题**：只有三个必要条件而没有循环等待，会死锁吗?
2. **答案**：不会。循环等待是形成"全体阻塞"的最后一环；没有环，等待图有向无环，总有一个线程能获得所需资源并最终释放。
3. **问题**：为什么数据库系统偏好"死锁检测+回滚"而非预防?
4. **答案**：数据库事务的访问模式动态多变，难以预先规定全局锁序；而事务具有可回滚性（undo log），检测到死锁后中止并重试一个事务代价可控，比限制所有事务的加锁模式更灵活。
5. **问题**：`mv a/x b/y` 与 `mv b/z a/q` 若都按"参数顺序"锁目录会死锁，如何修?
6. **答案**：改为全局固定的目录锁序（例如按目录 i–number 或路径字典序加锁），两个进程都以相同顺序获取目录锁，消除环。

---

# Lecture 8: 调度（Scheduling）

# 概述

调度器（dispatcher）是"怎么切换"的机制，调度策略（scheduler）是"切换给谁"的政策。本讲从最简单的 FIFO 出发，引入时间片与抢占（round robin），讨论响应时间最优的 SRPT 及其"用过去预测未来"的近似实现（优先级队列、4.4BSD 调度器），最后讨论多核调度与调度在现代系统中的角色。

# 核心概念与系统机制图解

## 基本术语与目标

- **调度问题**：给定一个可切换线程的调度器、一组就绪线程和若干核，**哪个线程在哪个核上跑多久**？
- **调度指标（Goals）**：
- 最小化**响应时间**（让用户保持忙碌）；
- 高效利用资源（核与磁盘都忙；上下文切换开销低）；
- 公平性（均衡分配 CPU，避免饥饿）。

## FIFO（先来先服务，非抢占）

- 就绪队列尾部入队，队首线程跑到退出或阻塞为止。
- 问题：**饥饿**（长任务堵死短任务）、**响应时间长**。

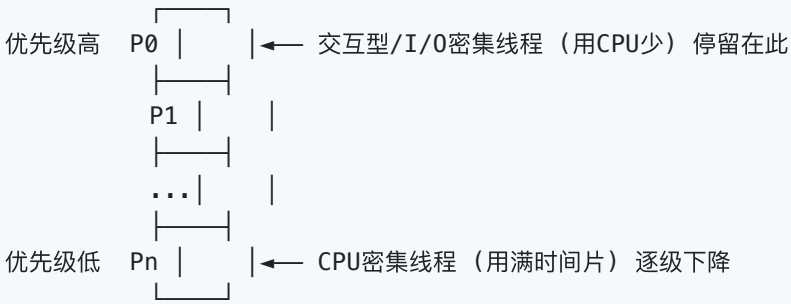
## Round Robin（时间片轮转，抢占）

- 每线程运行一个**时间片（time slice）**后被定时器中断强制切换，回到队尾。
- 实现：定时器硬件周期性中断。
- **时间片长短权衡**：太长→退化成 FIFO；太短→切换开销过大。
- 例子（Lecture 8 场景）：A 100ms、B 1ms、C 2ms 三个作业：
- FIFO 平均完成时间 ≈ 101.3ms；RR（1ms 时间片）≈ 36.7ms → RR 对短作业友好得多；
- 三个 10ms 作业：FIFO 平均 20ms；RR ≈ 29ms → 公平但不总是最优。

## SRPT（最短剩余处理时间优先）

- **定义**：总是运行"剩余时间最少"的线程，运行到完成（不中断）。
- **优点**：**理论上响应时间最优**（可证明）；偏好 I/O 密集线程，保持磁盘/网络忙碌。
- **缺点**：需要知道未来（**无法实现**）；可能饥饿。
- **近似**：用过去预测未来——"一个线程若已长时间不阻塞，很可能继续不阻塞"。I/O 密集型线程刚被唤醒（用 CPU 少）→ 高优先级。

## 优先级调度与多级队列




- **4.4BSD 调度器（1990s Unix）**：记录每线程近期 CPU 使用量；**用 CPU 最少的线程优先级最高**。CPU 密集线程积累 CPU 时间 → 优先级下降；等待运行 → 优先级回升（**不会永久饥饿**）；系统过载时退化为最高优先级队列内的 round robin。
- **用户干预**：Unix `nice` 值——`nice -n 19 cmd`（低优先级，最"客气"）、`nice -n -20 cmd`（最高优先级）。

### 多核调度

- 简单方案：共享就绪队列 + 每核私有调度器/定时器；运行优先级最高的 k 个线程；新线程优先级高于当前最低者时用 **IPI（核间中断）** 抢占。
- 问题与改进：
- 就绪队列锁竞争 → **每核独立队列 + 工作窃取（work stealing）**平衡负载；
- 线程在核上积累缓存状态 → **核亲和（core affinity）**尽量不迁移线程；
- **工作守恒（work-conserving）**：有可运行工作就绝不闲置核（理想属性，实现困难）。

### 调度器何时运行?

- 调度器**不是线程**，而是事件驱动的代码：线程解除阻塞时、定时器中断时、收到其它核 IPI 时。

## 代码示例与系统调用解说

### 示例：时间片轮转调度器的核心循环（Assign3 的用户级线程调度器骨架）

```
#include <list>

struct Thread {
    bool blocked = false;
    // ... 保存的寄存器现场、栈指针等 ...
};

class Scheduler {
public:
    void addReady(Thread* t) { readyQueue.push_back(t); }

    // 定时器中断处理器：把当前线程放回队尾，换下一个线程运行
    void onTimerInterrupt(Thread* current) {
        readyQueue.push_back(current);   // 1. 当前线程回队尾
        Thread* next = readyQueue.front();
        readyQueue.pop_front();          // 2. 取队首
        if (next && next != current) {
            contextSwitch(current, next); // 3. 保存 current 现场，恢复 next 现场
        }
    }

private:
    std::list<Thread*> readyQueue;
};
```

**【代码做了什么？】** 定时器中断到来时，调度器把当前线程放回就绪队列尾部，取出队首线程并上下文切换——这就是 round robin。

**【系统机制透视】** – 定时器中断打断用户线程，CPU 陷入内核（/调度器代码），此时**当前线程的寄存器现场保存在它的 TCB 中**；恢复下一个线程时从它的 TCB 装载寄存器——这与你看到的内核级上下文切换是同一机制。 – 时间片过短 → 切换开销占比高（保存/恢复寄存器 + 缓存/TLB 冷启动）；时间片过长 → 交互响应差。Linux 常见时间片约 4ms。 – Assign3 的"虚拟化"视角：一个系统线程被虚拟化成许多用户级线程，调度器就运行在这个系统线程里——这正是内核在多核之间调度系统线程的缩影。

## 关键要点

1. 调度是**政策**：FIFO（简单但会饥饿）→ RR（公平、抢占）→ SRPT（最优但需预知未来）。
2. 用过去预测未来：I/O 密集/交互线程用 CPU 少 → 保持高优先级；CPU 密集线程降级 → 近似 SRPT。
3. 优先级调度 + 老化（等待时优先级回升）避免饥饿。
4. 多核调度的核心难题：共享队列的锁竞争、缓存/核亲和、工作守恒。
5. 调度算法不应改变系统**结果**，只影响效率与响应时间；最好的调度器是自适应的。

## 常见陷阱与注意事项

- **把调度器当成线程**：它是事件驱动代码，运行在中断/陷阱上下文里。
- **时间片设置不当**：过长退化为 FIFO（短作业饥饿），过短则切换开销爆炸。
- **纯优先级调度导致饥饿**：必须配合优先级老化或时间片补偿。
- **忽略核亲和**：频繁迁移线程会因缓存/TLB 冷启动而显著变慢。
- **过度追求公平**：公平不等于最小平均响应时间（RR 对等长作业并不优于 FIFO）。

## 思考题

1. **问题**：为什么 SRPT 能同时改善响应时间和资源利用率？它有什么致命缺陷？
2. **答案**：SRPT 优先运行最短作业，短作业迅速完成（响应时间最优）；偏好 I/O 密集线程意味着磁盘/网络保持忙碌（利用率高）。缺陷是需要知道每个作业的剩余时间（未来），实际无法实现，且长作业可能饥饿。
3. **问题**：4.4BSD 调度器如何保证 CPU 密集线程不被交互线程饿死？
4. **答案**：优先级基于"近期 CPU 使用量"——CPU 密集线程等待运行时它的 CPU 用量在衰减、优先级逐渐回升，最终会成为最高优先级；因此只会"延迟"而不会"永久饿死"。
5. **问题**：多核系统为什么需要每核独立就绪队列和工作窃取？
6. **答案**：单一共享就绪队列会成为锁竞争瓶颈（所有核抢同一把锁）；每核独立队列消除竞争，但负载可能不均，因此用工作窃取：空闲核从繁忙核"偷"线程来执行。

---

# Lecture 9: 链接器与动态链接（Linkers and Dynamic Linking）

## 概述

本讲把注意力从 CPU 转向**主存**：一个进程的地址空间如何从源码一步步构建出来。我们沿着"编译器 → 汇编器 → 链接器（linkage editor）→ 加载器（loader）"的流水线，理解目标文件（object file）的结构、链接器的三遍扫描（pass）、符号解析，以及现代系统中的**共享库与动态链接**（跳转表机制）。

## 核心概念与系统机制图解

### 进程的内存布局

```
地址 0                                    地址 ∞
┌──────────┬──────────┬──────────────────┬──────────┐
│ 代码(text)│ 数据(data)│      堆(heap)     │  栈(stack)│
│ 只读指令  │ 全局/静态  │  动态分配(new/   │ 局部变量/ │
│          │ 变量      │  malloc), 向上增长│  调用帧, 向下增长│
└──────────┴──────────┴──────────────────┴──────────┘
```

```
int global = 7;         → 数据段
int* gptr = &global;    → 数据段(存地址)
void func(int x) {
    int local = x;      → 栈
    int* heap = new int(42);  → 堆
    ...
}
```

- **DRAM 技术参数**：易失、字节寻址（按 64 字节缓存行访问）、访问时间约 60—100ns（200—300 个 CPU 周期）；笔记本 16—64GB，服务器 512GB—4TB（NUMA）。

## 从源码到运行进程的流水线

```
x.c ──gcc(编译)──→ x.s ──as(汇编)──→ x.o ──┐
y.c ──gcc─────────→ y.s ──as─────────→ y.o ──┤ ld(链接)──→ a.out ──OS(加载)──→ 运行中进程
z.c ──gcc─────────→ z.s ──as─────────→ z.o ──┘        + 运行时库(runtime libs)
```

- **目标文件（object file）不完整**：不知道各符号最终落在内存哪里（函数调用地址、外部数据引用是"悬空"的），汇编器用占位值（如 0）。
- **链接器（linkage editor，Linux 的 ld / Windows 的 LINK）的任务**：
- 合并同类段（所有代码段合并、所有数据段合并）；
- 计算内存布局（代码段多大？数据段从哪开始？）；
- 修改地址以匹配布局（填充函数调用与外部数据引用的真实地址）。

## 目标文件的内容

| 内容 | 说明 |
| --- | --- |
| 段（sections） | 代码段（text）、数据段（data）：记录大小、起始地址、初始内容 |
| 符号表（symbol table） | 对外部有意义的外部函数/变量：名字 + 当前位置 |
| 未解析引用（unresolved references） | 本文件引用了但未定义的外部符号：位置 + 需要的地址 |
| 调试信息 | 源码行号（断点）、结构体布局、变量位置 |

## 链接器三遍扫描

```
Pass 1: 读入所有段的大小 → 计算内存布局(内存图)
Pass 2: 读入所有符号 → 在链接器内存中构建完整符号表
Pass 3: 读入段与未解析引用 → 用符号表更新地址 → 写出可执行文件
```

例（Lecture 9 的 main.o + stdio.o + math.o）：

```
内存图:   main.o text: 0..95   stdio.o text: 96..719   math.o text: 720..835
           main.o data: 836...  stdio.o data: ...         (链接后)
符号表:   printf → stdio.o T+44 → 140   sin → math.o T+0 → 720 ...
Pass 3:   main.o 中 offset 30 的 "call 0" → "call 140" (printf 的真实地址)
```

### 动态链接（Dynamic Linking）

- **动机**：静态链接使每个程序都包含整套库（浪费内存）；库更新需重新链接所有程序。
- **方案**：共享库（shared library）——内存中**只有一份**库，所有进程共享；库的加载地址直到程序运行时才知道 → 需要**运行时解析引用**。
- **跳转表（Jump Table）机制**：

```
程序加载时:                          动态链接后:
data段跳转表:                        data段跳转表:
JMP XXX   ← 未解析                   JMP printf ← 填入库函数真实地址
JMP XXX                              JMP sin
JMP XXX                              JMP scanf
程序启动时调用动态加载器(dynamic loader): 扫描跳转表 → 映射共享库 → 填充跳转指令
跳转表条目: {函数名, 所在共享库文件名, 跳转指令}
```

## 代码示例与系统调用解说

### 示例：跨文件符号解析（Lecture 9 经典例子）

```
/* main.c */
extern float sin();
extern printf(), scanf();
int main() {
    double x, result;
    printf("Type number: ");
    scanf("%f", &x);
    result = sin(x);
    printf("Sine is %f\n", result);
}

/* stdio.c */                          /* math.c */
FILE* stdin, stdout;                   double sin(double x) { ... }
int printf(const char* f, ...) { ... }
int scanf(const char* f, ...)  { ... }
```

编译链接：`gcc main.c stdio.c math.c -lm -o sine`（`-lm` 链接数学库，这里演示动态链接）。

**【代码做了什么？】** `main` 引用 `printf` / `scanf` / `sin` 三个外部函数。汇编器在 main.o 中留下三个未解析引用；链接器把 stdio.o 与 math.o 的代码段拼入最终布局，并把三个 `call` 的占位地址替换为真实地址。

**【系统机制透视】** – 若采用**动态链接**：`sin` 位于共享库 `libm.so`，链接时只在 main.o 的数据段生成跳转表条目；程序启动时动态加载器把 libm.so 映射进地址空间并填充跳转指令，后续所有 `call sin` 都经由跳转表间接跳转。 – 优点：多进程共享一份库代码（内存节省）、库升级无需重编程序；代价：一次间接跳转 + 启动时解析开销。 – 你每天用的 `ldd`、`LD_LIBRARY_PATH`、`dlopen` 都与这一讲直接相关。

## 关键要点

1. 进程内存布局 = 代码 + 数据 + 堆（向上）+ 栈（向下）；链接器负责决定各段的位置。
2. 目标文件携带"段 + 符号表 + 未解析引用"，链接器三遍扫描完成合并、布局、重定位。
3. 静态链接产生完整独立程序；动态链接通过跳转表在运行时解析共享库引用。
4. 共享库 = 内存中一份库被多进程共享；加载时重定位。
5. 堆栈增长方向、动态内存分配调用（malloc/new 会向 OS 请求扩大数据段）都与本讲的内存布局有关。

## 常见陷阱与注意事项

- **未定义符号**：链接器报 "undefined reference"——符号表里没有该符号。
- **多重定义**：两个目标文件定义了同名全局符号。
- **忘记链接库**：如 `-lm`、`-lpthread`；C++ 还需 `-lstdc++`。
- **把动态库路径搞错**：`LD_LIBRARY_PATH` 设置不当导致运行时 "cannot open shared object"。
- **循环依赖的共享库**：链接时库顺序错误。

## 思考题

1. **问题**：为什么汇编器不能直接生成最终可执行文件?
2. **答案**：汇编器单独处理一个源文件，不知道其它文件的代码/数据最终落在内存何处，也不知道库函数地址，因此只能留下占位符和未解析引用；只有链接器掌握全部目标文件后才能计算布局、解析所有引用。
3. **问题**：动态链接相比静态链接的主要代价是什么?
4. **答案**：调用需经跳转表间接跳转（一次额外跳转），程序启动时需运行动态加载器解析库引用；另外共享库版本不兼容（DLL hell）可能引入运行时故障。

---

# Lecture 10—11: 动态存储管理（Dynamic Storage Management）

## 概述

本讲研究"如何管理一块内存/存储区域以满足各种分配需求"——既适用于应用中的堆，也适用于操作系统与磁盘空间管理。核心挑战是**不可预测性**：不知道一个已分配块何时会被释放。我们从栈分配（LIFO，简单高效）讲到堆分配（任意顺序，困难），覆盖空闲链表（first fit/best fit）、slab 分配器、位图，以及存储回收的两种策略（引用计数与垃圾回收）。

## 核心概念与系统机制图解

### 动态存储的两个基本操作

```
allocate(size) → ptr    分配 size 字节, 返回指针
free(ptr)               释放之前分配的块
挑战: 不可预测—不知道分配出去的块多久后被释放 → 极难
```

### 栈分配（Stack Allocation，LIFO）

- **适用**：分配/释放满足"后分配先释放"（LIFO）——可预测。
- **实现**：一个栈指针。分配 = 调整指针；释放 = 指针调回去。极高效。
- **示例**：过程调用（X 调 Y，Y 再调 Z）、树的遍历、表达式求值、递归下降解析。
- **特点**：已分配空间连续、空闲空间连续、无碎片化。代价：要求可预测的 LIFO 模式。

### 堆分配（Heap Allocation）

- **适用**：释放顺序不可预测（树、图、复杂数据结构）。
- **问题**：内存被分割成已分配块与空闲块（洞/holes）——**碎片化（fragmentation）**。
- **目标**：让"洞"的数量少、尺寸大。

### 空闲链表（Free List）策略




free list: 把所有洞串成链表
- First fit: 从链表头开始，找到第一个足够大的洞
- Best fit : 扫描全链表，返回最接近需求尺寸的洞，剩余放回链表
- 释放时: 与相邻空闲块合并

问题: 洞会越切越碎(碎片化)，大分配失败或堆被迫增长，扫描开销大

### Slab 分配器

- **定义**：把一块内存（slab）切成**等大小的块**；为每个常用分配尺寸准备一个池（pool），每个池有独立空闲链表。




分配: 去对应尺寸的 slab 的空闲链表取一块; 池空则申请新 slab 切块
释放: 归还到所在 slab 的空闲链表; slab 全空可整体释放

- **优点**：常见情形下分配/释放极快；操作系统广泛使用（内核对象池，如进程描述符按 struct 大小分池）。
- **缺点**：**内部碎片**——slab 内未被使用的空间；分配尺寸频繁变化时尤其浪费。

### 位图（Bitmap）

- **定义**：一个比特数组，每个比特表示一块固定大小内存（或磁盘块）的分配状态（0=空闲，1=已分配）。
- 适合管理**固定大小块**（slab 内部、磁盘块）；查找空闲块需扫描比特数组（文件系统讲座详述）。

### 存储回收（Reclamation）：何时能释放?

- 原则：只有"不再被访问"的内存才能释放；假设只有"有指针指向"的数据才可访问。
- 两个经典错误：
- **悬垂指针（Dangling Pointer）**：释放太早——内存还在使用。
- **内存泄漏（Memory Leak）**：没人释放——内存永远无法再用。
- 两种自动方案：

| 方案 | 机制 | 优点 | 缺点 |
| --- | --- | --- | --- |
| **引用计数** | 每个对象记录指向它的指针数，归零即释放 | 即时回收、实现简单（`std::shared_ptr`、文件系统 inode 链接数） | **无法处理循环引用**（A 指 B、B 指 A，计数永不归零） |
| **垃圾回收（GC）** | 不显式 free；GC 扫描找出所有"活"对象（从根可达），回收其余 | 无悬垂指针、可压缩内存消除碎片（Java/Go/JS） | 昂贵：占 10–20% CPU、2–5 倍内存超分配、回收时停顿 |

### 标记–清扫（Mark and Sweep）图解

```
Pass 1 (Mark): 从根(静态变量/局部变量)出发, 递归标记所有可达对象
Pass 2 (Sweep): 扫描所有对象, 把活对象拷贝到连续内存(可压缩), 更新指针, 释放其余
根 ──► A ──► B ──► C        D(不可达→回收)
```

## 代码示例与系统调用解说

### 示例：悬垂指针 vs 引用计数（C++）

```
#include <memory>
#include <iostream>

struct Node { int val; };

int main() {
    // --- 悬垂指针 ---
    int* p = new int(42);
    delete p;             // 释放过早
    std::cout << *p;      // 未定义行为: 悬垂指针解引用

    // --- 引用计数: std::shared_ptr ---
    std::shared_ptr<Node> a = std::make_shared<Node>();
    {
        std::shared_ptr<Node> b = a;    // 引用计数 1 → 2
        // ... 使用 a/b ...
    }                                   // b 析构, 计数 2 → 1
    // a 仍在, 对象存活
    // a 离开作用域 → 计数 0 → 自动释放
    return 0;
}
```

**【代码做了什么？】** 上半部分演示悬垂指针（释放后仍使用）；下半部分用 `std::shared_ptr` 让对象生命周期由引用计数自动管理。

**【系统机制透视】** – 引用计数的开销：每次拷贝/析构智能指针都有原子加减；循环引用（如双向链表、父子互指）会导致计数永不归零 → 泄漏（C++ 需用 `std::weak_ptr` 打破环，Python 需 GC 的循环检测器）。 – 与操作系统内存管理的关系：文件系统 inode 的 `nlink`（链接计数，Lecture 22）就是引用计数的经典系统级应用——目录条目全部删除后才真正释放文件数据。 – 堆管理的宏观流程：`malloc` / `new` 先查空闲链表/slab；堆空间不足时通过 `brk` / `mmap` 系统调用向 OS 申请扩大数据段/映射新区域（与 Lecture 9 的内存布局衔接）。

## 关键要点

1. 栈分配（LIFO）简单高效零碎片，但要求可预测的分配顺序；堆分配支持任意顺序但面临碎片化。

2. 空闲链表策略（first fit/best fit）简单但易碎片化、扫描开销大；slab 按尺寸分池大幅提速。
3. 位图适合固定大小块的分配管理（也是磁盘空闲空间管理的基础）。
4. 回收错误的两面：悬垂指针（过早释放）与内存泄漏（过晚释放）。
5. 引用计数即时但怕环；GC 无悬垂但昂贵——系统设计需在两者间权衡。

## 常见陷阱与注意事项

- **释放后使用（use-after-free）**：悬垂指针解引用是未定义行为，valgrind/ASan 可检测。
- **内存泄漏**：忘 delete / 循环引用使 shared_ptr 计数不清零。
- **只 new 不 delete 的大循环**：堆无限增长直至 OOM。
- **碎片化导致的大块分配失败**：即使总空闲空间足够，也没有连续大块。
- **slab 尺寸假设失效**：分配尺寸分布变化时 slab 内部碎片激增。

## 思考题

1. **问题**：为什么栈分配没有碎片化而堆分配有?
2. **答案**：栈严格 LIFO，所有已分配空间与所有空闲空间各自连续；堆任意顺序分配/释放，空闲块与已分配块交错，形成大量小洞（碎片）。
3. **问题**：引用计数最大的缺陷是什么？给出一个例子。
4. **答案**：循环引用。两个对象互相持有对方指针时，各自计数都 ≥1，永不归零，内存泄漏。例如双向链表相邻节点互指、A 对象持有 B、B 持有 A。
5. **问题**：slab 分配器能消除碎片化吗?
6. **答案**：不能。它消除了"不同尺寸块交错"造成的搜索开销，但 slab 内未使用的空间（内部碎片）依然存在；分配尺寸变化频繁时浪费更明显。

---

# Lecture 12: 信任与操作系统（Trust and Operating Systems）

## 概述

本讲从哲学与工程角度探讨**信任**：操作系统是软件世界的"信任之根"（trusted computing base），所有应用都依赖它提供安全与正确执行。我们学习信任的定义、建立信任的三种方式（假设/推断/替代），并通过 Linux 信任体系与 2024 年 xz/ssh 供应链攻击案例，理解过度信任（over-trust）的风险。

# 核心概念与系统机制图解

## 什么是信任?

- **定义**（AI 伦理文献）："一方愿意对另一方的行为保持脆弱（vulnerable），基于对对方将执行对信任方重要之行动的期望，无论信任方能否监控或控制对方。"
- **哲学版本**：信任是一种**不加质疑的态度**——我们停止质疑其可靠性，假定它会正常工作。
- **为什么需要信任**：
- **扩展能动性（agency）**：信任让你能做超出直接控制范围的事；
- **提高效率**：不必事事亲力亲为；
- **保持心智健全（sanity）**：不必持续担忧。
- **风险**：信任带来依赖与脆弱性；信任被违背可能造成严重伤害（例如 OS 静默断开蓝牙连接，导致血糖监测仪停止报警）。

## 过度信任 vs 不值得信任

- **过度信任（over-trust）**：信任方把信任扩展到不合理范围（信任方的问题）。
- **不值得信任（untrustworthiness）**：受托方客观上缺乏诚信/可靠性（受托方的问题）。

## 建立信任的三种方式

| 方式 | 机制 | 强度 |
| --- | --- | --- |
| **假设（Assumption）** | 没有证据地信任（如有人喊"小心！车来了"） | 弱、有风险 |
| **推断（Inference）** | 依据过往行为/特征推断（用得久、口碑好、品牌） | **最强** |
| **替代（Substitution）** | 有后备方案兜底（杂技演员信任搭档因为下面有安全网） | 取决于后备方案本身可信 |

## 软件中的信任：操作系统是信任之根

```
应用 ──依赖──► 操作系统内核 ──依赖──► 硬件/固件(BIOS)
               (trusted computing base)
应用的安全性不会超过它所运行的操作系统
```

- **软件如何建立信任**：假设方式不可用；**推断 = 通过"不信任"走向信任**（测试、验证、插桩、代码评审）；**替代 = 出错时检测并纠正**（日志、一致性检查、超时、冗余）。
- **确认偏误（Confirmation Bias）**：看起来正常工作的系统让我们停止质疑——一种过度信任。

## 案例：2024 xz/ssh 供应链攻击

```
攻击者伪装成开源开发者 "Jia Tan" 数年 → 逐步获得 xz 维护权
→ 在 xz 中植入后门（同时操纵 OSS-Fuzz 扫描器禁用检测）
→ sshd 依赖 systemd → systemd 依赖 xz 压缩库
→ 动态链接时后门被带入 sshd → 可远程接管任何 Linux 系统
→ 仅因有人好奇 sshd 的小延迟而偶然发现
教训：强信任体系也会因过度信任而失败
```

- **Linux 对 AI 生成代码的政策**（2026 年 4 月）：AI 可协助写代码，但**只有人类可以贡献代码**，人类对每一行负全责。

## 代码示例与系统调用解说

### 示例：信任边界与最小权限（代码层面的"替代"思想）

```
#include <cstdio>
#include <sys/types.h>
#include <unistd.h>

// 演示：程序启动后立即放弃特权（drop privileges）
int main() {
    if (getuid() == 0) {
        // 需要 root 才能做的事，做完立即降权
        setuid(1000);   // 放弃 root 特权
        setgid(1000);
    }
    // 之后以普通用户身份运行——即使被攻破，损害也被限制在信任边界内
    FILE* f = fopen("/etc/passwd", "r");  // 普通用户无法写系统关键文件
    if (!f) perror("open denied (by design)");
    return 0;
}
```

**【代码做了什么？】** 程序在完成特权操作后调用 `setuid` / `setgid` 降权，以最小权限继续运行。

**【系统机制透视】** – 这是"**最小权限原则**"的实践：把信任边界尽量缩小。操作系统通过用户/组 ID、权限位、`capabilities` 实施这一边界——内核本身是唯一真正"全权"的软件。 – 与 Lecture 12 主题的联系：我们信任内核执行权限检查，正是"推断 + 替代"的结合——公开源码、代码评审（推断），加上权限检查失败时的显式报错（替代）。

## 关键要点

1. 信任 = 愿意脆弱 + 期望对方尽责 + 无法监控；它扩展能动性但也带来风险。
2. 信任的三种建立方式：假设（弱）、推断（最强）、替代（依赖后备方案）。
3. 操作系统是信任之根（trusted computing base）：应用的安全性不可能超过其 OS。
4. 软件的信任靠"通过不信任建立信任"：测试、评审、日志、一致性检查、冗余。

5. 供应链攻击（xz/ssh）证明：过度信任 + 确认偏误可攻破看似健全的信任体系。

## 常见陷阱与注意事项

- **确认偏误**：系统"看起来正常"就不再审视。
- **过度信任第三方组件**：每个依赖（库、日志系统）都是攻击面。
- **特权不降权**：常以 root 运行、用后不释放权限。
- **没有后备/监控**：缺乏日志与一致性检查，错误无法被发现和纠正。

## 思考题

1. **问题**：为什么说"推断是建立信任的最强方式"?
2. **答案**：推断基于可验证的证据（过往行为、构造方式、口碑），比无条件假设可靠；比替代更根本——替代只是把信任转移给后备系统，而推断直接评估受托方本身的可信度。
3. **问题**：xz/ssh 攻击中，信任链是如何被利用的?
4. **答案**：ssh 开发者信任 systemd 日志系统，systemd 信任 xz 压缩库，而 xz 被长期渗透的"合法"维护者植入了后门——信任链上任何一环的过度信任都被攻击者利用，且攻击者还操纵了扫描工具禁用检测（破坏"替代"防线）。

---

# Lecture 13–14: 虚拟内存（Virtual Memory）

## 概述

本讲把虚拟化思想应用到**内存**：让多个进程共享同一物理内存，同时获得多任务、透明、隔离与效率。我们沿历史演进：单任务 → 加载时重定位 → **基址/界限（base and bound）**（第一个硬件 MMU）→ **分段（segmentation）**，理解每步解决什么问题、又留下什么局限，为下一讲的分页（paging）铺路。

## 核心概念与系统机制图解

### 内存共享的目标（Lecture 13 的评分表）

| 目标 | 含义 |
| --- | --- |
| **多任务（Multitasking）** | 多个进程可同时驻留内存 |
| **透明（Transparency）** | 进程察觉不到共享——每个进程都像独占内存 |

| 目标 | 含义 |
| --- | --- |
| **隔离（Isolation）** | 进程不能破坏彼此或 OS |
| **效率（Efficiency）** | 共享不严重损害 CPU 与内存效率 |

## 演进路线与评分

```
单任务(批处理/MS-DOS)          加载时重定位                  基址/界限(MMU)
OS 独占高地址，一次一程序       加载器像链接器一样改地址      硬件寄存器: base + bound
多任务 ✗  透明 ✗  隔离 ✗        多任务 ✓  透明 ✗  隔离 ✗     多任务 ✓  透明 ✓  隔离 ✓
效率 ✓                          效率 ✓(但碎片/无法增长)      效率 ✓
  局限: 不能共享，坏程序        局限: 尺寸静态声明、          局限: 每进程一个连续区域，
  能破坏 OS                      无隔离、碎片化、不能移动      不支持共享/只读代码、碎片化
```

## 动态地址翻译（Dynamic Address Translation）

- **核心思想**：程序看到的是**虚拟地址空间（virtual address space）**，硬件 MMU 在每次内存访问时把它翻译成**物理地址空间（physical address space）**中的地址。

```
CPU ──虚拟地址──► MMU ──物理地址──► 内存
                   │
                   └─(同时检查越界/保护)
```

- 每个进程有独立的虚拟地址空间（都从 0 开始）；物理地址空间被 OS 划分给各进程。

## 基址/界限（Base and Bound）MMU

```
两个硬件寄存器:
  Base  = 进程在物理内存中的起始位置
  Bound = 进程虚拟地址空间的大小
每次访问(并行完成):
  物理地址 = 虚拟地址 + Base
  若 虚拟地址 >= Bound → 越界故障(fault)
```

**示例**：虚拟地址 140，Base=6000 → 物理地址 6140；程序里所有地址（PC、SP、数据引用）都经 MMU 加上 Base，程序本身无需重定位——这就是"动态"重定位。

## 进程 ⇔ OS 切换时的地址翻译

- OS 运行时**关闭地址翻译**（虚拟地址 = 物理地址），通过处理器状态寄存器（PSR）的位控制"翻译开/关、用户/内核模式"。
- **陷入（trap）OS 时原子地**：保存程序计数器 → 跳到中断向量（interrupt vector）指向的 OS 入口 → 关闭翻译与用户模式。
- **从 trap 返回时原子地**：恢复翻译与用户模式 → 恢复保存的 PC。

### 分段（Segmentation）：多个区域

- **动机**：一个连续区域不够用——程序由代码、数据、栈等多段组成，需要分别管理（分别增长、交换、共享、保护）。
- **机制**：段表（segment map），每段有 base、bound、保护位（如代码段只读）。

```
段表(每进程):
 段号  类型    Base    Bound  保护
 0     Code   1000    1000    R/O
 1     Data   3000    2000    R/W
 2     Stack  8000    2000    R/W
地址高位选段，低位是段内偏移；或由指令类型隐式选段(x86 段前缀)
```

- **优点**：各段独立增长/交换、可移动压缩消除碎片、可共享（共享代码段）。
- **缺点**：段数量固定仍有限制（无法 mmap 文件）；变长段仍有碎片；地址空间划分僵化。

## 代码示例与系统调用解说

### 示例：查看进程的虚拟内存布局（Linux /proc）

```
# 编译并运行一个简单程序
./myprogram &
# 查看其虚拟地址空间布局
cat /proc/$!/maps
# 输出示例（每行: 虚拟地址范围   权限   偏移   设备    inode   路径）
# 00400000-00401000 r-xp 00000000 08:01 12345 /home/user/myprogram   ← 代码段(只读可执行)
# 00600000-00601000 r--p 00000000 08:01 12345 /home/user/myprogram   ← 数据段(只读)
# 00601000-00602000 rw-p 00001000 08:01 12345 /home/user/myprogram   ← 数据段(可写)
# 7ffc00000000-7ffc00021000 rw-p 00000000 00:00 0 [stack]           ← 栈
# 7f0000000000-7f0000020000 r-xp ... /lib/x86_64-linux-gnu/libc.so.6 ← 共享库代码
# ... [heap] ...
```

**【代码做了什么？】** `/proc/<pid>/maps` 打印进程虚拟地址空间中每个映射（代码、数据、堆、栈、共享库）的虚拟地址范围与权限。

**【系统机制透视】** – 这就是"虚拟地址空间"的真实面貌：进程"以为"自己独占从 0x0 到 0x7fffffffffff 的地址空间，但物理内存中这些页可以散布在任何位置（分页，下一讲）。 – 权限位（r-x、rw-p）就是 MMU 执行保护检查的依据：试图写入只读页会触发段错误（segmentation fault）——隔离与保护正是这样硬件化的。 – 共享库出现在每个进程同一虚拟地址（如 libc），物理上只有一份——分段/分页的共享机制。

## 关键要点

1. 虚拟化内存 = 让每个进程拥有从 0 开始的私有虚拟地址空间，MMU 动态翻译为物理地址。

2. 基址/界限是第一个简单 MMU：一次加法和一次比较，同时实现重定位、隔离、透明。
3. 基址/界限的局限（一个连续区域）催生分段：多段独立管理、可共享、可保护。
4. 分段的局限（固定段数、碎片）催生分页：固定大小页，无外部碎片。
5. trap 进出 OS 时原子地切换"翻译开关 + 用户/内核模式"。

## 常见陷阱与注意事项

- **混淆虚拟地址与物理地址**：程序里所有指针都是虚拟地址。
- **以为进程能直接访问物理内存**：用户进程永远看不到物理地址（除 OS 的特殊映射）。
- **越界与保护**：基址/界限和段表都检查越界；忘记保护位（如代码段可写）会破坏隔离。
- **把"重定位"理解成加载期一次性完成**：base-and-bound 是每次访问动态完成，程序无需修改。

## 思考题

1. **问题**：基址/界限机制中，为什么程序无需修改任何地址就能在任意位置运行?
2. **答案**：MMU 在每次内存访问时把虚拟地址加上基址寄存器，程序看到的是从 0 开始的连续虚拟空间；物理位置由基址决定，与程序内容无关——"动态重定位"。
3. **问题**：分段相比基址/界限解决了什么问题，又引入了什么新问题?
4. **答案**：解决"单一连续区域"限制——各段独立增长/交换/共享/保护；引入"段数固定、变长段的外部碎片、地址空间划分僵化"等新问题。
5. **问题**：OS 运行时为什么关闭地址翻译?
6. **答案**：OS 代码与数据结构直接操作物理内存（早期设计），关闭翻译后虚拟地址=物理地址，便于内核管理物理资源；进出 trap 时切换 PSR 位实现。

---

# Lecture 15: 分页（Paging）

## 概述

分页把虚拟与物理地址空间都切成**固定大小的页**，用**页表（page map/page table）**映射虚拟页到物理页，彻底消除外部碎片并支持任意稀疏的地址空间。本讲深入 x86-64 的四级页表（PML4→PML3→PML2→PML1）翻译过程，讨论页表共享（2MB/1GB 大页）、**TLB**（翻译后备缓冲器）以及内部碎片。

# 核心概念与系统机制图解

## 分页的基本思想

- 虚拟与物理地址空间都划分成固定大小块：**页（page）**。常见 4KB（x86，myth 机器）或 16KB（MacBook）。
- **页表（Page Table / Page Map）**：把虚拟页号（VPN）映射到物理页号（PPN）的表；每进程一张。

```
虚拟地址空间              页表(Page Map)              物理内存
VPN 0 ──────────────► ┌──────────┐ ──────────► PPN 0
VPN 1 ──────────────► │ VPN→PPN  │ ──────────► PPN 1
VPN 2 ──────────────► │ 映射条目 │ ──────────► PPN 3
...                   │ (PTE)    │              ...
VPN n ──────────────► └──────────┘              PPN n
页表条目(PTE)通常含: PPN + Present(存在位) + Writeable(可写位) + 其它标志
虚拟地址 = [VPN][offset]    物理地址 = [PPN][offset]
```

## 固定大小的好处

- 物理内存管理：OS 维护空闲物理页链表——分配 = 取一页，释放 = 还一页（无碎片）。
- 虚拟地址空间管理：程序的一段（segment）就是一组页，可从任意页边界开始。

## x86-64 四级页表（为什么需要多级？）

- 64 位虚拟地址、4KB 页、8 字节 PTE → 若用**单级**页表：`2^36 项 × 8B = 512GB`/进程 —— 不可行！
- **方案**：把页表本身也按页切分（每页 512 个 8B 条目），再加一级索引……逐级嵌套：

```
64位虚拟地址: [PML4(9位)][PML3(9位)][PML2(9位)][PML1(9位)][offset(12位)]
   │            │           │           │           │
   ▼            ▼           ▼           ▼           ▼
PML4基址寄存器 → PML4表(512项) → PML3表(512项) → PML2表(512项) → PML1表(512项) → 物理页
每级表恰好一页(4KB = 512×8B); 有 Present 位的级可省略(稀疏地址空间只分配用到的表)
```

- **翻译示例**（Lecture 15）：
- 访问代码页 0x0：PML4[0x0]→PML3[0x0]→PML2[0x0]→PML1[0x0]→PTE 得到物理页；
- 访问栈页 0xFFFFFFFFF000：四级索引全是 0x1FF（512）——栈页在虚拟空间顶端；
- 访问数据页 0x1000：PML4[0]→PML3[0]→PML2[0]→PML1[1]。
- **这像什么数据结构？** 像一棵 4 层 512 叉树——只有被访问的路径才分配页表页。

## 内存共享（x86-64）

- 两个进程的 PML1 条目指向**同一个物理页** → 共享 4KB；
- 让两个 PML2 指向同一张 PML1 表 → 共享 2MB（512×4KB）；

- 同理可共享 1GB、512GB 对象 → **大页（2MB/1GB）**天然支持。

### TLB（Translation Lookaside Buffer）

- **问题**：一次访问要查 4 级页表 = 4 次内存访问，太慢！
- **方案**：在 MMU 里放一个小而快的**翻译缓存**：VPN→PPN 映射。
- 典型容量 64–2048 条目；全相联；命中率通常 >95%（局部性）。
- **命中**：直接用 PPN + 检查保护位；**未命中**：走页表并填充 TLB。
- **OS 相关操作**：
- 切换页表（上下文切换）时必须**使 TLB 失效**（x86 写 PML4 基址自动刷新；有的架构给 TLB 条目加 PID 避免全刷）；
- 修改页表后需用 `INVLPG` 指令使对应 TLB 条目失效。

### 碎片化的两种形式

- **外部碎片**：进程之间的碎片——分页用固定大小页**消除**了它。
- **内部碎片**：页内部未使用的空间——分页**引入**了它（每段最后一页平均浪费半页）；页越大内部碎片越严重。

## 代码示例与系统调用解说

### 示例：mmap 映射文件并访问（分页的用户可见形态）

```
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <cstdio>
#include <cstring>

int main() {
    int fd = open("data.bin", O_RDWR | O_CREAT, 0644);
    if (fd < 0) { perror("open"); return 1; }
    ftruncate(fd, 4096);                        // 保证文件有一页大小

    char* p = (char*)mmap(nullptr, 4096,        // 映射 4KB（一页）
                          PROT_READ | PROT_WRITE,
                          MAP_SHARED, fd, 0);
    if (p == MAP_FAILED) { perror("mmap"); return 1; }

    strcpy(p, "hello from mmap");               // 写：触发缺页→分配物理页→读文件内容
    printf("read back: %s\n", p);

    msync(p, 4096, MS_SYNC);                    // 把脏页写回文件
    munmap(p, 4096);                            // 解除映射
    close(fd);
    return 0;
}
```

**【代码做了什么？】** `mmap` 把文件内容映射进进程虚拟地址空间；访问映射区域如同访问内存；`msync` 强制写回。

**【系统机制透视】** – `mmap` 建立"虚拟页 ↔ 文件块"的页表条目（present 位初始为 0）；首次读写触发**缺页中断**，内核分配物理页、从文件读入数据、置 present 位（Assign5 就是用户态模拟这一流程！）。 – `MAP_SHARED` 下多个进程映射同一文件 → 共享同一物理页（页表级共享）；`MAP_PRIVATE` 则用写时复制隔离。 – 这展示了分页的完整价值：文件可以被当作内存访问、按需加载、跨进程共享——而这一切对用户透明。

## 关键要点

1. 分页 = 固定大小页 + 页表；消除外部碎片，支持稀疏、可共享的地址空间。
2. x86-64 用四级页表（PML4→PML1）避免单级页表 512GB 的荒谬开销；只有用到的路径才分配页表页。
3. TLB 缓存 VPN→PPN，命中率 >95%；上下文切换需使其失效。
4. 共享可通过"页表条目指向同一物理页"实现，天然支持 4KB/2MB/1GB 大页。
5. 分页把外部碎片换成内部碎片（每段最后一页最多浪费一页）。

## 常见陷阱与注意事项

- **把页表大小算成单级**：x86-64 必须 4 级，否则内存爆炸。
- **忘记 TLB 失效**：修改页表后不刷新 TLB 会用到陈旧映射（安全与正确性双重问题）。
- **混淆内部/外部碎片**：分页消除外部、引入内部。
- **页大小选择**：页越大页表越小、I/O 越高效，但内部碎片越大。

## 思考题

1. **问题**：为什么 x86-64 不用一张巨大的单级页表?
2. **答案**：64 位地址 + 4KB 页 → 2^52 个潜在页，8B 条目需要 2^55 字节（32PB）；而四级页表只需分配实际使用的路径，一个只用了 3 页的程序只需 4 张页表页（约 16KB）。
3. **问题**：TLB 命中为什么能大幅加速内存访问?
4. **答案**：未命中需 4 次页表内存访问；命中只需一次 TLB 查找（几纳秒）。局部性使 TLB 命中率超 95%，平均访问时间接近一次快速查找。
5. **问题**：两个进程如何共享一个 2MB 的只读库?
6. **答案**：让两个进程 PML2 表的同一项指向同一张 PML1 页表，从而共享其下全部 512 个 4KB 页（共 2MB）；物理上只有一份库代码。

---

# Lecture 16–17: 请求调页（Demand Paging）

## 概述

真正的"虚拟"内存允许程序**不把所有信息装入内存**也能运行：用到的页在内存，闲置的页在磁盘（交换区/backing store），按需搬移。本讲讲解**缺页机制（page fault）**、取页策略（按需取页 vs 预取）、页面置换策略（随机/FIFO/MIN/LRU/时钟算法），以及内存过载导致的**抖动（thrashing）**——这是 Assign5/Assign6 的理论核心。

## 核心概念与系统机制图解

### 为什么能这么做？——局部性（Locality of Reference）

- 大多数程序大部分时间只使用其代码和数据的一小部分。
- 内存（DRAM）比磁盘快约 10 万倍、比 SSD 快约 1000 倍；磁盘/SSD 每比特成本低约 100 倍。理想：**像磁盘一样便宜、像 DRAM 一样快**的"虚拟内存"。

## 缺页机制（Page Fault）

- 页表条目中的 **Present 位 = 0** 表示该页在 backing store（交换空间/分页文件）。
- CPU 访问 present 位为 0 的虚拟地址 → **缺页陷阱（page fault trap）** → OS 处理：

```
1. 检查访问是否合法(否则 → 段错误, 终止进程)
2. 找一个空闲物理页(若无, 先按置换策略淘汰一页)
3. 从 backing store / 可执行文件读入该页
4. 更新页表条目: 指向新物理页, 置 Present 位
5. 恢复执行触发缺页的指令(指令必须可重启!)
```

- **硬件支持**：x86-64 把出错地址锁存到特权寄存器 CR2；指令需可重启（如 `push` 会先改 SP 再写内存，缺页时要能回滚）。

## 取页策略（Fetching Policy）

| 策略 | 做法 | 评价 |
| --- | --- | --- |
| **按需取页（Demand Fetching）** | 进程启动时**一页都不加载**，引用到才取 | 简单；只读代码页从可执行文件取、未初始化数据/栈返回**零页**、脏数据页写回 backing store |
| **预取（Prefetching）** | 预测未来需要的页提前加载 | 需预测未来，难；折中：缺页时多读几页（顺序访问时有效）。磁盘缺页 5–10ms，预取 .04ms |

## 页面置换策略（Replacement Policy）

- 内存满后每次缺页都必须淘汰一页： | 策略 | 做法 | 评价 | | :--- | :--- | :--- | | **Random** | 随机选页 | 简单、出乎意料地有效 | | **FIFO** | 淘汰在内存最久的页 | 简单、对页公平；可能淘汰常用页 | | **MIN（最优）** | 淘汰**未来最久才被访问**的页 | 理论最优，需预知未来，不可实现 | | **LRU** | 淘汰**最久未使用**的页 | 用过去预测未来，应近似 MIN |
- **Lecture 16 的对比实验**（引用序列 A..E 各若干次，3 个页框）：
- FIFO：**10 次缺页**；MIN（最优）：**6 次**；LRU：**8 次**——LRU 明显优于 FIFO，接近最优。

## LRU 的实现难题与时钟算法（Clock Algorithm）

- 精确 LRU 需要硬件记录每页的访问时间戳——成本过高，不实用。
- **实用硬件支持**：页表条目中两个位——**引用位（referenced/accessed）**：页被读/写时由 CPU/MMU 置位；**脏位（dirty）**：页被修改时置位。
- **时钟算法（Clock / 第二机会 Second Chance）**——LRU 的近似：

```
把物理页排成圆环，一个"指针(hand)"指向某页：
缺页需要淘汰页时：
  while (true):
    若 当前页引用位 = 1:  清除引用位(给第二次机会)，指针前移
    若 当前页引用位 = 0:  选中该页淘汰！
                          (若脏位=1，先写回磁盘)
                          指针前移
  ┌──────────────────────────────────┐
  │  P1 ── P2 ── P3 ── P4 ── P5  │   ← 环形链表 + hand
  └──────────────────────────────────┘
```

- **hand 速度的含义**：慢 = 内存充足、缺页少；快 = 内存不足、缺页频繁（Assign6 直接实现该算法）。

### 全局 vs 每进程置换

- **全局置换**：所有进程的页放进同一个置换池，彼此竞争（无性能隔离；大多数系统采用）。
- **每进程置换**：每进程独立页框池，互不干扰（需决定每进程分多少页框）。

### 抖动（Thrashing）

- **定义**：活动工作集（working set）超过物理内存 → 每次缺页淘汰的都是活跃页 → 立刻又缺页 → 几乎所有时间都在换页。
- **数学感受**（Lecture 16）：DRAM 100ns，磁盘 10ms。若内存仅差 1%（每 100 次访问 1 次缺页）：`0.99×100ns + 0.01×10ms ≈ 100,099ns` —— **慢了 1000 倍**！
- **对策**：OS 暂停部分进程（调度器只调度"装得下"的作业集）；个人电脑上用户可自行关闭程序；内存便宜 → 买够内存。

## 代码示例与系统调用解说

### 示例：用 mprotect + SIGSEGV 模拟缺页（Assign5 的核心思路）

```
#include <csignal>
#include <cstdio>
#include <cstdlib>
#include <sys/mman.h>
#include <unistd.h>

char* region;

void faultHandler(int, siginfo_t* si, void*) {
    // 捕获访问受保护页的 SIGSEGV —— 用户态版本的"缺页中断"
    char* faultAddr = (char*)si->si_addr;
    long page = ((long)faultAddr - (long)region) / 4096;
    printf("[page fault] loading page %ld from backing store\n", page);
    // (Assign5 中这里: 分配物理页→解密读入→mprotect 置可读写)
    mprotect(region + page * 4096, 4096, PROT_READ | PROT_WRITE); // 模拟"置 Present 位"
}

int main() {
    region = (char*)mmap(nullptr, 4 * 4096, PROT_NONE,
                         MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    struct sigaction sa{};
    sa.sa_sigaction = faultHandler;
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGSEGV, &sa, nullptr);

    region[0] = 'A';          // 首次访问 → 故障 → 按需"取页"
    region[5000] = 'B';       // 第二页 → 再次故障
    printf("done: %c %c\n", region[0], region[5000]);
    return 0;
}
```

**【代码做了什么？】** 用 `PROT_NONE` 映射 4 页，任何访问都触发 SIGSEGV；处理器把页"加载"出来并 `mprotect` 为可读写，然后程序继续——完美模拟"缺页 → 取页 → 继续执行"。

**【系统机制透视】** – 真实内核的缺页处理与这段代码同构：内核在缺页陷阱里分配物理页、从 backing store 读入、更新页表（置 Present）、恢复指令执行。 – `mprotect` 用页表保护位制造"伪缺页"——这正是 Assign5 在**用户态**模拟内核特性的技巧（讲义明确说"本应在内核里写的代码，我们用 mmap 在用户态实现"）。 – 注意"指令可重启"：真实硬件保证缺页返回后重新执行同一指令；用户态模拟里，SIGSEGV 处理器返回后也会重试出错指令。

## 关键要点

1. 请求调页让程序无需全部装入内存即可运行：Present 位 + 缺页陷阱是核心机制。
2. 取页策略：按需取页（默认）vs 预取（赌顺序访问）。
3. 置换策略：MIN 最优但不可实现；LRU 近似 MIN；时钟算法用引用位近似 LRU（Assign6 实现）。
4. 引用位/脏位是页表条目中的关键硬件支持。

5. 抖动 = 工作集超内存 → 换页风暴 → 千倍减速；需控制并发度或加内存。

## 常见陷阱与注意事项

- **忘记检查缺页地址合法性**：非法访问（如空指针）会无限缺页或错误地"取页"。
- **脏页不写回**：淘汰脏页前必须写回 backing store，否则数据丢失。
- **置换掉正在使用的页**：会导致立即再次缺页（抖动前兆）。
- **时钟算法中忘记清除引用位**：指针将永远找不到可淘汰页。
- **把 backing store 与文件系统缓存混淆**：backing store 是页的"影子"，独立于文件缓存。

## 思考题

1. **问题**：为什么 LRU 比 FIFO 更好？它为何无法精确实现？
2. **答案**：LRU 淘汰"最久未使用"的页，利用时间局部性预测未来，近似 MIN；精确实现需为每页记录访问时间戳（硬件代价过高），因此用引用位做近似（时钟算法）。
3. **问题**：时钟算法中"清除引用位"的作用是什么？
4. **答案**：给"最近被引用过"的页第二次机会——若在下一轮扫描前它又被引用，引用位重新置 1 继续存活；清除引用位是把"过去"的引用记录重置，让算法只考察"最近一轮"的访问。
5. **问题**：内存只差 1% 为什么会导致 1000 倍变慢？
6. **答案**：每 100 次访问就有 1 次缺页（10ms 磁盘 I/O），平均访问时间从 100ns 恶化到约 100μs；且每次淘汰的往往是活跃页，缺页相互触发，形成换页风暴。

---

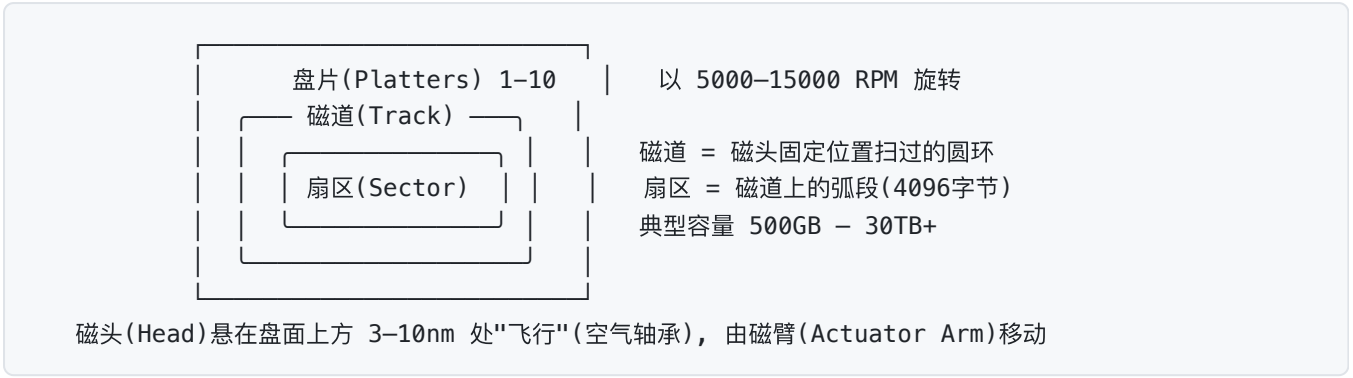
# Lecture 18: 磁盘（Magnetic Disks）

## 概述

文件系统建立在磁盘之上，而磁盘是"高延迟、顺序访问快"的设备。本讲介绍硬盘（HDD）的物理结构、一次读写的时间构成（寻道 + 旋转延迟 + 传输），以及操作系统与 I/O 设备的交互方式：**内存映射 I/O**、设备寄存器、轮询 vs 中断、**PIO vs DMA**，最后是现代的高效设备接口（命令队列/响应队列 + 门铃）。

## 核心概念与系统机制图解

### 磁盘物理结构




### 一次磁盘读写的时间构成

```
1. 寻道(Seek): 磁臂移动到目标磁道          典型 3–10 ms
2. 选择磁头
3. 旋转延迟(Rotational Latency): 等目标扇区转到磁头下   平均半圈 ≈ 4ms @7500RPM
4. 传输(Transfer): 扇区在磁头下掠过时读写              150–280 MB/s
──────────────────────────────────────────────────────────
   延迟(Latency) = 寻道 + 旋转延迟 ≈ 5–15 ms
```

- 磁盘被抽象为**线性块数组**：`read(startSector, count, addr)` / `write(...)`；磁道/扇区几何结构被设备隐藏（坏扇区自动重映射）。

### 与设备通信：内存映射 I/O（MMIO）

- CPU 把 I/O 设备的**设备寄存器**映射到物理地址空间：对特定地址做 load/store 就是读写设备寄存器（无缓存 load/store）。
- 设备寄存器位有三类用途：
- **参数**（CPU 写）：如起始扇区号；
- **状态**（CPU 读）：如"操作完成""出错"；
- **控制**（CPU 写）：如"开始磁盘读"。
- 注意：设备寄存器不像普通内存——"开始"位可能永远读为 0，状态位可能被设备自行改变。

### 设备完成信号：轮询 vs 中断

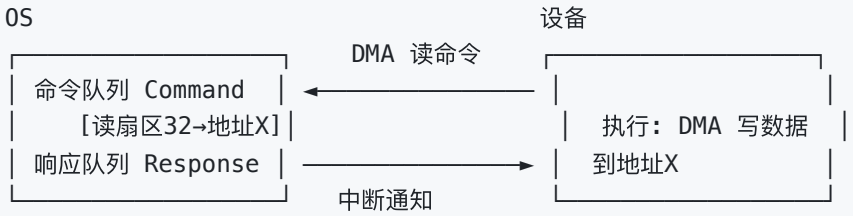
```
轮询(Polling):              中断(Interrupt):
OS 自旋读 ready 位          设备完成时强制 CPU 陷入(trap) OS
浪费 CPU 时间                 CPU 在设备工作时可做别的
```

- 中断处理流程：设备发中断 → CPU 跳到中断向量（与系统调用、缺页共用）→ OS 识别设备、服务（确认中断、启动新操作）→ 返回陷阱，恢复执行。
- 中断让 OS 高效：可同时让多个设备忙碌并运行用户代码；多核机器分散中断负载。

### 数据传输：PIO vs DMA

| 方式 | 机制 | 评价 |
| --- | --- | --- |
| PIO（Programmed I/O） | CPU 亲自 load/store 搬运数据 | 简单但浪费 CPU |
| DMA（Direct Memory Access） | 设备直接把数据搬到物理内存 | 现代主流：CPU 只写"缓冲地址"到设备寄存器 |

### 现代设备接口（DMA + 门铃）




OS 写"门铃(doorbell)"地址(uncached store) 通知设备有新命令
设备 DMA 读命令 → DMA 传数据 → DMA 写完成响应 → 发中断

## 代码示例与系统调用解说

### 示例：一次磁盘读取的系统调用视角（read）

```
#include <fcntl.h>
#include <unistd.h>
#include <cstdio>

int main() {
    int fd = open("/dev/sda", O_RDONLY);        // 打开块设备
    if (fd < 0) { perror("open"); return 1; }

    char buf[4096];                             // 一页/一个扇区组
    ssize_t n = pread(fd, buf, sizeof(buf),      // 从扇区偏移处读
                      0x100000);                // 字节偏移 = 扇区号 × 512
    if (n < 0) perror("pread");
    close(fd);
    return 0;
}
```

**【代码做了什么？】** 直接以字节偏移读取块设备，由内核把"偏移 → 扇区号"并驱动磁盘完成传输。

**【系统机制透视】** – 应用发出 `pread` 系统调用 → 陷入内核 → 文件系统/块层把字节偏移换算成扇区号 → 内核构建设备命令（写入命令队列 + 门铃）→ 设备 DMA 把数据搬到内核缓冲 → DMA 完成 → 设备中断 → 内核把数据拷贝到用户缓冲 → 系统调用返回。 – 这一路上：PIO 只用于极小的控制交互（门铃），数据搬运全靠 DMA；中断把"等待 I/O"的进程置为 BLOCKED，让 CPU 去跑别的线程——这与 Lecture 2–3 的线程状态机完美衔接。

## 关键要点

1. 磁盘延迟 = 寻道（3–10ms）+ 旋转延迟（~4ms）+ 传输（150–280MB/s）；顺序访问远比随机访问快。
2. 磁盘向软件暴露线性块数组；几何细节（内外磁道扇区数、坏块重映射）被设备隐藏。
3. 设备通过内存映射寄存器交互：参数/状态/控制三类位。
4. 轮询浪费 CPU，中断让 OS 高效并发处理多个设备；DMA 让设备直接搬数据，CPU 只发命令。
5. 现代接口 = 命令/响应队列 + 门铃 + DMA + 中断。

## 常见陷阱与注意事项

- **把磁盘访问想成"即时"**：一次随机读 5–15ms ≈ 数百万个 CPU 周期。
- **轮询密集设备**：大量设备都轮询会让 CPU 空转。
- **忘记设备寄存器与普通内存的区别**：读"开始位"永远 0、状态位会自发改变。
- **PIO 搬大块数据**：浪费 CPU；大传输必须 DMA。
- **忽视寻道代价**：随机小块 I/O 是磁盘性能杀手（调度算法见 Lecture 20）。

## 思考题

1. **问题**：为什么现代磁盘把"线性块数组"暴露给软件而不是磁道/扇区结构?
2. **答案**：隐藏物理几何（内外磁道扇区数不同、坏块重映射、预留空间）使软件接口简单稳定；设备内部自行管理几何细节，可随技术演进而不改变接口。
3. **问题**：DMA 相比 PIO 节省了什么?
4. **答案**：PIO 中 CPU 要逐字搬运数据（占用 CPU 时间）；DMA 中设备直接访问内存，CPU 只需设置缓冲地址与发起命令，可以转而执行其它线程——吞吐量与 CPU 利用率都大幅提升。
5. **问题**：为什么"等待 I/O 完成"时进程进入阻塞而非忙等?
6. **答案**：磁盘延迟毫秒级，忙等会让 CPU 空转数毫秒；阻塞后调度器把 CPU 交给其它就绪线程，I/O 完成时中断唤醒该进程——资源利用率与响应时间都更优。

---

# Lecture 19–21: 文件系统（File Systems）

# 概述

文件系统解决四个问题：**磁盘空间管理**（文件如何组织在磁盘上）、**命名**（从文件名找到数据块）、**可靠性**（崩溃后数据不丢失）、**保护**（用户间隔离与受控共享）。本讲从"文件是什么"出发，比较三种块组织方案（连续分配、链表、FAT、多级索引），再深入 BSD inode（12 直接 + 间接 + 双重间接）、块缓存与延迟写、空闲空间位图、块大小权衡与磁盘调度——Assign7 就是实现 Unix V6 的多级索引读取。

# 核心概念与系统机制图解

## 文件（File）

- **用户视角**：命名的字节集合，持久存储。
- **内核视角**：磁盘块的集合 + 元数据（属性）。
- **访问模式**：
- **顺序访问**：约 90% 的应用（编辑器、编译器）；逐字节处理。
- **随机访问**：按位置访问任意字节（数据库、请求调页的数据集）。
- **键控/索引访问**：按内容查找（数据库实现，通常不是 OS 提供）。
- **重要统计事实**：**大多数文件很小**（几 KB 内，每文件开销必须低），但**磁盘空间与大部分 I/O 由大文件占据**（大文件性能必须好）；文件可能不可预测地增长。

## inode（Index Node）

- **定义**：每文件一个的元数据结构，保存：文件大小、占用的扇区、访问时间（最后读/写）、保护信息（owner、group、rwx）。文件打开时 inode 驻留内存，平时存储在磁盘上。

## 块组织方案比较（Lecture 19—20）

### 1. 连续分配（Contiguous / Extents）（如 IBM OS/360）

```
inode: {起始扇区, 长度}
┌──┬──┬──┬──┬──┬──┬──┬──┐
│ 文件A(连续) │ 文件B(连续) │
└──┴──┴──┴──┴──┴──┴──┴──┘
优点: 简单、顺序/随机访问都快、顺序 I/O 最少寻道
缺点: 碎片化使大文件可能无法分配; 创建时必须预知大小; 无法扩展(过度分配)
```

### 2. 链接分配（Linked Files）（如 TOPS-10、Xerox Alto）

```
inode → [数据|→] → [数据|→] → [数据|⌀]
每块含指向下一块的指针
优点: 可扩展、无碎片、元数据小
缺点: 随机访问需追链(昂贵); 顺序访问也大量寻道
```

### 3. FAT（File Allocation Table，MS-DOS）

```
目录项: A: 6          FAT表(驻内存): 6→4→3→2→end
把"链接"集中到一张表: 每磁盘块一个表项, 存下一块号/结束标记/空闲标记
优点: 顺序访问快(块基本连续时), 随机访问快(表在内存), FAT 兼作空闲表, 块内无指针
缺点: 空闲空间易碎片化; FAT 必须常驻内存
历史: 16位FAT最多32MB; FAT32(1996) 32位+簇(2-32KB), 4KB簇支持1TB
现状: 闪存盘、相机等仍广泛使用
```

### 4. 多级索引（Multi-level Indexes，4.3BSD Unix / Unix V6）——Assign7 的主角

```
inode(14个块指针):
  0..11: 直接块(前12个数据块)          → 读块5: 直接查 inode[5]
  12:    间接块(1024个4字节指针)        → 读块23: 查 inode[12]→间接块→第11项
  13:    双重间接块(指向1024个间接块)    → 读块1040: inode[13]→双重间接→间接块→...
最大文件 ≈ 4GB(加三重间接可达4TB); 间接块按需分配
优点: 简单、无需预声明大小、小文件访问快(直接块)、比FAT省内存
缺点: 大文件随机访问需多读索引块(双重间接"二次故障"); 链表式空闲表局部性差
```

## 块缓存（Block Cache）与延迟写

- 用部分主存保留最近访问的磁盘块（LRU 置换）——inode、间接块等常用块命中缓存，解决大文件慢访问。
- **同步写（write-through）**：立即写盘——安全但慢。
- **延迟写（delayed writes）**：等约 30 秒再写——快、可合并多次小写、临时文件可能根本不用写盘；**风险**：崩溃丢失最近数据。

## 空闲空间管理

- 早期 Unix：空闲块链表——初始有序时局部性好，随后迅速打乱。
- **位图（free map/bitmap）**：每块一比特（1=空闲）；1TB 磁盘 ≈ 2^28 块 ≈ 32MB 位图；分配时找"靠近文件上一块"的空闲块（局部性）。
- **接近满盘**：位图扫描昂贵、局部性差 → 解法：**不让磁盘满**——把容量"虚报"少 10%（90% 满即拒绝写入）。

## 块大小权衡

- 512B 块：I/O 低效（寻道多）、间接块只能装 128 个指针（指针占 1% 空间）。
- 4KB 块：I/O 高效，但小文件内部碎片严重（可能浪费近一半空间）。
- **4.3BSD 折中**：4KB 大块 + 512B **碎片（fragment）**——只有文件最后一块可用碎片；多个文件的碎片可共享一个大块。

## 磁盘调度（Disk Scheduling）

目标：最小化寻道时间。

```
FIFO : 按到达顺序执行 —— 简单，不优化寻道(大量长距离移动)
SPTF : 每次选"定位时间最短"的请求 —— 最小化寻道，但可能饥饿
SCAN : 电梯算法—磁头单向移动，途中服务所有请求 —— 单向小寻道
CSCAN: 只朝一个方向服务请求，到端后快速返回 —— 公平且无回程服务
```

## 代码示例与系统调用解说

### 示例：文件读写与 fsync（Unix 文件 API）

```
#include <fcntl.h>
#include <unistd.h>
#include <cstdio>
#include <cstring>

int main() {
    int fd = open("notes.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd < 0) { perror("open"); return 1; }

    const char* msg = "file system lecture\n";
    ssize_t n = write(fd, msg, strlen(msg));      // 写入(可能只进块缓存!)
    if (n < 0) perror("write");

    fsync(fd);                                    // 强制把数据刷到磁盘
    close(fd);

    // 顺序读取
    fd = open("notes.txt", O_RDONLY);
    char buf[256];
    ssize_t r = read(fd, buf, sizeof(buf));
    printf("read %zd bytes: %.*s", r, (int)r, buf);
    close(fd);
    return 0;
}
```

**【代码做了什么？】** 创建文件、写入、`fsync` 刷盘、再顺序读回。

**【系统机制透视】** – `write` 通常只把数据放进**块缓存**并返回（延迟写）——崩溃可能丢数据；`fsync` 强制把脏块写盘（Lecture 23 讲崩溃恢复时会再见到它）。 – `open` 背后：内核按路径名逐级查目录（Lecture 22）→ 把 inode 读入内存 → 创建文件描述符（指向"打开文件描述"对象，含当前文件偏移）。 `read`/`write` 以文件偏移为基础做顺序访问；`pread`/`pwrite` 支持随机访问（数据库用）。 – Assign7 中你将**手写**从磁盘镜像读出 inode、追间接块、解析目录项的全过程——本讲所有机制都会落地为代码。

## 关键要点

1. 文件 = 命名字节集合 + 磁盘块集合 + inode 元数据；访问模式决定结构设计。

2. 连续分配（简单但有碎片、需预知大小）→ 链表（可扩展但随机访问差）→ FAT（表化链接，随机访问快）→ 多级索引（现代 Unix：直接+间接+双重间接）。
3. 块缓存 + 延迟写提升性能，但把"耐久性"推迟——`fsync` 用于关键数据。
4. 空闲空间位图管理 + "别让磁盘满"策略；4KB 块 + 512B 碎片兼顾 I/O 效率与碎片。
5. 磁盘调度（FIFO/SPTF/SCAN/CSCAN）用请求排序换寻道时间。

## 常见陷阱与注意事项

- **假定 write 已落盘**：延迟写 + 崩溃 = 丢数据；关键数据必须 fsync。
- **假定文件大小固定**：文件会增长，设计时考虑块按需分配（间接块）。
- **小文件也要低开销**：为所有文件统一分配大块会浪费空间（内部碎片）。
- **随机小 I/O 打爆磁盘**：顺序 vs 随机延迟差 3–4 个数量级。
- **空闲位图与 inode 不一致**：崩溃恢复问题（Lecture 23 专讲）。

## 思考题

1. **问题**：为什么多级索引（BSD inode）比 FAT 更省内存，同时又能高效处理小文件?
2. **答案**：FAT 表覆盖整个磁盘（1TB → 数十亿表项）必须常驻内存；多级索引只在文件**实际使用**时分配间接块，inode 只占固定一小块。小文件只用 12 个直接指针，一次访问即得数据。
3. **问题**：读取一个超过 12 块的文件的第 1040 块需要几次磁盘访问（缓存未命中时）？
4. **答案**：inode[13]（双重间接）→ 双重间接块 → 间接块 → 数据块，即 4 次访问（其中 3 次是索引块）。这正是"双重间接二次故障"问题，块缓存（常命中索引块）可缓解。
5. **问题**：延迟写为什么能显著减少磁盘 I/O?
6. **答案**：多次小写可合并成一次大写（同一块只写一次）、很快被删除的临时文件可能完全不用写盘；代价是崩溃时丢失最近未刷盘的数据——性能与耐久性的经典权衡。

---

# Lecture 22: 目录与链接（Directories and Links）

## 概述

上一讲解决"给定 inode 找数据块"，本讲解决"给定**名字**找 inode"：i-number 与 inode 数组、目录（directory）如何存储 `<名字, i-number>` 对、路径名查找（`open("/a/b/c")`）的逐级解析、工作目录，以及**硬链接（hard link）**与**符号链接（symbolic link）**的区别与使用场景。

# 核心概念与系统机制图解

## i-number 与 inode 数组

- **i-number**：inode 数组的下标，唯一标识一个 inode；OS 内部用 i-number 标识文件。
- inode 数组存储在磁盘上：早期 Unix 放在磁盘**起始处**；后来移到**中部**（缩短寻道）；BSD FS 分散为**多块分布全盘**（inode 靠近数据块）。

## 目录（Directory）

- **定义**：把"名字 → i-number"映射起来的特殊结构。现代系统用**层级目录树**。
- **Unix/Linux 目录**：目录本身就是一个文件（inode 的 type == directory），内容是无序的 `<name, i-number>` 对列表；只有 OS 能写。根目录 `/` 的 i-number 固定为 1。

```
assign0 目录内容(例子):
  movietest.cc  96      questions.txt  656
  samples       228     movie.cc       780
  movie.h       1441    buggy.c        6312
  Makefile      1443    ...
```

## 路径名查找：open("/a/b/c")

```
1. 读 inode 1 (根目录 "/")
2. 在根目录块中找 "a" → i-number 17
3. 读 inode 17 (目录 a)
4. 在 a 的块中找 "b" → i-number 23
5. 读 inode 23 (目录 b)
6. 在 b 的块中找 "c" → i-number 42
7. 读 inode 42 (文件 c), open 完成
每次目录查找 = 读目录 inode + 读其数据块
```

- **工作目录（working directory）**：OS 为每进程保存一个"当前目录"；不以 `/` 开头的路径从工作目录开始查找；`pwd` 打印它。

## 硬链接（Hard Link）

- **定义**：目录项就是"链接"——一个 `<name, i-number>` 对。多个目录项可以指向**同一个 i-number** → 多个名字引用同一文件。
- **引用计数**：inode 中记录链接数（nlink）；`rm` 只是删除一个目录项（减少链接数）；计数归零才真正释放文件数据。
- **你天天见的硬链接**：`rm`（删链接）、每个目录里的 `.`（指向自身）与 `..`（指向父目录）。

```
ln /a/b/c  t        → 在 /g 下创建 t: {t, 42}
目录b: {c, 42}      目录g: {t, 42}       inode 42: nlink=2
```

- **硬链接的限制**：
- 不能链接到**目录**（防止目录图出现环）；
- 不能跨**文件系统**（i-number 只在同一文件系统内有意义）。
- → 因此 BSD Unix 增加了**符号链接**。

### 符号链接（Symbolic Link / Symlink）

- **定义**：一种特殊文件（inode type == symbolic link），**内容就是另一个路径名**。
- **查找规则**：解析路径时遇到符号链接，把其内容**拼接到剩余路径前**继续查找；若内容以 `/` 开头则从根目录重新开始。

```
cd /a;  ln -s e/f b;  cat /a/b/c
→ 查找 /a 发现 b 是符号链接 "e/f"
→ 继续查找 e/f/c → 等价于 cat /a/e/f/c
```

- **优点**：可跨文件系统、可链接目录。
- **缺点**：可形成**环**（无限循环）；容易产生**悬空链接**（指向不存在的文件）。

## 代码示例与系统调用解说

### 示例：硬链接与符号链接（shell + C）

```
echo "hello" > f1
ln f1 f2                # 硬链接: f1 与 f2 指向同一 inode
ln -s f1 f3             # 符号链接: f3 的内容是 "f1"
ls -li                  # -i 显示 i-number: f1/f2 相同, f3 不同
rm f1                   # 删除一个硬链接; f2 仍可读("hello" 未丢)
cat f2                  # → hello
cat f3                  # → hello (符号链接仍解析到 f2? 不: f3→"f1" 已不存在 → 悬空!)
ls -l f3                # f3 -> f1 (悬空符号链接)
```

```
#include <cstdio>
#include <unistd.h>
#include <sys/stat.h>

int main() {
    link("f1", "f2");              // 创建硬链接（相当于 ln f1 f2）
    symlink("f1", "f3");           // 创建符号链接（相当于 ln -s f1 f3）

    struct stat st1, st2;
    stat("f1", &st1);              // 跟随符号链接的 stat
    stat("f2", &st2);
    printf("same inode: %s\n", st1.st_ino == st2.st_ino ? "yes" : "no");

    struct stat st3;
    lstat("f3", &st3);             // lstat 不跟随符号链接：看到的是链接本身
    printf("f3 is symlink: %s\n", S_ISLNK(st3.st_mode) ? "yes" : "no");
    return 0;
}
```

**【代码做了什么？】** 用 `link`/`symlink` 系统调用创建两类链接，用 `stat`（跟随链接）与 `lstat`（不跟随）区分它们。

**【系统机制透视】** – 硬链接不复制数据——两个名字共享 inode 与数据块，删除一个名字只是 nlink--；文件"真正删除"发生在 nlink 归零且没有进程打开它时。 – 符号链接是一个独立的小文件（存路径字符串），解析由**路径名查找**过程完成；删除目标文件后，符号链接变成悬空链接（读取报 ENOENT）——它不参与 inode 计数。 – 目录项的 `nlink` 就是 Lecture 10 讲的**引用计数**在文件系统中的应用；fsck 会检查目录链接数是否与目录项数一致（Lecture 23）。

## 关键要点

1. i-number 唯一标识文件；inode 数组存于磁盘（位置从起始→中部→分散演化）。
2. 目录 = `<名字, i-number>` 对的文件；`open("/a/b/c")` 沿路径逐级查目录。
3. 硬链接 = 共享 inode 的多个目录项；`rm` 删链接、计数归零才释放文件。
4. 硬链接不能指向目录、不能跨文件系统；符号链接是"存路径的文件"，可跨 FS、可指向目录，但会悬空/成环。
5. `stat` vs `lstat`：是否跟随符号链接。

## 常见陷阱与注意事项

- **把硬链接当拷贝**：修改任一名字下的内容，另一名字可见（同一文件）。
- **创建指向目录的硬链接**：系统禁止（`ln dir` 报错）。
- **符号链接悬空**：目标被删除后链接仍在但不可用。

- **符号链接环**：`ln -s b a; ln -s a b` 之类会造成查找无限循环（OS 有循环上限检测）。
- `rm` **后以为数据没了**：若还有其它硬链接，数据仍在。

## 思考题

1. **问题**：为什么硬链接不能指向目录?
2. **答案**：目录是树的节点；若允许目录硬链接，目录图可能出现环（如目录 a 的子项指向祖先目录），路径查找将无限循环，且"删除子树"的语义变得复杂。`.` 与 `..` 是仅有的例外（不增加可导航环）。
3. **问题**：符号链接与硬链接在"删除原文件"后的行为有何不同?
4. **答案**：硬链接下删除一个名字，其它名字仍有效（数据与 inode 因计数未归零而保留）；符号链接下删除目标文件，链接变成悬空链接——因为它只是存了一个不再存在的路径字符串。
5. **问题**：`open("/a/b/c")` 最坏情况需要多少次磁盘访问（无缓存）？
6. **答案**：每个路径分量至少一次"读目录 inode + 读目录数据块"，即 3 个目录 × 2 ≈ 6 次以上访问；块缓存命中目录块时可大幅减少（这正是块缓存存在的理由之一）。

---

# Lecture 23–24: 文件系统崩溃恢复（File System Crash Recovery）

## 概述

崩溃恢复对操作系统其余部分很简单（重启即清零），但对文件系统是生死攸关的：用户期望磁盘上的信息在崩溃后依然存在。本讲梳理三大方案：**崩溃后修复（fsck）**、**有序写入（ordered writes）**、**预写日志（write-ahead logging / journaling）**——最后者是现代 ext4/NTFS 与 Assign8 的核心。

## 核心概念与系统机制图解

### 崩溃恢复为什么难

- **数据丢失**：延迟写意味着最近约 30 秒的修改可能还在内存（块缓存）里没上盘。
- **不一致（Inconsistency）**：一次修改往往涉及**多个块**（如加块到文件 = 更新空闲位图 + 更新 inode），磁盘无法原子地完成多块写；崩溃可能落在中间：
- 空闲位图已更新但 inode 还没指向新块 → **块泄漏**（块既不在任何文件里也不在空闲表中）；
- 新目录项已写但 inode 链接数未更新 → 计数与目录不一致。
- **块缓存可能重排写入顺序**。

### 方案 1：崩溃后修复——fsck（file system check）

```
启动时运行 fsck:
  1. 检查"干净位(clean bit)": 上次正常关机 → 跳过
  2. 否则全盘扫描元数据: inode、间接块、空闲位图、目录
  3. 找出不一致并修复
典型不一致与修复:
  ● 块既在 inode 又在空闲位图 → 从空闲位图移除(防双重分配)
  ● inode 链接数与目录项不符 → 修正计数
  ● 同一块属于两个 inode(先删A后建B, 延迟写乱序) → 随机选一个归属/复制块/两边都删
  ● inode 计数>0 但不在任何目录 → 放入 /lost+found 目录
```

- **fsck 的局限**：
- 恢复一致性但**不保证不丢信息**（恢复后系统可能仍不可用，如高层目录损坏）；
- 安全问题：块可能在崩溃中从密码文件迁移到别处；
- **太慢**：现代大磁盘不可接受——5TB 全盘顺序读约 8 小时，随机读 10% 需数周。

### 方案 2：有序写入（Ordered Writes）

- **思想**：规定写盘顺序，把"不一致"换成"较轻的泄漏"。
- **例子：给文件加一块**，按此顺序写：

```
i.  先写空闲位图块(标记新块已分配)
ii. 再写 inode(指向新块)
崩溃分析:
  ● 只写完 i → 位图说空闲、inode 没指 → 无问题
  ● 写完 i 和 ii → 一致
  ● 写完 i 没写 ii → 块"泄漏"(既不在文件也不空闲) —— 比不一致轻, 可后台 fsck 回收
结论: 绝不出现"同一块同时在空闲表和 inode 中"(双重分配)
```

- **通用原则**：指针指向的数据要先初始化好；重用资源前先清掉旧指针。
- **代价**：简单的有序写 = 同步写（write-through），拖慢文件操作；改进版在块缓存中记录**依赖关系**（写 inode 前先写位图块），避免同步写，但依赖环需强制写打破——实现微妙。

### 方案 3：预写日志（Write-Ahead Logging / Journaling）

- **思想**（数据库界老方法）：**先**把"这次操作要改什么"追加写进一个日志文件，**再**执行实际的块更新（顺序随意）。

```
操作: 给 inode 862 加块 99421 (索引93)
步骤: 1. 写日志: "加块99421到inode862第93块"  (append-only, 顺序写无寻道)
      2. 执行实际块更新(任意顺序)
      3. 事务完成后可截断日志
崩溃恢复: 重放日志, 完成所有未完成更新
保证: 一旦操作开始, 最终必然完成
```

- **日志条目的两种形式**：
- 逻辑操作：`把块 99421 加到 inode 862 的第 93 项`；
- 物理补丁：`把块 6159972 偏移 324 处的 4 字节改为 9942`；
- 条目必须**幂等（idempotent）**——重放多次结果相同（恢复过程本身也可能崩溃）。
- **一致性组（事务，transaction）**：一次逻辑操作可能含多个日志条目；要么全部生效、要么全不生效。用"事务开始/结束"标记日志分组（Assign8 的做法）；只处理完整的事务。
- **检查点（checkpoint）与日志截断**：日志无限增长恢复会变慢；定期记录"日志头位置 + 冲刷所有脏块"，之后截断日志。
- **记多少？** 通常只记录**元数据**（空闲位图、inode、间接块）；记录全部文件数据太贵。

### 日志方案评价

| 优点 | 缺点 |
| --- | --- |
| 恢复快（只重放日志） | 每次元数据操作前要同步写日志 |
| 消除不一致 | 延迟写仍可能丢失最近数据（需 fsync） |
| 日志顺序写、无寻道 | 磁盘自身故障仍需复制/备份 |
| 元数据可用延迟写（有日志兜底） | — |

- **延迟日志写**：日志条目只需在"相关块写盘之前"落盘（不需要同步写）——把**耐久性（durability）与一致性（consistency）分离**。
- **结论（Lecture 23）**：性能、耐久、一致性三者不可兼得——必须决定想从哪些故障中恢复。

## 代码示例与系统调用解说

### 示例：重放日志恢复一致性（Assign8 的核心，约 10–15 行）

```
// 伪代码: 崩溃后重放 write-ahead log
// log 中的每条记录是 "物理补丁" 或 "分配/释放标记", 按事务分组
void recover(std::vector<LogEntry>& log) {
    for (const auto& txn : groupByTransaction(log)) {
        if (!txn.complete) continue;        // 不完整的组(崩溃中断)直接跳过
        for (const auto& entry : txn.entries) {
            switch (entry.kind) {
            case PATCH:                     // 覆盖磁盘块中若干字节
                writeBlock(entry.block, entry.offset, entry.data);
                break;
            case MARK_FREE:                 // 把块标为空闲(更新位图)
                freemap.set(entry.block, FREE);
                break;
            case MARK_ALLOCATED:
                freemap.set(entry.block, ALLOCATED);
                break;
            }
        }
    }
}
```

**【代码做了什么？】** 按事务分组扫描日志：只重放**完整**事务（崩溃发生在事务中间则整组丢弃，保证原子性），对每条记录执行补丁或位图更新。

**【系统机制透视】** – **幂等性**：重放可能被再次崩溃打断、再次重放——补丁必须"重复应用结果相同"（覆盖写天然幂等）。 – **日志先于数据**：若先改数据块再写日志，崩溃后日志里没有记录可重放——所以必须"写前日志"（write-ahead）。 – **与 Assign8 的衔接**：作业里你用 FUSE 挂载 V6 文件系统，给所有元数据更新加日志，崩溃后用上面的逻辑恢复——你会亲身体会"10–15 行代码背后是整套一致性理论"。

## 关键要点

1. 崩溃恢复难在：延迟写丢数据 + 多块更新不一致 + 缓存重排写序。
2. fsck：全盘扫描修复，保证一致性但慢、可能丢信息——大磁盘不可接受。
3. 有序写入：用写序把"不一致"换成"泄漏"，但同步写拖慢性能。
4. **预写日志**：先记日志后改数据、事务原子性、幂等条目、检查点截断——现代文件系统（ext4/NTFS）的标准答案。
5. 性能 / 耐久 / 一致性三者不可兼得，必须明确故障模型。

## 常见陷阱与注意事项

- **日志写与数据写顺序颠倒**：必须先日志后数据，否则崩溃无法恢复。
- **日志条目非幂等**：重放两次产生不同结果（如"分配计数++"）。
- **重放不完整事务**：必须整组跳过，否则破坏事务原子性。

- **忘记 fsync 关键数据**：延迟写 + 崩溃 = 数据丢失。
- **日志无限增长**：不设检查点，恢复时间越来越长。

## 思考题

1. **问题**：为什么"块既在 inode 里又在空闲位图里"比"块泄漏"更严重?
2. **答案**：前者是双重分配——两个文件可能共享同一块，写入会互相覆盖、数据损坏且难修复；后者只是少了一块可用空间（泄漏），数据完好、可通过后台回收。
3. **问题**：预写日志如何保证"操作要么全部生效要么全不生效"?
4. **答案**：用事务标记把相关日志条目组成一致性组；恢复时只重放完整的事务，不完整事务（崩溃中断）整组丢弃。配合"先写日志再改数据"，就能保证已提交操作最终完成。
5. **问题**：为什么日志条目必须幂等?
6. **答案**：恢复过程本身可能再次崩溃，日志可能被重放多次；只有幂等条目（重复应用结果不变）才能保证多次重放与一次重放效果一致。

---

# Lecture 25: 真相、信任与技术（Truth, Trust, and Technology）

## 概述

本讲把 Lecture 12 的信任框架应用到现代社会的信息生态：为什么不同群体对同一事实有截然相反的信念?技术（社交媒体算法、生成式 AI、深度伪造）如何放大**过度信任**与**不值得信任**?我们学习确认偏误、算法注意力经济、AI 幻觉与"认知信任的崩塌"，并讨论"如何选择值得信任的信息来源"。

## 核心概念与系统机制图解

### 大规模错位信任

- 公式：**(我所相信的) » (我所感知的)**——个人没有资源独立验证一切，必须选择信任他人的结论。
- 不同群体信任不同的信息来源 → 其中必有一些是不值得信任的。
- **为什么大规模过度信任?**
- 判断谁值得信任很难；
- 有缺陷的推断技术：
  - **确认偏误（Confirmation Bias）**："我信任它因为它印证了我的信念"；

- **虚假的数字信任（False trust in numbers）**："那么多人说，肯定是真的"。

### 案例 1：社交媒体算法

- **机制**：注意力 = 金钱 → 强化偏见与恐惧能增加注意力（用户对相反观点没兴趣）→ 用户看到大量印证自己信念的内容 → 平台从你的确认偏误中获利。
- **伤害**：选举期间党派内容失衡；法院认定算法导致焦虑、抑郁、躯体畸形焦虑与自杀意念。
- **要点**：点赞/互动/转发 ≠ 真相；"算法推给我" ≠ "编辑选择"；不是所有注意力都值得优化。

### 案例 2：生成式 AI（ChatGPT、Claude 等）

- **为什么人们信任它**：权威语气 + 详细解释 + 大量具体"事实" + 即使错了也自信满满。
- **现实**：AI 会**幻觉（hallucinate）**，尤其在不预测、无预警地出错时，对事实准确性不可靠；事实核查责任落在用户身上；AI 藏在应用里让人忘记信息来源；**一次幻觉能扩散成"多个独立来源"的假象**。
- **要点**：不经验证不要信任生成式 AI 的"真相"；把输出当"待验证的假设"；所有结果必须独立验证（**替代 substitution**）。

### 案例 3：深度伪造与合成媒体

- 历史上，照片/视频/音频很难伪造，人们有理由推断信任它们；新技术让令人信服的伪造成为可能。
- 危害不仅是"假被当真"，更是**整个认知信任的崩塌**：人们不再相信任何证据能可靠地告诉我们真相（"一个不再相信任何东西的民族无法自己做决定"——Hannah Arendt）。
- **机会**：更好的内容标记、有效的检测、支持可信机构、保护个人免受语音/形象克隆。

### 结论

- 信任是社会分裂的核心；决定信谁越来越难。
- 过度信任与不值得信任都是大问题；确认偏误极难避免。
- 技术放大两者：不值得信任的来源显得更可信；我们过度信任熟悉或方便的来源。
- 最佳希望：**有长期可信记录的制度化机构**——但人们会信任它们吗？

## 代码示例与系统调用解说

### 示例：用"替代"原则给 AI 输出加验证（思想实验的代码形态）

```
// 演示"用替代(substitution)验证 AI 输出"的工程模式
// 真实系统里: 交叉验证、引用溯源、事实库核对、人工复核
#include <string>
#include <iostream>

struct AIOutput {
    std::string claim;
    std::vector<std::string> sources;   // 可核实的来源
};

bool verifyAgainstPrimarySource(const AIOutput& out) {
    // 用权威来源/原文交叉核对关键论断(替代: 备用可信系统兜底)
    return crossCheck(out.claim, out.sources);
}

int main() {
    AIOutput answer = askLLM("CS111 midterm 时间?");
    if (!verifyAgainstPrimarySource(answer)) {
        std::cout << "未验证, 拒绝采用\n";   // 不可验证 = 不可信
    }
}
```

**【代码做了什么？】** 把 AI 输出当作"假设"，强制用可核实的来源交叉验证后才采纳。

**【系统机制透视】** – 这正是 Lecture 12 的**替代（substitution）**机制：不把信任全部押在单一系统上，而是引入"后备验证系统"；后备系统自身也必须是可信的（权威来源、可审计）。 – 从 OS 的角度看，这与"日志 + 一致性检查 + 冗余"建立软件信任是同一模式——只是应用领域从系统正确性扩展到信息真实性。

## 关键要点

1. 大规模错位信任源于"个人无法独立验证 + 有缺陷的推断（确认偏误、数字崇拜）"。
2. 社交媒体算法把注意力变现，系统性放大确认偏误——互动量 ≠ 真相。
3. 生成式 AI 以权威姿态输出幻觉，且"一次幻觉可伪装成多个来源"——必须独立验证。
4. 深度伪造威胁的不是单条证据，而是整个认知信任体系（Hannah Arendt 之问）。
5. 对抗策略：以制度化可信机构为锚、内容标记、检测技术、支持独立核实。

## 常见陷阱与注意事项

- **把流畅度当正确性**：AI 说得越自信、越详细，越需要警惕。
- **把"很多人转发"当证据**：虚假信息可以批量复制传播。
- **只订阅强化自己信念的信息源**：确认偏误的自我强化循环。
- **对"看起来正常"的系统停止审视**：过度信任的系统性来源。

## 思考题

1. **问题**：为什么说"算法向你展示了它"不等于"编辑选择了它"?
2. **答案**：算法的目标是最大化注意力（留存与广告收入），会优先推送最强化你既有信念的内容；编辑选择承担了事实把关责任，而算法不做真伪判断——两者在"可信度"上完全不同。
3. **问题**：用信任的三种建立方式分析"如何对待 AI 生成的事实性回答"。
4. **答案**：假设（直接信）不可接受；推断（它过去答对过）太弱；正确做法是**替代**——用独立权威来源交叉验证，把 AI 输出当作待验证假设，验证链本身要可信。

---

# Lecture 26: 闪存（Flash Memory）

## 概述

闪存（SSD）已取代磁盘成为主流存储。本讲从闪存单元的物理特性（写不对称、擦除单位、磨损）出发，解释为什么闪存不能直接当磁盘用，进而介绍**闪存转换层（FTL）**：块映射、页头（A/W/G 位）、垃圾回收、**写放大（write amplification）**与**磨损均衡（wear leveling）**，以及文件系统与 FTL 的协作（TRIM）。

## 核心概念与系统机制图解

### 闪存单元特性

对比:
  与磁盘比: 无活动部件(可靠、抗冲击); 随机访问延迟低100–1000倍; 每比特成本高3–10倍
  与DRAM比: 非易失(断电保留); 每比特成本低5–20倍; 慢100–1000倍
单元行为:
- 读: 快(~10–100微秒), 以页(page)为单位访问(4–16KB, 更像磁盘)
- 写(1→0): 较快(~100–1000微秒)
- 写(0→1): 很慢(~1000–10000微秒)!
- 必须先"擦除(erase)"(写成全1)才能重写
- 擦除以"擦除单元(erase unit)"为单位(1–8MB, 含很多页), 约2ms
- 擦除会磨损: 每个擦除单元只能擦除约100–100,000次 → 磨损(wear out)

### 为什么闪存不能直接当磁盘用 → FTL

- 现有文件系统按"可原地覆写块"设计，与闪存"擦除–重写"模型冲突。
- **闪存转换层（Flash Translation Layer, FTL）**：SSD 内的软件，向文件系统暴露**线性块数组**（虚拟块号），内部把虚拟块映射到物理页。代价：性能折损、空间浪费、实现专有。

### 直接映射 FTL（简单但差）

```
块N → 物理页N; 写块N = 读整个擦除单元 → 擦除 → 重写整个单元
问题: 每次写加2ms擦除; 反复写同一块快速磨损; 崩溃丢旧数据
(只用于廉价U盘+FAT)
```

### 块映射 FTL（现代做法）

```
维护"块映射": 虚拟块 → 物理页
读: 查映射表取物理位置
写: 找空闲已擦除页 → 写入 → 更新映射 → 旧页标记为垃圾
每页头部记录元数据:
  A(Allocated)位: 1=空闲 0=已分配
  W(Written)位:  1=未写(全1) 0=已写
  G(Garbage)位:  0=不再使用, 复用前必须擦除
状态机: 111(刚擦除) → 011(已分配未写) → 001(已写成功) → 000(已删除/垃圾)
  (011 状态用于检测写块过程中崩溃)
崩溃恢复: 扫描所有页头, 重建映射表与空闲表
```

### 垃圾回收与写放大

```
问题: 垃圾页堆积 → 容量缩水
垃圾回收: 找垃圾多的擦除单元 → 把活页复制到干净单元(更新映射) → 擦除旧单元
写放大: 每写入 1 个新数据单元, 实际写入 1/(1-U) 个单元(U=平均利用率)
  U=0.5 → 写放大2;  U=0.9 → 10;  U=0.99 → 100
改善: 让利用率呈"双峰分布"——大多数单元接近满(U≈1)、少数接近空(U≈0);
      按"温度"分仓: 热块(频繁覆写)与冷块分开存放; 垃圾回收专挑热而空的单元
```

### 磨损均衡（Wear Leveling）

- 热擦除单元磨损快 → 偶尔对**冷单元**做垃圾回收（收不回多少空间，但得到一个"新"单元）→ 把擦除分散到整个设备。

### FTL 与文件系统协作

- **重复映射**：文件块 → 逻辑磁盘块 → 闪存页，两层翻译浪费；早期"闪存优化文件系统"因便宜 FTL 的普及而失败。
- **信息缺失**：FTL 不知道 OS 何时释放块（只有覆写时才知道）→ 垃圾回收常搬运已删除文件的数据块。
- **TRIM 命令**：文件系统主动告知闪存"这些块已释放" → FTL 提前标记为垃圾。
- **趋势**：当前文件系统与 FTL 共存，由 FTL 负责磨损均衡、坏块、擦除调度；闪存鼓励"不在原地更新（out-of-place update）"的文件系统设计。

## 代码示例与系统调用解说

### 示例：TRIM / discard 的用户可见行为（Linux）

```
# 创建文件 → 写入 → 删除：让 SSD 知道块已释放
echo "data" > f
sync
rm f

# 手动 TRIM（fstrim 对所有挂载 FS 发送 discard）
sudo fstrim -v /
# 输出：/: 12.3 GiB (13201264640 bytes) trimmed
```

**【代码做了什么？】** `rm` 删除文件后，`fstrim` 向 SSD 发送 TRIM 命令，告知哪些块不再需要保留。

**【系统机制透视】** – 没有 TRIM 时，FTL 只有等块被**覆写**才知道旧副本是垃圾；删除的文件块会被垃圾回收白白搬运（浪费写入寿命与性能）。 – TRIM 是文件系统 ↔ FTL 的信息通道：文件系统把"逻辑释放"翻译成"物理可回收"，让写放大与磨损更可控——这是"分层系统间信息泄露/传递"的典型例子（Lecture 28 的"分层"主题）。

## 关键要点

1. 闪存"1→0 快、0→1 需擦除、擦除以大单元为单位、擦除磨损"——与磁盘模型根本不同。
2. FTL 把闪存伪装成线性块磁盘；核心是块映射 + 页头位（A/W/G）+ 崩溃重建。
3. 垃圾回收造成**写放大 1/(1–U)**；双峰利用率分布可显著改善。
4. **磨损均衡**把擦除分散到全设备，延长寿命。
5. TRIM 让文件系统告知 FTL 释放的块，减少无效搬运。

## 常见陷阱与注意事项

- **把闪存当磁盘原地覆写**：频繁小写会触发整单元擦除重写（写放大）。
- **忽略磨损**：热数据集中导致部分单元快速报废。
- **不做崩溃处理**：写页中途崩溃（A/W/G 状态机检测）需要重建机制。
- **忘记 TRIM**：删除的数据仍被 FTL 当作有效数据搬运。
- **用磁盘思维优化 SSD 布局**：寻道优化（SCAN 等）对 SSD 无意义；应优化写放大与磨损。

## 思考题

1. **问题**：为什么闪存写放大公式是 1/(1–U)？利用率 90% 意味着什么？

2. **答案**：垃圾回收每腾出 1–U 的空间就要重写 U 的旧数据；每写入 1/(1–U) 个单元才能释放 1 个单元的空间。U=0.9 意味着每次有效写入实际消耗 10 倍的写带宽——既慢又加速磨损。
3. **问题**：TRIM 命令解决了 FTL 的什么问题?
4. **答案**：FTL 无法区分"已删除的块"与"有效数据"（只能等覆写）；TRIM 让文件系统在删除时主动告知，FTL 可立即把这些页标记为垃圾，避免垃圾回收搬运无用数据，降低写放大。

---

# Lecture 27: 虚拟机（Virtual Machines）

## 概述

进程抽象只是底层机器的一个**子集**（部分指令、虚拟内存、系统调用）；如果让"进程"拥有**完整的机器**（全部指令与寄存器、物理内存视图、MMU、I/O 设备、陷阱与中断），它就是一台**虚拟机（VM）**。本讲讨论虚拟机监视器（hypervisor）的实现：模拟 vs **直接执行 + 陷阱模拟**、虚拟 I/O 与虚拟内存（影子页表/嵌套页表），以及虚拟机如何成就现代数据中心与云计算。

## 核心概念与系统机制图解

### 进程 vs 虚拟机：抽象层级的差异

进程抽象(OS 提供):
- 内存: 虚拟内存页的线性数组
- CPU: 所有非特权指令+寄存器
- 系统调用: 文件/进程操作等
- 是底层机器的子集

虚拟机抽象(hypervisor 提供):
- CPU: 全部指令+寄存器(含特权指令)
- 内存: 物理内存页 + MMU(页表等)
- I/O 设备: 定时器/磁盘/网卡/显示
- 陷阱与中断: 系统调用只是普通陷阱
  - 是一台"完整私有机器"

- 运行在 VM 里的 OS 叫**客户操作系统（guest OS）**；管理 VM 的 OS 叫 **hypervisor（虚拟机监视器，VMM）**；一台物理机可同时运行多个不同 guest OS 的 VM。

### 实现方式 1：完全模拟（Simulation）

- 用程序模拟 CPU 指令、MMU 与物理内存（一个大数组）、I/O 设备（磁盘用镜像文件）。
- 太慢：CPU/内存慢 100 倍，I/O 慢 2 倍。

### 实现方式 2：直接执行 + 陷阱模拟（Direct Execution）——主流

```
把 guest OS 放在"用户模式"运行:
  ● 绝大多数指令以全速直接执行
  ● 特权指令(如 CLI/STI/POPF/HALT)执行时产生非法指令陷阱 → hypervisor 模拟
  ● 特权指令在内核代码中相对罕见 → 模拟开销占比小
```

**CLI（关中断）的模拟示例**：

```
1. guest OS 在用户模式执行 CLI → 非法指令陷阱
2. hypervisor 的陷阱处理器运行(IDT 指向 hypervisor)
3. hypervisor 检查到是 CLI → 模拟: 把"本 VM 的中断屏蔽"标记置位
4. 返回陷阱: CPU 回到用户模式, guest OS 从 CLI 下一条继续, 中断已被屏蔽
```

**guest 内的系统调用**（应用 syscall → guest OS sysret）：

```
应用执行 syscall → 陷阱进 hypervisor(机器IDT) → hypervisor 模拟 syscall
  → 设置 vCPU 为内核模式 → 返回用户模式 → guest OS 执行系统调用
  → guest OS 执行 sysret → 再次陷阱进 hypervisor → hypervisor 模拟 sysret
  → vCPU 回用户模式 → 应用继续
```

## 虚拟 I/O 设备

- guest OS 读写"虚拟设备寄存器"时，hypervisor 让这些访问触发陷阱，由陷阱处理器模拟设备功能；完成后在虚拟 CPU 上模拟一个中断。
- 减少陷阱数：为 guest 写**新的设备驱动**，用 hypervisor 调用（系统调用）代替设备寄存器访问 → **半虚拟化（paravirtualization）**。

## 虚拟化虚拟内存：影子页表与嵌套页表

```
guest 虚拟AS → guest "物理"AS → host "机器"AS
影子页表(Shadow Page Maps):
  hypervisor 维护"guest虚拟→机器物理"的合成页表, 装入真实MMU
  guest 每次改页表都要 hypervisor 拦截更新影子表 → 开销高
x86-64 硬件扩展(Intel EPT / AMD NPT):
  增加一层页表: VM内页表(VPN→PPN) + hypervisor页表(PPN→MPN)
  硬件自动完成两级翻译, 无需影子表 → 大幅降低开销
```

## 虚拟机的价值与历史

```
封装性: VM 封装全部执行状态 → 可复制、保存、迁移
数据中心整合: 一台机器跑多个"单应用 VM"(隔离+利用率) → 云计算的基石
历史: 1960s IBM 发明(一台机器多用户); 80-90年代沉寂(人人有PC);
      1990s 中期因 Windows 垄断地位而复兴(兼容性测试); 2000s 数据中心整合 → 云
```

## 代码示例与系统调用解说

### 示例：用 Linux KVM/QEMU 体验虚拟机（命令行形态）

```
# 1. 创建磁盘镜像(模拟一块"物理磁盘")
qemu-img create -f qcow2 myvm.qcow2 8G
# 2. 启动一台虚拟机(客户 OS 的"裸机")
qemu-system-x86_64 \
    -enable-kvm \              # 使用硬件虚拟化扩展(嵌套页表)
    -m 2048 \                  # 给 VM 2GB "物理内存"
    -drive file=myvm.qcow2,format=qcow2 \
    -cdrom ubuntu.iso          # 安装 guest OS
```

**【代码做了什么？】** 创建磁盘镜像并启动一个完整 x86 机器（CPU、内存、磁盘、BIOS）运行客户操作系统。

**【系统机制透视】** – `-enable-kvm` 开启 Intel VT-x/AMD-V：CPU 硬件直接支持"guest 模式"，特权指令自动陷入 hypervisor（VMM），配合 EPT 嵌套页表实现两级地址翻译——这正是 Lecture 27 讲的"直接执行 + 硬件辅助"。 – 磁盘镜像是"用文件模拟物理磁盘"：文件系统（qcow2 格式）在宿主机上存储 VM 的块设备——最外层的文件系统"虚拟化"了里层的文件系统。

## 关键要点

1. 进程抽象是机器的子集；VM 让"进程"拥有完整机器（CPU/内存/MMU/I/O/陷阱）。
2. hypervisor 用"直接执行 + 特权指令陷阱模拟"达到接近原生的性能。
3. 虚拟 I/O 与半虚拟化减少陷阱开销；影子页表被硬件嵌套页表（EPT/NPT）取代。
4. VM 的封装性（复制/迁移/隔离）催生数据中心整合与云计算。
5. 虚拟机 = 操作系统级虚拟化的"终极形态"，与线程（CPU 虚拟化）、文件（存储虚拟化）一脉相承。

## 常见陷阱与注意事项

- **把模拟与虚拟化混为一谈**：完全模拟慢 100 倍；直接执行才是主流。
- **忘记特权指令陷阱**：没有陷阱机制，guest 能执行特权指令破坏宿主机。
- **影子页表开销**：guest 频繁改页表时维护开销大（硬件嵌套页表解决）。
- **忽略 I/O 虚拟化开销**：设备模拟是 VM 性能瓶颈之一（半虚拟化/直通缓解）。

## 思考题

1. **问题**：为什么"直接执行"能让 guest OS 以近全速运行?

2. **答案**：guest 的大多数指令（算术、访存、控制流）在用户模式下直接执行，无需模拟；只有特权指令（相对稀少）才陷阱进 hypervisor 模拟——模拟占比小，总体接近原生速度。
3. **问题**：影子页表解决了什么问题？为什么后来被嵌套页表取代？
4. **答案**：影子页表把"guest 虚拟→机器物理"合成为真实 MMU 使用的表，解决了"guest 页表不能直接用"的问题；但 guest 每次修改页表都要 hypervisor 拦截并重建影子表，开销高。硬件嵌套页表（EPT/NPT）在硬件中完成两级翻译，免去影子表维护。

---

# Lecture 28: 课程复习（Course Review）

## 概述

期末复习课：梳理三大主题（并发管理、内存管理、存储管理）的全部知识点，并提炼课程最重要的五条**大思想（big ideas）**：虚拟化、并发、原子性、局部性、分层。

## 知识地图（Lecture 28）

```
┌─ 并发管理 ──────────────────────────────────────────┐
│ 进程与线程: 创建、调度                               │
│ 同步: 竞态/不一致, 锁/条件变量/monitor, 锁的实现     │
│ CPU调度: 时间片, round robin, 优先级                 │
│ 死锁: 四条件(互斥/不可抢占/持有并等待/循环等待),     │
│       检测与预防(全局锁序)                           │
├─ 内存管理 ──────────────────────────────────────────┤
│ 链接: 静态与动态                                     │
│ 动态分配: 栈与堆, 悬垂指针/内存泄漏                  │
│ 重定位: 静态/动态(基址界限, 分段, 分页)              │
│ x86-64 页表, TLB, OS与地址空间, 碎片                 │
│ 请求调页: 按需取页/预取, 置换(随机/FIFO/MIN/LRU/     │
│   时钟/全局vs局部), 抖动                             │
├─ 存储管理 ──────────────────────────────────────────┤
│ 磁盘: 机制/操作/中断/PIO/DMA                         │
│ 文件: 访问模式, inode, 块布局(连续/链表/多级索引/    │
│   FAT/Unix), 块大小权衡                              │
│ 空闲空间: 链表/位图                                  │
│ 缓存: 块缓存, 延迟写                                 │
│ 磁盘调度: FIFO/SPTF/CSCAN                            │
│ 目录: 硬链接/符号链接                                │
│ 崩溃恢复: fsck/有序写/预写日志                       │
│ 闪存: FTL                                            │
└──────────────────────────────────────────────────────┘
期中截止: 链接/动态分配/静态与动态重定位(基址界限/分段/分页)
```

## 五条大思想（带走的核心）

1. **虚拟化（Virtualization）**：把一样东西变成另一样或许多个——CPU→线程、存储→文件、主存→地址空间（乃至整台机器→虚拟机）。
2. **管理并发（Managing Concurrency）**：同步是最难的部分——竞态、死锁、调度都源于"多个执行流共享资源"。
3. **原子性（Atomicity）**：让一组操作看起来不可分割——同步、文件系统一致性（事务/日志）都靠它。
4. **局部性（Locality）**：过去常能预测未来——调度（SRPT 近似）、TLB、分页（LRU/时钟）、文件缓存全部建立在此假设上。
5. **分层（Layering）**：用高层抽象隐藏底层细节——把难题解决掉，让其他人生活更美好。

## 后续课程建议（Lecture 28）

CS112（内核实现项目）、CS140E（OS 设计与实现）、CS240（OS 高级专题）、CS143（编译器）、CS144（计算机网络）、CS145（数据库）、CS244C（高级网络与分布式系统）。

## 思考题（总复习）

1. **问题**：用"虚拟化"一句话解释线程、文件、地址空间、虚拟机四个抽象。
2. **答案**：线程虚拟化 CPU（一个核变多个执行流）、文件虚拟化磁盘（块集合变命名字节序列）、地址空间虚拟化内存（物理内存变每进程私有虚拟空间）、虚拟机虚拟化整台机器（一台物理机变多台完整机器）。
3. **问题**：原子性在"锁"和"日志文件系统"中分别起什么作用?
4. **答案**：锁用原子操作（关中断/原子读改写）保证临界区互斥；日志文件系统用"事务"把一组日志条目变为原子单元（全有或全无），保证崩溃后文件系统一致。
5. **问题**：局部性如何同时解释 TLB、LRU 置换和磁盘调度?
6. **答案**：TLB 靠时间/空间局部性缓存翻译（命中率>95%）；LRU 用"过去最久未用=未来最久不用"预测替换；磁盘调度把请求按位置排序（SCAN/CSCAN）利用顺序访问局部性减少寻道——三者都是"过去预测未来"。

---

# 核心系统调用与 API 速查表（Quick Reference）

## 进程与线程

| API / 系统调用 | 头文件 | 说明 |
| --- | --- | --- |
| `fork()` | `<unistd.h>` | 创建子进程（复制当前进程）；父进程返回子 PID，子进程返回 0 |
| `execvp(path, argv)` | `<unistd.h>` | 用新程序覆盖当前进程的代码/数据；成功不返回 |
| `waitpid(pid, &status, opts)` | `<sys/wait.h>` | 等待子进程结束并回收退出状态（防僵尸） |
| `getpid() / getppid()` | `<unistd.h>` | 获取当前进程/父进程 PID |
| `exit(status)` | `<stdlib.h>` | 终止进程（返回状态给父进程） |
| `std::thread t(f, args)` | `<thread>` | C++ 创建线程；`t.join()` 等待，`t.detach()` 分离 |
| `pthread_create/join` | `<pthread.h>` | POSIX 线程创建/等待（C 语言） |
| `clone(fn, stack, flags, …)` | `<sched.h>` | Linux 底层线程/进程创建（fork 的实现基础） |
| `setuid/setgid` | `<unistd.h>` | 放弃/切换特权（最小权限原则） |

## 同步

| API | 头文件 | 说明 |
| --- | --- | --- |
| `std::mutex` | `<mutex>` | 互斥锁：`lock()` 阻塞获取，`unlock()` 释放，`try_lock()` 非阻塞 |
| `std::lock_guard` / `std::unique_lock` | `<mutex>` | RAII 锁包装（构造加锁、析构解锁）；`unique_lock` 可配合条件变量 |
| `std::condition_variable` | `<condition_variable>` | 条件变量：`wait(lock)` 原子释放锁并睡眠；`notify_one()` / `notify_all()` 唤醒 |
| `std::atomic<T>` | `<atomic>` | 原子类型（如 `std::atomic<bool>`），编译为原子指令（如 `exchange`） |
| `pthread_mutex_lock/unlock` | `<pthread.h>` | POSIX 互斥锁 |
| `pthread_cond_wait/signal` | `<pthread.h>` | POSIX 条件变量 |

## 内存

| API / 系统调用 | 头文件 | 说明 |
| --- | --- | --- |
| `malloc / free`、`new / delete` | `<stdlib.h>` / C++ | 堆内存分配/释放（走空闲链表/slab；不足时向 OS 申请） |
| `mmap(addr, len, prot, flags, fd, off)` | `<sys/mman.h>` | 把文件/匿名区域映射进虚拟地址空间（Assign5 核心） |
| `munmap(addr, len)` | `<sys/mman.h>` | 解除映射 |
| `mprotect(addr, len, prot)` | `<sys/mman.h>` | 修改页的访问权限（PROT_NONE/READ/WRITE，制造伪缺页） |
| `msync(addr, len, flags)` | `<sys/mman.h>` | 把脏页写回文件 |
| `brk/sbrk` | `<unistd.h>` | 扩展/收缩数据段（堆） |
| `std::shared_ptr<T>` | `<memory>` | 引用计数智能指针（注意循环引用需 weak_ptr） |

## 文件与目录

| API / 系统调用 | 头文件 | 说明 |
| --- | --- | --- |
| `open(path, flags, mode)` | `<fcntl.h>` | 打开文件/创建设备描述符；内核逐级查目录、载入 inode |
| `close(fd)` | `<unistd.h>` | 关闭文件描述符 |
| `read(fd, buf, n)` / `write(fd, buf, n)` | `<unistd.h>` | 从当前偏移顺序读写（块缓存/延迟写） |
| `pread/pwrite(fd, buf, n, off)` | `<unistd.h>` | 指定偏移的随机读写（数据库常用） |
| `fsync(fd)` / `fdatasync(fd)` | `<unistd.h>` | 强制把脏块刷到磁盘（耐久性） |
| `lseek(fd, off, whence)` | `<unistd.h>` | 移动文件偏移 |
| `ftruncate(fd, len)` | `<unistd.h>` | 设置文件长度 |
| `stat / fstat / lstat` | `<sys/stat.h>` | 获取文件元数据（inode 信息：大小、权限、nlink；lstat 不跟随符号链接） |
| `link(old, new)` | `<unistd.h>` | 创建硬链接（共享 inode，nlink++） |
| `symlink(target, linkpath)` | `<unistd.h>` | 创建符号链接（内容为路径字符串） |
| `unlink(path)`（`rm`） | `<unistd.h>` | 删除目录项（硬链接计数--；计数归零才释放数据） |

| API / 系统调用 | 头文件 | 说明 |
| --- | --- | --- |
| `mkdir/rmdir/opendir/readdir` | `<sys/stat.h>` `<dirent.h>` | 目录操作 |
| `chmod/chown` | `<sys/stat.h>` | 修改权限/属主（保护机制） |
| `dup/dup2` | `<unistd.h>` | 复制文件描述符（重定向，如 `ls > out`） |
| `pipe(fds)` | `<unistd.h>` | 创建管道（进程间通信，生产者–消费者的系统形态） |
| `fstrim / TRIM` | 命令行 / `ioctl` | 通知 SSD 释放的块（闪存管理） |

## 进程间通信与其它

| API / 系统调用 | 头文件 | 说明 |
| --- | --- | --- |
| `pipe`、`dup2` | `<unistd.h>` | 管道与重定向 |
| `signal / sigaction` | `<csignal>` `<signal.h>` | 信号处理（如捕获 SIGSEGV 模拟缺页） |
| `nice(inc)` / shell `nice -n` | `<unistd.h>` | 调整进程调度优先级（–20..+19） |
| `ptrace` | `<sys/ptrace.h>` | 进程追踪（gdb 的基础） |
| `getuid/setuid` | `<unistd.h>` | 用户身份查询/切换（信任边界） |

---

> **备注**：本笔记中的图表（ASCII 图）根据讲义幻灯片重新绘制；代码示例为教学用途的简化版本。讲义中还有更多深入案例（如 Lecture 5 的 Pipe 逐版本推演、Lecture 9 的链接器三遍扫描实例、Lecture 20 的读块 23/1040 例子），建议结合原始 PDF 学习：
> https://web.stanford.edu/class/archive/cs/cs111/cs111.1266/