第一部分：计算机安全的基本概念与系统基础

安全策略、预期机器、虚拟地址空间、程序运行与特权边界

计算机安全课程

内容

安全的定义与边界

1. 安全策略与状态转移
2. 安全判定的理论边界
3. 预期机器与具体实现

系统实现

4. 虚拟地址空间与内存保护
5. 程序运行模型与上下文加载
6. 系统调用与特权边界

主线

策略定义允许的行为，机制建立并强制边界，攻击寻找实现中通向坏状态的路径

安全策略与状态转移

安全首先规定允许发生的行为

常见但不准确的问法

- 有没有防火墙?
- 有没有杀毒软件?
- 有没有加密?
- 有没有 MFA?

这些属于**安全机制**

安全策略回答的问题

- 谁可以访问什么?
- 在什么条件下可以访问?
- 哪些修改是允许的?
- 哪些信息流动是不允许的?
- 出错后必须保留什么证据?

安全策略

安全首先是一组允许与禁止的行为约束。没有策略，就无法判断系统是否安全

把安全写成状态转移



安全策略

设 W 为允许的状态转移集合，则

$$(\sigma, a, \sigma') \in W$$

表示：系统在状态 σ 下执行操作 a 并到达 σ'，这是策略允许的

策略违背

如果

$$(\sigma, a, \sigma') \notin W,$$

这次状态变化就是策略违背。安全机制负责阻止、检测或恢复这类转移

状态包含什么

状态比主体–客体–操作更一般

安全状态是所有会影响安全判断的事实。它不只包含谁访问哪个文件

可能属于状态的事实

- 身份、角色、所属组织
- 文件、记录、密钥、进程
- 会话是否认证、令牌是否过期
- 时间、地点、设备、网络环境
- 当前协议轮次、随机数、密钥版本
- 是否审批、是否留痕、用途是否匹配

操作也不只是 read/write

- 修改一条成绩记录
- 导出一批个人信息
- 建立一个网络会话
- 执行一次签名或认证
- 运行一个进程
- 多方共同计算一个函数

例子：教师提交成绩

操作

$$a = \mathrm{SubmitGrades}(\text{教师}, \text{课程}, \text{成绩库}, \text{日志})$$

这次操作涉及身份、时间、权限、事务和审计的状态转移

允许

- 教师身份已认证
- 只修改本人任课课程
- 当前仍在录入窗口
- 修改经业务规则检查
- 审计日志追加且不可由普通用户删除

禁止

- 学生账号修改成绩
- 教师修改其他课程
- 绕过业务系统直接写数据库
- 发布后无审批再次改分
- 删除修改记录

威胁、漏洞、风险——常见的概念

先有策略违背，后有风险分析

非法状态转移 → 可行路径 → 发生可能性与后果 → 风险

术语	它描述什么	成绩系统例子
资产	需要维持在安全状态中的对象、能力或信息	成绩库、身份信息、提交权限、审计日志
威胁	可能促成非法转移的主体、事件或能力	盗号、内鬼、恶意学生、勒索软件
漏洞	让非法转移可执行的缺口	越权接口、弱口令、可删除日志、SQL 注入
风险	非法转移发生的可能性及其后果	改分、隐私泄露、追责失败、业务中断

安全机制：约束状态转移的手段

安全机制（security mechanism / control）

安全机制用于实现、强制、检测或恢复安全策略

管理	物理	系统	密码
审批	门禁	OS 权限	加密
职责分离	机房	沙箱	MAC
最小权限	介质控制	隔离	签名
培训与追责	设备销毁	日志	密钥协商
		MFA	零知识

安全服务：安全策略中的典型要求

安全服务	从状态转移看	例子
机密性	禁止未经授权的读取、推断或信息流	学生不能看到他人成绩
完整性	禁止未经授权的修改、删除或回滚	成绩修改必须走业务流程
可用性	合法转移在需要时能够发生	教师应能在窗口内提交成绩
认证	确定参与者身份是否与系统状态一致	当前会话是否真属于某教师
授权	判断该身份在当前状态下能否执行操作	只能改本人任课课程
审计	保存可验证的状态转移证据	谁在何时改了什么

同一种机制，在不同策略下可能完全不同

场景	策略重点	典型机制	主要取舍
成绩系统	完整性、可追责；发布后修改受控	RBAC、审批、日志、提交窗口	便利性 vs. 可追责
医院病历	诊疗关系授权；急救时允许 break-glass	关系授权、强审计、急诊例外	机密性 vs. 可用性
政府数据	分级、最小知悉、外传受控	分级分类、专网、DLP、介质管控	灵活性 vs. 泄露风险

机制与策略

机制是否合适，取决于它能否以可接受成本实现给定策略

安全判定的理论边界

工程语言如何变成数学语言?

工程说法	数学说法
系统当前所有安全相关事实	一个状态 s
所有可能的系统情况	状态空间 S
允许出现的系统情况	授权状态集合 A
不允许出现的系统情况	非授权状态集合 U
用户或程序执行一次操作	状态转移 $s \to s'$
系统永远不进入坏状态	安全性质

对应关系

前面的工程描述可以写成状态集合和状态转移

形式化的安全定义

状态空间二元划分

设系统所有可能状态为 S，根据安全策略把它划分为

$$S = A \cup U, \qquad A \cap U = \emptyset,$$

其中 A 是授权状态，U 是非授权状态

安全系统

若系统从授权初始状态 $s_0 \in A$ 出发，在所有允许的执行过程中都不会进入 U，则系统满足该安全策略



是否存在完美安全验证器?

理想目标

设有程序 $V(P, x)$：输入任意程序 P 和输入 x，它总能停机，并准确回答

“运行$P(x)$ 是否会进入坏状态U?”

固定实例

对某一个已经**固定**的程序和输入，命题当然有确定真值

统一算法

问题在于是否存在一个**统一算法**，能对**任意程序、任意输入**都在有限时间内给出正确答案

要求适用于任意程序和任意输入。

为什么它会碰到停机问题?

用一个固定的通用图灵机就够了

固定一台通用图灵机 U。把任意机器 M 和输入 w 编码为 $\langle M, w\rangle$，则

$$U(\langle M, w\rangle)$$

可以模拟 $M(w)$。判断这台**固定机器** U 对任意编码输入是否停机，就已经包含了通用停机问题

归约到安全判定

构造程序 $P_{M,w}$:

- 它先模拟 $M(w)$
- 若 $M(w)$ 停机，则 $P_{M,w}$ 进入坏状态 U
- 若 $M(w)$ 永不停机，则 $P_{M,w}$ 永不进入该坏状态

如果完美安全验证器存在，我们就能用它解决停机问题；这样的统一验证器不存在

找到漏洞与证明安全：不对称

找到一个反例

只要找到一条有限执行轨迹

$$s_0 \to s_1 \to \cdots \to s_k \in U,$$

就已经证明系统**不安全**。

- 一个崩溃输入
- 一条越权路径
- 一组触发竞争条件的调度

证明没有任何反例

目前没找到并不等于永远不存在

- fuzzing 可以不断寻找见证
- 静态分析必须做近似
- 模型检验需要有限模型或其他约束

不对称性

很多漏洞发现问题只需找到一个见证即可证伪；一般程序的安全证明不能靠无限搜索完成

真实计算机是有限的：是否问题可解？

物理有限性

若把**整台封闭机器**的所有安全相关状态都限制在有限比特中，那么它确实可以建模为一个巨大的有限状态系统

理论上

有限状态系统的可达性是可判定的：把所有状态和边遍历完，总能知道 U 是否可达

但要小心这个视角

真实系统还持续接收网络、时间、设备和用户输入。只有把这些环境也纳入有限模型，才能得到封闭的有限状态系统

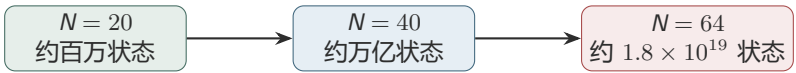
有限性与可行性

有限不等于工程上可以穷举

工程障碍：状态空间爆炸

如果一个系统的安全相关状态由 N 个二进制位决定，潜在状态数最多为

$$2^N.$$



有限系统

对有限系统，问题从不可判定变成原则上可判定；但状态数常以指数速度增长，穷举在工程上仍然不可行

两类做法

抽象、模型检验、符号执行和静态分析用紧凑表示避免显式枚举全部状态；类型系统、沙箱和硬件保护则直接限制系统允许发生的状态转移

危险状态能否用少量规则描述?

容易概括的危险模式

- 长度必须小于缓冲区
- 指针不得为空
- 用户输入不得直接拼入 SQL
- 某个角色永远不得执行某操作

少量规则就能覆盖大量坏状态

难以概括的危险模式

- 特定线程交错触发的竞争
- UAF 与复杂对象生命周期
- 多阶段协议中的状态错配
- 多模块组合后才出现的异常路径

边界越复杂，越难靠少量规则完整描述

规则复杂度

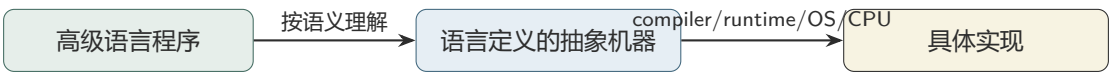
危险状态若能用少量规则描述，分析和强制通常更容易

预期机器与具体实现

程序员面对的是一台抽象机器

抽象机器（abstract machine）

高级语言通过语法和语义定义程序员所面对的计算模型。程序员按照这个模型理解程序，而不必考虑程序最终由晶体管、流水线、页表和物理存储器怎样实现



例如

在 C 程序员看来，`f()` 是一次函数调用，局部变量是程序对象，`return` 表示返回调用者。
至于返回地址具体保存在哪里、由哪条机器指令取出，属于抽象机器的**实现细节**

预期机器：程序应当具有的有效行为

预期机器（expected machine）

这里把程序员和系统设计者**期望程序具有的有效行为**称为 **预期机器。**
它包含高级语言语义，也包含系统设计中希望成立的安全约束

程序员预期

- 数组写入只修改该数组
- 函数返回到调用它的位置
- 普通数据不能任意改变控制流
- 一个进程不能修改另一个进程
- 用户程序不能直接修改内核状态

具体实现必须做到

- 编译器生成机器代码
- ABI 规定调用约定
- CPU 实现 call/ret
- MMU 实现地址隔离
- OS 实现进程和权限
- 运行时维护语言所要求的对象关系

安全问题

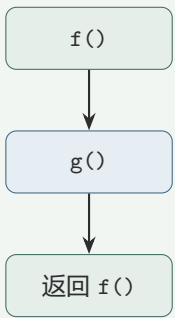
具体实现有时允许预期模型没有考虑的行为，攻击者可以利用这些行为

例子：函数调用的预期语义与具体实现

预期机器

程序员看到：

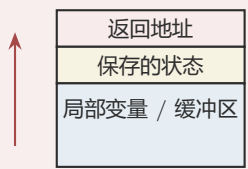
$$f() \to g() \to return$$



传统机器上的一种实现

调用时，普通栈中同时保存：

- 局部变量
- 缓冲区
- 保存的寄存器
- 返回地址



实现泄漏：数据错误变成了控制流错误



预期机器中的理解

向数组写数据和决定函数返回到哪里本来是两种不同性质的操作

具体实现中的事实

传统栈把普通数据和控制流元数据放在同一片可写内存附近。因此，一个本来属于**数据完整性**的问题，可能沿着具体实现路径变成**控制流完整性**问题

语义与实现的差距

上层认为不能发生的事，在底层实现中却可能发生；攻击者可以利用这些额外能力

C：现代系统软件的源头，也是安全问题的重要源头

程序员心中的预期机器

写下 `buf[i] = x` 时，程序员通常只希望它修改数组中的一个元素，而不会认为一次索引错误就应该获得修改控制数据乃至劫持控制流的能力

C 给程序员很大自由

- 裸指针和指针运算
- 数组通常无边界检查
- 手工管理对象生命周期
- 数据表示接近机器
- 大量错误落入 undefined behavior

高手能够写出符合预期的 C 程序，但语言本身并不强制程序始终如此

错误可以穿透抽象层

程序错误 → 内存破坏 → 控制数据破坏 → 控制流劫持

C 的高效和贴近机器，也使程序错误更容易获得**机器级后果**

不同层次的约束

Canary、NX、ASLR、CFI、CET 与内存安全语言，都在不同层次缩小预期行为与机器实际允许行为的差距

C/Unix：成功的工程取舍，也留下长期代价

1970 年代的约束与选择

面对内存小、CPU 慢、硬件差异大的机器，C/Unix 选择

简单　贴近硬件　低开销　相信程序员

换来易实现、易移植和性能可预测

工程优势

- 很薄的语言与系统抽象
- 指针接近机器地址
- 很少强制运行时检查
- OS 接口简单而统一

长期代价

- 内存安全长期依赖程序员自律
- 错误容易获得机器级后果
- 风险假设被语言、库和 ABI 继承
- 后来不断由 NX、ASLR、Canary、CFI、CET 补强

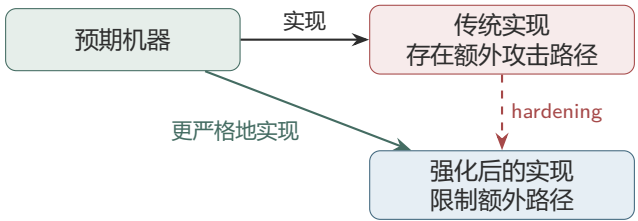
资源约束

稀缺资源会变化。适应当时资源条件的工程取舍，今天需要重新权衡性能、兼容性与安全

安全防御：让实际执行更接近预期机器

不改变高级语言的抽象机器

对于 NX、Stack Canary、CFI、Shadow Stack、CET 等机制，高级语言中函数、数组、指针和函数调用的基本语义并没有改变。
改变的是抽象机器在真实系统上的实现方式和约束方式



实现不变量

具体执行需要保持合法控制流、合法内存访问和权限边界

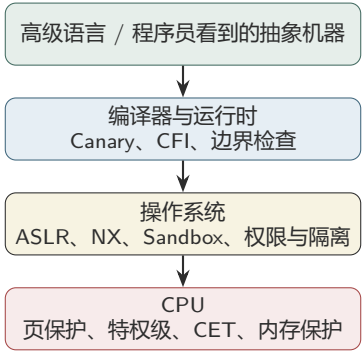
现代防御怎样强化预期机器的实现?

机制	预期性质	实现层增加的约束
NX / W⊕X	普通数据不应直接成为可执行代码	页表把可写性与可执行性分离
Stack Canary	栈对象越界不应悄悄改变调用状态	函数返回前检查关键栈状态是否被破坏
Shadow Stack / CET	函数应返回到合法调用点	返回地址另存于受保护区域，并在返回时检查
CFI	间接控制流只能到程序允许的位置	运行时检查间接 call/jmp/return 的目标集合
ASLR	攻击者不应稳定掌握关键对象的位置	使代码、库、栈等地址在不同执行间随机化
Sandbox / seccomp	程序只能使用被授予的系统能力	在系统调用和资源访问层限制可执行操作

实现层约束

这些机制没有重新定义 C/C++ 的函数或数组，而是在编译器、运行时、OS 或 CPU 层增加约束

实现约束可以放在不同层



Defense in depth

同一预期性质可由多个实现层共同保证：底层更难绕过，上层拥有更丰富的程序语义

强化实现与改变编程模型

强化原有实现

NX、ASLR、CFI、CET、Canary 等：

- 原有源程序基本不变
- 原有语言抽象基本不变
- 在实现层增加安全约束

这是本节主要讨论的路线

改变编程模型

换成具有不同语言语义的编程模型，例如：

- 更严格的类型系统
- 自动边界检查
- 所有权与生命周期
- capability-based language

这时程序员看到的抽象机器本身也发生了变化

两类变化

改变抽象机器 $\neq$ 强化抽象机器的实现

现代系统安全中的很多经典机制属于后者

从状态空间重新理解这一思想

状态空间视角

安全策略把状态空间划分为授权状态 A 与非授权状态 U。攻击者寻找的是：$A \longrightarrow U$ 的可达路径

脆弱的具体实现

实现中存在额外的底层状态和转移，使攻击者获得了预期模型之外的路径：

$$A \to \cdots \to U$$

强化后的具体实现

通过硬件、OS、编译器和运行时机制：

- 删除某些实现路径
- 对某些路径增加检查
- 隔离关键元数据
- 强制维持安全不变量

实现目标

防御通过约束具体实现，使真实机器的可观察行为尽可能符合预期机器

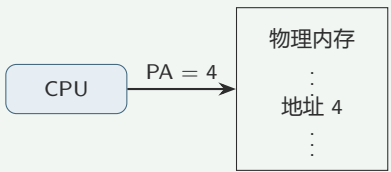
虚拟地址空间与内存保护

程序使用的地址，通常并不是内存芯片上的物理地址

深入理解计算机系统第 9 章

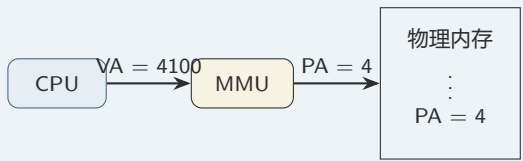
从物理地址到虚拟地址

如果直接使用物理地址



CPU 发出的地址就是内存中的实际位置

现代系统：虚拟寻址



程序产生的是**虚拟地址 VA**
MMU 在真正访问内存前，把 VA 转换成**物理地址 PA**

关键变化

程序不再直接决定自己访问哪一个物理内存位置

地址空间：同一个字节可以有两种地址

虚拟地址空间

如果是 32 位程序，虚拟地址可以写成

$$0, 1, 2, \ldots, 2^{32} - 1$$

程序因此看到一段连续的约 4 GiB 地址空间

程序中的

- 指令地址
- 指针
- 栈地址
- malloc 返回的地址

通常都是虚拟地址

物理地址空间

物理地址则描述机器实际可访问的存储位置

一般情况下

$$\text{Virtual Address} \neq \text{Physical Address}$$

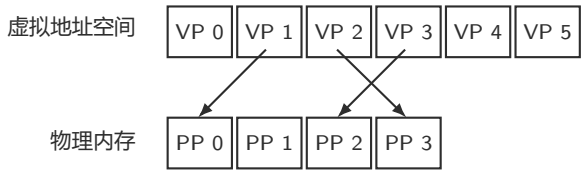
虚拟地址与物理地址的对应关系不是程序自己决定的

OS 建立映射，MMU 使用映射

虚拟存储的基本思想

为什么要分页

如果逐字节记录虚拟地址到物理地址的映射，映射表将大得无法使用
因此，虚拟地址空间和物理内存都被划分成固定大小的**页（page）**
以常见的 $4\,\mathrm{KiB} = 2^{12}$ 字节页面为例



Virtual Page

虚拟地址空间中的页称为虚拟页，记为 VP

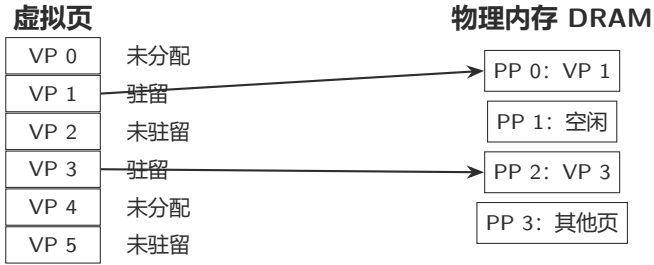
Physical Page

物理内存中的页称为物理页 PP，也常称 page frame

重要

相邻的虚拟页不要求映射到相邻的物理页

CS:APP 的视角：DRAM 是虚拟页的缓存



未分配
程序还没有使用这个虚拟页

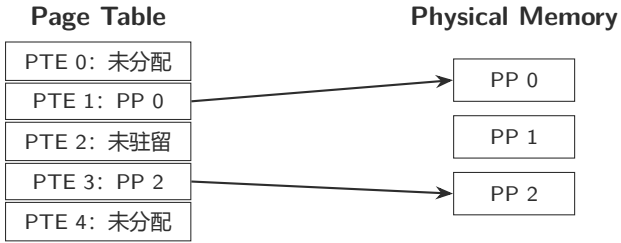
驻留
页面当前在 DRAM 中

未驻留
页面已经分配，但当前不在 DRAM 中

缓存观点

DRAM 可以看成虚拟地址空间中当前活跃页面的高速缓存

页表：记录虚拟页与物理页的关系



Page Table Entry，PTE

每个虚拟页对应一个页表项
页表项至少需要告诉 MMU

- 页面当前是否在物理内存中
- 如果在，它位于哪一个物理页
- 对这个页面允许什么类型的访问

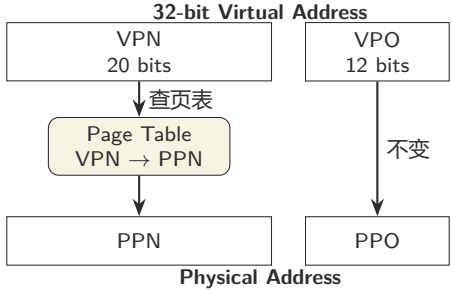
页表本身在哪里？

页表也是数据结构，本身存放在物理内存中

OS 创建和修改页表
MMU 在地址翻译时读取页表

一个虚拟地址怎样变成物理地址

假设使用 32 位虚拟地址，页面大小为 $4\,\mathrm{KiB} = 2^{12}$ bytes



为什么页内偏移不用翻译?

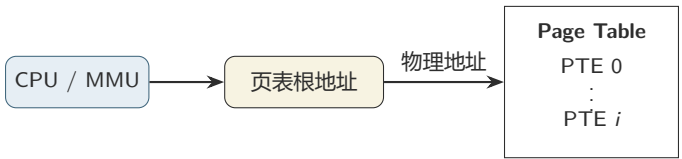
虚拟页和物理页大小相同
所以虚拟页中的第 x 个字节，映射后仍然是物理页中的第 x 个字节
因此

$$\boxed{\mathrm{VPO} = \mathrm{PPO}}$$

真正需要转换的是**页号 VPN**

页表从哪里开始?

MMU 必须知道当前页表在物理内存中的位置



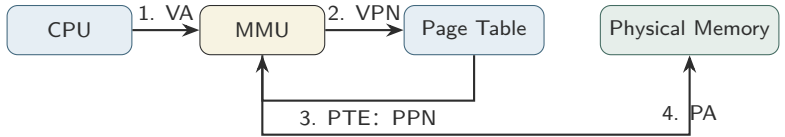
Page Table Base Register

概念上，CPU 中有一个寄存器记录当前地址空间页表的根地址
在 x86 中，可以把 **CR3** 理解为指向当前页表层次结构的根

上下文切换

进程 A 运行时，CPU 使用 A 的页表
切换到进程 B 时，OS 切换当前地址空间的页表根
因此同一个虚拟地址在不同进程中可以得到不同的物理地址

页面已经在内存中：Page Hit



CPU 执行指令并产生一个虚拟地址 VA

MMU 取出 VPN，找到对应 PTE

PTE 给出目标物理页号 PPN

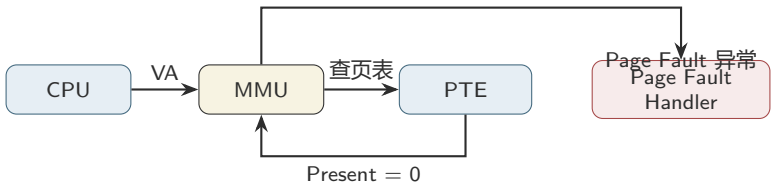
PPN 与页内偏移组合成 PA

使用 PA 完成真正的物理内存访问

对程序是透明的

用户程序只是执行了一条普通的 load/store 指令

页面不在内存中：Page Fault



发生了什么?

目标虚拟地址是合法的，但对应页面目前不在 DRAM 中
于是 MMU 无法立即给出目标物理地址

重要

这不是普通的函数调用
CPU 通过异常机制停止当前用户程序，转入内核

Page Fault 之后发生什么



内核确认这个虚拟地址是否合法

找到目标页面的后备内容

找到一个可用物理页，必要时换出其他页面

将目标页调入 DRAM 并修改相应的页表项

从异常返回

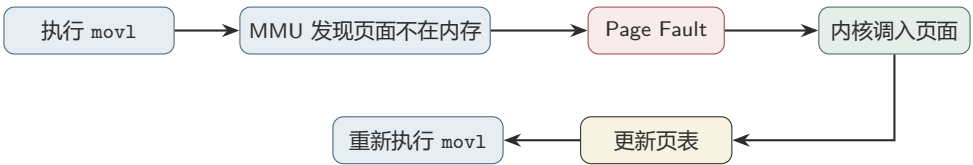
CPU **重新执行刚才没有完成的指令**

为什么发生 Page Fault 后程序还能继续

假设 32 位程序正在执行

```
movl (%eax), %ebx
```

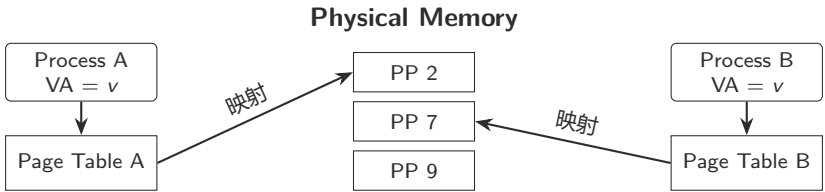
其中 %eax 保存一个虚拟地址 v



关键

第一次执行时，这条指令并没有完成
内核把页面准备好以后，CPU 再执行一次同一条指令
第二次地址翻译成功，程序才继续向下运行

每个进程都有自己的地址空间



同一个虚拟地址可以对应不同的物理地址

例如

$$A : v \longrightarrow PP2$$

$$B : v \longrightarrow PP7$$

两个进程都认为自己访问的是地址 v，实际上最终访问的是不同的物理页面

进程隔离的硬件基础

A 的页表中没有到 B 私有页面的映射，普通用户程序不能仅靠构造一个指针访问 B 的内存

共享也是通过页表实现的



共享库的代码	进程间共享内存	文件映射

页表同时提供两种能力

映射到不同物理页 $\Rightarrow$ **隔离**　　映射到同一个物理页 $\Rightarrow$ **共享**

页表不仅记录地址，还记录权限

概念化的 Page Table Entry

Present	User	Write	NX	Physical Page Number

地址翻译信息

- Present
- Physical Page Number

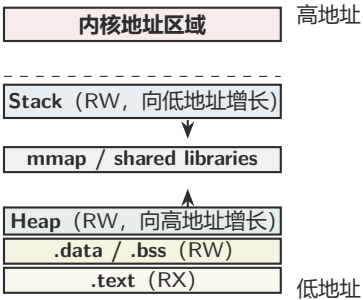
保护信息

- User / Supervisor
- Read / Write
- Execute / NX

一次地址翻译也是一次访问控制

MMU 不仅问这个地址在哪里，还要问当前 CPU 权限下，这次读 / 写 / 执行是否允许

典型进程虚拟地址空间



现在可以真正理解这张图

这里画出的不是物理内存中的几个固定区域
它们是**虚拟地址范围**
页表决定

- 每个范围对应哪些物理页
- 哪些页面当前驻留
- 哪些页面可读、可写、可执行

由内核记录

`task_struct`: 进程的专属任务结构体，包括：

`mm_struct`: 进程的虚拟存储状态，包括：

`vm_area_struct`: 存储区域记录表

为什么不直接使用一张巨大的页表

对于 32 位虚拟地址

$$2^{32} \text{ bytes}$$

如果页面大小为 4 KiB

$$\frac{2^{32}}{2^{12}} = 2^{20}$$

因此整个虚拟地址空间共有 2^{20} 个虚拟页

如果每个页表项占 4 字节

$$2^{20} \times 4 = 4\,\text{MiB}$$

即使一个进程只实际使用很少的虚拟地址，单级页表仍然要为整个虚拟地址空间准备页表项

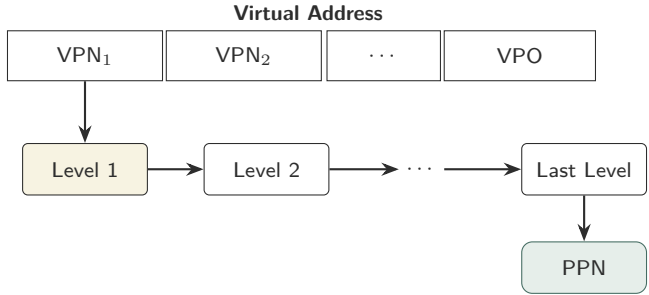
问题

大量虚拟地址根本没有使用，却仍然需要为它们准备 PTE

解决办法

使用**多级页表**，只有某一片虚拟地址真正被使用时，才为它建立下一层页表

多级页表：把 VPN 再分成几段



树结构

每一级页表项可以指向下一层页表
最后一级页表项才给出目标物理页号

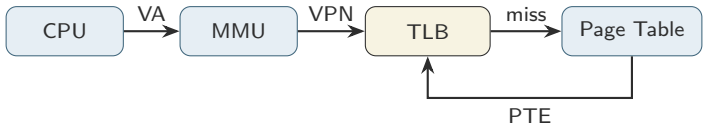
没有使用的大块虚拟地址范围，可以根本不建立对应的下级页表

x86

实际 x86 / x86-64 使用多级页表，CR3 指向当前地址空间页表层次结构的根

访存速度

页表本身位于内存中，如果每一次访问数据前都必须先访问页表，地址转换会产生明显的额外开销



TLB：Translation Lookaside Buffer

TLB 是专门缓存最近地址转换结果的小型高速缓存

TLB hit：直接得到 PPN，不必访问内存中的页表
TLB miss：再去查页表，并把结果放入 TLB

TLB 只是加速

TLB 只是地址转换的加速器，*VA* → 页表映射 → *PA* 才是基本机制

用户态、内核态与地址空间是不同概念

态（mode）是 CPU 权限状态

- 用户态：典型 x86 Ring 3
- 内核态：典型 x86 Ring 0
- 决定能否执行特权操作

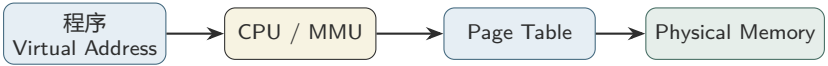
地址空间描述地址映射

- 用户地址保存程序、库、栈、堆等
- 内核也有自己的代码和数据映射
- 是否允许访问由页表权限和当前 CPU 权限共同决定

两者不同

Ring 0 / Ring 3 描述谁在执行　　虚拟地址空间描述这个地址映射到哪里

虚拟存储：OS 与 CPU 共同维护的抽象



OS 的工作

- 为进程建立和维护页表
- 分配、回收和调入页面
- 在进程切换时切换地址空间
- 决定页面权限

CPU / MMU 的工作

- 根据页表完成 VA $\rightarrow$ PA
- 检查读、写、执行以及用户/内核权限
- 发现缺页或越权时产生异常

基础机制

地址映射 + **页面权限** + **异常机制**构成了进程隔离、DEP/NX、ASLR 等安全机制的底层基础

思考题：进程能看到什么？

1. 一个用户进程能够直接访问哪些对象？程序代码、数据、heap、stack、共享库属于哪一类？`task_struct`、`mm_struct`、`vm_area_struct` 又属于哪一类？
2. 页表描述进程虚拟地址到物理地址的映射。页表本身存在哪里？用户进程能否直接读取或修改自己的页表？如果可以修改，会发生什么？

进一步思考

“存在于当前地址空间的映射中”和“用户态程序能够访问”是不是同一个概念？

进程生命周期与执行环境

fork 创建进程，fd 连接资源，exec 替换程序，wait 回收进程

一个进程包含什么



用户进程直接看到

自己的代码和数据、虚拟地址、寄存器、文件描述符整数

内核维护

页表、fd 表、打开文件对象、socket 对象、`task_struct`、`mm_struct` 等

本节的问题

创建一个进程时，哪些内容复制，哪些资源共享，哪些内容被新程序替换

Shell 启动程序



子进程

1. `fork()` 得到新的进程
2. 必要时调整文件描述符
3. `execve()` 换成要运行的程序

父进程

Shell 自己不会因为启动命令而消失
前台任务通常等待子进程，后台任务则继续接受命令

fork：一次调用，两次返回

最基本的 fork 程序

```
pid_t pid = fork();

if (pid < 0) {
    perror("fork");
}
else if (pid == 0) {
    printf("child: pid=%d\n", getpid());
}
else {
    printf("parent: child=%d\n", pid);
}
```

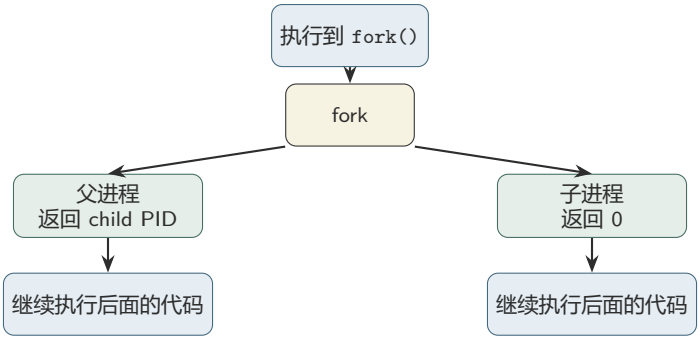
返回值

调用者	返回值
父进程	子进程 PID
子进程	0
失败	-1

关键

源代码中只有一次 `fork()`
成功后父、子两个进程都从 `fork()` 后面继续执行

fork 之后



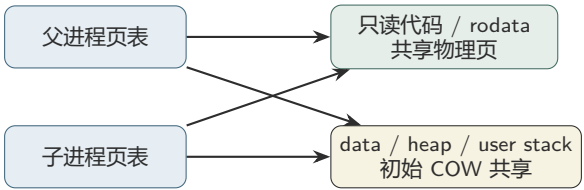
相似

子进程继承父进程当时的程序执行环境，因此从相同代码位置开始

不同

PID、调度状态、页表对象、进程元数据等已经属于两个独立进程
谁先运行由调度器决定

fork 后的内存：逻辑复制，物理尽量共享



Copy-on-Write

父子逻辑上各有自己的虚拟地址空间
私有可写页开始可映射到同一物理页，写入时触发 page fault 后再复制

所以 fork 不是简单 memcpy

复制的是进程看到的地址空间状态
底层物理页可以暂时共享

文件描述符：进程可见的是一个整数



用户看到

$0 =$ stdin　　$1 =$ stdout　　$2 =$ stderr

`open()`、`socket()`、`accept()` 等返回一个 fd

内核维护

fd 表项指向内核对象

普通文件的 open-file 对象保存文件偏移、打开状态等；socket 对象保存连接状态

fork 后的 fd：表分开，对象可以共享



普通文件

父子相应 fd 引用同一个 open-file 对象，因此共享文件偏移

socket

父子相应 fd 可以操作同一个连接
`close(fd)` 只释放当前进程的一个引用

引用

只要仍有进程持有这个对象的 fd 引用，对象就不能因为另一方的 `close()` 而消失

dup 与 dup2：复制的是描述符

dup

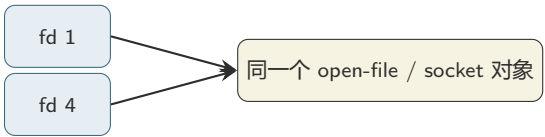
```
newfd = dup(oldfd)
```

返回当前最小的空闲 fd
新旧两个 fd 引用同一个 open-file 对象

dup2

```
dup2(oldfd,newfd)
```

让指定的 `newfd` 引用 `oldfd` 对应的对象
若 `newfd` 已打开，原引用先被关闭



例如 `dup2(4,1)`

之后 fd 1 与 fd 4 指向同一对象

不是复制文件或 socket 数据

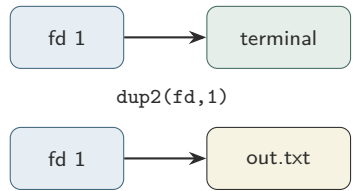
重定向：改变标准描述符的指向

把 stdout 重定向到文件

```
int fd = open("out.txt",
              O_WRONLY | O_CREAT | O_TRUNC,
              0644);

dup2(fd, STDOUT_FILENO);
close(fd);

printf("hello\n");
```



程序本身不用改变

程序仍然执行

```
write(1,...)
```

只是 fd 1 后面的对象变了

fork 的应用：进程并发服务器

典型服务器循环

```
for (;;) {
    connfd = accept(listenfd, ...);

    if (fork() == 0) {
        close(listenfd);
        echo(connfd);
        close(connfd);
        _exit(0);
    }

    close(connfd);
}
```

父进程

保留 `listenfd`
关闭自己的 `connfd`
继续 `accept()` 下一个客户

子进程

关闭不需要的 `listenfd`
保留 `connfd` 服务当前客户

为什么必须 close

fork 后父子都持有这些 fd；不需要的一方必须释放自己的引用

子进程终止：exit、zombie 与 wait



exit

`return` from `main` 最终会走到进程终止处理
libc 的 `exit()` 会做用户态清理，再进入内核终止进程

zombie

子进程停止执行后，内核仍保留 PID、退出状态等少量信息
父进程用 `wait()` / `waitpid()` 读取并回收

fork 后的子进程

子进程在 exec 失败等路径中常用 `_exit()`，避免重复刷新父进程复制来的 stdio 缓冲和执行继承的 atexit 处理

并发服务器：SIGCHLD 与 waitpid

子进程退出时回收

```
void sigchld_handler(int sig)
{
    int olderrno = errno;

    while (waitpid(-1, NULL, WNOHANG) > 0)
        ;

    errno = olderrno;
}
```

WNOHANG

没有已经退出的子进程时立即返回
handler 不必阻塞等待

为什么是 while

SIGCHLD 不是可靠的事件计数器
一次收到信号后，应尽可能回收所有已经退出的子进程

execve：保留进程身份，替换程序映像

被替换

成功执行 `execve()` 后，旧用户地址空间被新程序替换

- code / data
- heap
- user stack
- mmap / shared libraries
- 相应的 VMA 和页表映射

仍是这个进程

- PID 不因为 exec 改变
- 当前目录等进程属性可继续存在
- 未设置 close-on-exec 的 fd 可继续存在

内核重新建立

新的 `mm_struct` 所描述的用户虚拟空间、VMA、相应页表以及新的用户栈和入口上下文

exec 系列与 execve

libc 提供的 exec 家族

- `execv(path, argv)`
- `execl(path, ...)`
- `execvp(file, argv)`

这些接口在参数形式、PATH 查找和环境处理上有所不同

内核核心接口

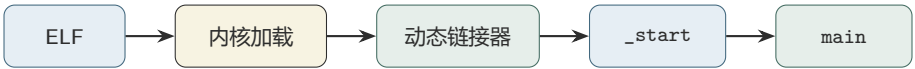
`execve(path,argv,envp)`

它负责把当前进程的用户程序映像替换成新的可执行文件

返回

`execve()` 成功时不会返回，因为原来的用户程序映像已经被替换；失败时才返回 -1

ELF、_start 与 main



execve 建立初始用户上下文

内核准备新地址空间和初始用户栈，并把用户 PC 设置到 ELF 规定的入口
这不是一次普通的用户态 `call`

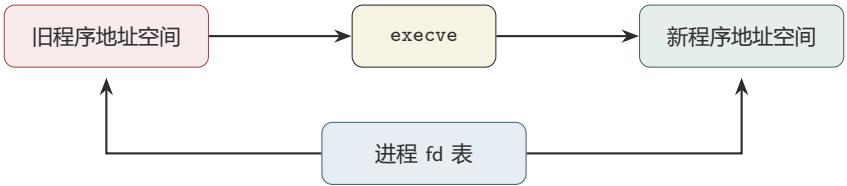
初始栈包含

- argc
- argv[]
- envp[]
- auxiliary vector
- 参数和环境字符串

main

`main()` 不是进程入口；启动代码和运行库准备完成以后才按照普通函数调用关系进入 `main`

exec 前后的文件描述符



默认

打开的 fd 可以跨 exec 保留
因此新程序可以直接继承旧程序已经打开的文件、pipe、socket

FD_CLOEXEC

设置 close-on-exec 的描述符会在 exec 成功时自动关闭
它控制的是 fd 是否跨过程序映像替换这一边界

fork + dup2 + exec：Shell 重定向

简化模型

```
pid = fork();

if (pid == 0) {
    int fd = open("out.txt", O_WRONLY|O_CREAT|O_TRUNC,
                  0644);
    dup2(fd, STDOUT_FILENO);
    close(fd);
    execlp("ls", "ls", "-l", NULL);
    _exit(127);
}

waitpid(pid, NULL, 0);
```

三个步骤

1. `fork`：产生新进程
2. `dup2`：安排新进程看到的 I/O 资源
3. `exec`：替换成真正要运行的程序

结果

`ls` 并不知道 Shell 做过重定向
它只是正常地向 fd 1 写数据

Daemon：长期运行的后台进程

守护进程

经典 Unix daemon 长期运行，不依赖某个控制终端存在



与终端分离

`setsid()` 建立新的 session 并脱离原控制终端
经典 double-fork 可使最终 daemon 不是 session leader，避免重新获得控制终端

运行环境

关闭不需要的继承 fd
stdin/stdout/stderr 通常重定向到合适对象，日志写入专门日志设施

inetd：fork、dup 与 exec 的综合例子

经典 Unix 超级服务器

`inetd` 是 daemon，统一监听多个网络服务，请求到来后再启动真正的服务器程序



inetd 父进程

保留监听 socket，继续等待其他请求

inetd 子进程

保留当前连接，把 socket 安排为标准 I/O，设置服务身份，然后 exec 真正服务器

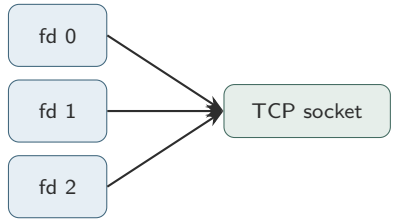
inetd 为什么把 socket 放到 0、1、2

子进程中的典型处理

```
dup2(connfd, STDIN_FILENO);
dup2(connfd, STDOUT_FILENO);
dup2(connfd, STDERR_FILENO);

if (connfd > STDERR_FILENO)
    close(connfd);

execve(server, argv, envp);
```



exec 以后

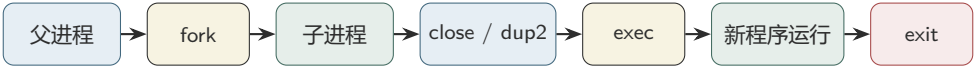
服务器程序可以直接

```
read(0,...)

write(1,...)
```

而不需要知道连接原来的 fd 编号

程序生命周期



fork
新的进程身份
继承当前执行环境

fd / dup
决定进程持有哪些资源
以及标准 I/O 指向哪里

exec / exit / wait
换程序、结束进程
父进程取得结束状态

思考题：进程到底继承了什么

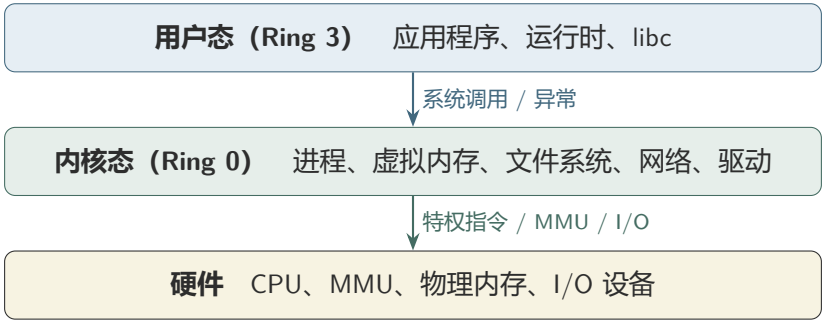
1. `fork()` 后父子进程的虚拟地址相同，为什么之后修改变量不会互相覆盖？
2. 父子进程各有自己的 fd 表，为什么对普通文件的读写位置仍可能互相影响？
3. `dup2(fd,1)` 为什么可以实现输出重定向？它复制了文件内容吗？
4. `execve()` 替换了 code、heap、stack 和 VMA，为什么网络 socket 却可以继续存在？`FD_CLOEXEC` 在这里控制什么？
5. 为什么进程并发服务器中父进程和子进程都必须关闭自己不再需要的 socket 描述符？
6. 为什么子进程已经退出，父进程仍然需要 `wait()` 或 `waitpid()`？

综合问题

Shell 的 `cmd > out.txt` 可以怎样用 `fork + open + dup2 + exec + waitpid` 实现？

系统调用与特权边界

三层系统：用户程序、内核与硬件



硬件提供边界机制

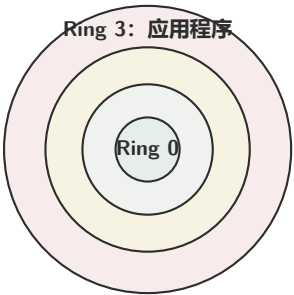
特权级、页保护、特权指令检查、异常/中断入口、定时器

OS 利用这些机制

构造进程、文件、用户、权限、调度、设备管理等高层抽象

用户程序不能直接控制硬件，只能通过内核提供的受控接口

Ring 3 为什么不能直接控制机器?



典型特权操作

- 修改页表基址 CR3
- 装载 IDT（`lidt`）
- 关闭/开启中断（`cli/sti`）
- 写控制 CPU 状态的 MSR
- 直接进行受保护的 I/O 操作

重点

CPU 在执行指令时检查当前特权级。
Ring 3 违规执行特权操作，CPU **不执行它，而是产生异常。**

普通函数调用与系统调用

普通函数调用

Ring 3 $\xrightarrow{\texttt{call}}$ Ring 3

- 权限级不变
- 继续使用用户栈
- 跳转目标由程序本身决定
- 如 `printf()`、`strlen()`

系统调用

Ring 3 $\xrightarrow{\texttt{int \$0x80}}$ Ring 0

- 进入 OS 预先设置的内核入口
- CPU 完成权限级切换
- 内核决定允许做什么
- 如 `read()`、`write()`、`fork()`

32 位 x86 Linux

经典系统调用接口使用 `int $0x80`。64 位系统使用不同的快速入口机制

32 位 Linux：一次系统调用怎样进入内核?

以 `write(1, buf, len)` 为例：

用户程序准备

```
movl $4,   %eax    # write
movl $1,   %ebx    # fd
movl $buf, %ecx    # buffer
movl $len, %edx    # length
int  $0x80
```

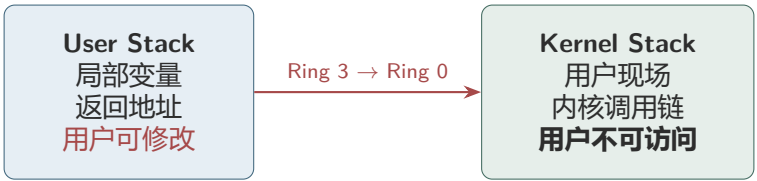
发生的事情

1. CPU 执行 `int $0x80`
2. 根据 IDT[0x80] 找到内核入口
3. Ring 3 → Ring 0
4. CPU 切换到内核栈并保存必要的用户态现场
5. 内核读取 EAX，得到系统调用号
6. 根据 EAX 分派到具体系统调用
7. 完成检查和操作后返回用户态

区别

CPU 只知道：**发生了 0x80 号软件中断。**
CPU 并不知道 EAX=4 表示 `write`；这是 Linux 内核定义的约定

为什么进入内核后要换栈?



问题不在方便，而在可信性

用户能够任意设置自己的 ESP，也能够修改用户栈内容。
如果内核把自己的返回地址、寄存器现场和局部变量继续放在用户栈上，攻击者就能修改内核的控制状态

问题：这三段代码分别会发生什么？

假设下面代码运行在普通用户进程中（32 位 x86 Linux）。

A：请求输出字符串

```
movl $4, %eax    # write
movl $1, %ebx    # fd
movl $msg, %ecx
movl $5, %edx
int $0x80
```

B：修改页表

```
movl %eax, %cr3
```

C：永远循环

```
L:
  jmp L
```

讨论 6–7 分钟

1. A 如何进入内核？CPU 与 OS 各做什么？
2. B 能否成功？若不能，是谁阻止它？
3. C 不主动调用内核，能否永久独占 CPU？为什么？

提示：分别从 system call、exception、interrupt 三条路径考虑

课堂题答案：三种进入内核的路径

A：主动请求

`int $0x80`

↓

系统调用入口

↓

EAX 分派

System Call

B：违规操作

写 CR3

↓

CPU 权限检查失败

↓

#GP 异常

Exception

C：强制抢占

死循环

↓

定时器产生中断

↓

内核调度

Interrupt

三种方式

系统调用：主动进入

异常：CPU 检测

中断：外部/异步事件

CPU 提供机制，OS 构造策略

问题	CPU / 硬件提供	OS 构造
内存隔离	MMU、页表权限、CR3、NX	进程虚拟地址空间、内存管理
特权边界	Ring、特权指令检查、异常入口	用户态/内核态边界、权限检查
系统调用	`int`、IDT、特权级切换	系统调用号、系统调用表、参数检查
调度	定时器中断、CPU 状态保存/恢复	进程状态、就绪队列、调度算法
设备访问	MMIO、端口 I/O、中断、DMA	设备驱动、设备抽象、访问控制

总结

CPU 不理解进程、文件、用户、权限这些高层概念；
OS 利用 CPU 提供的不可绕过机制实现这些对象和规则