EDA 系统软件分析和设计方法学

讲义

2026 年春季

毕朝日

复旦大学集成电路与微纳电子创新学院

目录

第二讲　软件设计平台

> The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
>
> — 艾兹赫尔 • 迪杰斯特拉（Edsger W. Dijkstra）

上一讲我们讨论了三个根本性问题：EDA 是什么，何为好的软件，以及人与软件应当建立怎样的关系。我们得出的核心结论是：在一个复杂性不断膨胀的世界里，管理复杂性的能力决定了软件的质量。但管理复杂性需要工具，正如建筑师需要脚手架，外科医生需要手术台——**软件工程师需要一整套支撑其工作的基础设施，这套基础设施就是软件设计平台。**

本讲将从底层向上，逐层剖析软件设计平台的构成：计算机硬件提供了什么样的物理基础？操作系统如何在硬件之上创造了一个有序的虚拟世界？编程语言和库又如何在操作系统之上搭建了更高层的抽象？最后，当单机的算力不再够用，分布式计算和云平台如何将这一切延伸到网络的尺度？这些问题看似与 EDA 没有直接关系，实则关系密切——**EDA 工具本身就是软件设计平台的产物，同时也是芯片设计师的软件设计平台。**理解平台的原理，是理解 EDA 工具设计决策的前提。

2.1　软件设计平台：层层抽象的艺术

2.1.1　何为软件设计平台

开发一款应用软件——无论是一个网页浏览器还是一款布局布线工具——不可能从零开始。你需要一台计算机来运行代码，需要一个操作系统来管理硬件资源，需要一种编程语言来表达算法，需要各种库来避免重复造轮子。这些构成了你工作所依赖的软硬件环境，统称为**软件设计平台**。

软件设计平台的结构可以用一个分层模型来理解：最底层是计算机硬件，提供计算、存储和通信的物理能力；其上是操作系统，将硬件资源抽象为可管理的逻辑实体；再上

是编程语言，提供表达计算过程的符号系统；最上层是各类库和框架，封装了常见任务的成熟实现。应用软件坐落在这座抽象金字塔的顶端，享受着每一层提供的服务。

这种分层结构的核心智慧在于**关注点分离**——每一层只需要关心自己直接下方那一层提供的接口，而不必知道更底层的实现细节。应用程序员不需要知道 CPU 的流水线结构，操作系统开发者不需要知道 DRAM 的刷新时序——每一层的抽象都在为上层的开发者屏蔽复杂性。这正是第一讲所讨论的 DRY 哲学在系统架构层面的体现：*将共性需求下沉到平台层面统一解决，使得每一个应用都不必重复实现这些基础功能。*

2.1.2　平台的历史演进：从裸机到层叠的抽象

理解平台的分层结构为什么长成今天这个样子，最好的方法是回到历史的起点。

1946 年，约翰 • 冯 • 诺伊曼（John von Neumann）等人在宾夕法尼亚大学完成了 ENIAC 的设计。ENIAC 重约 30 吨，占地 170 平方米，包含 17,468 根真空管——但它没有操作系统，没有编程语言，甚至没有存储程序的概念。“编程”这台机器的方式是手工连接电缆和设置开关，每更换一个计算任务都需要数天的重新布线。在这个时代，“平台”就是裸露的硬件本身。冯 • 诺伊曼随后提出的“存储程序”体系结构改变了一切：程序和数据都以二进制形式存储在同一个内存中，计算机可以自动依次执行内存中的指令。这一架构奠定了现代计算机的基本形态，至今未变。但即便有了存储程序，早期的程序员仍然需要用机器码或汇编语言编程——他们直接操作寄存器、内存地址和 I/O 端口，每一行代码都与具体硬件紧密耦合。

*这种编程方式的问题不在于它不能工作——它当然能工作。问题在于它*不可规模化。*当程序只有几百行时，一个聪明的工程师可以在脑子里跟踪每一个寄存器的状态；当程序扩大到数千行，这种方式就开始崩溃。人脑的工作记忆容量——上一讲提到的米勒的 7 ± 2——成为了不可逾越的瓶颈。*解决的方法不是让人脑变得更强大，而是引入抽象层来降低认知负担。

1957 年，IBM 的约翰 • 巴科斯（John Backus）领导的团队发布了 FORTRAN 编译器——第一个被广泛使用的高级语言编译器。FORTRAN 允许工程师用接近数学公式的语法编写程序，编译器负责将其翻译为机器码。这是平台演进史上的第一次重大飞跃：**编程语言在程序员和硬件之间插入了一个抽象层。**

几乎在同一时期，另一种抽象正在酝酿——操作系统的雏形。1956 年，通用汽车公司和北美航空公司为 IBM 704 开发了 GM-NAA I/O 系统，它是最早的批处理操作系统之一，能够自动加载和执行一批作业，而不需要操作员为每个作业手工设置计算机。这是操作系统的萌芽：**在硬件和应用程序之间插入一个管理者。**

此后的六十年，平台的每一层都在不断演进和丰富：操作系统从简单的批处理发展到支持多任务、多用户、网络和图形界面的复杂系统；编程语言从 FORTRAN 和 COBOL 发展到 C、C++、Java、Python、Rust；库从简单的数学函数集合发展到庞大的框架生

态。但底层逻辑始终不变：**每一层的存在，都是为了让上层的开发者不必重复解决已经解决过的问题。**

2.1.3 EDA 与 Unix：一段共生的历史

为什么全球几乎所有的 EDA 工具都运行在 Linux 上？要回答这个问题，必须追溯到 1969 年的贝尔实验室。

那一年，肯·汤普森（Ken Thompson）和丹尼斯·里奇（Dennis Ritchie）在一台闲置的 PDP-7 小型计算机上开始开发一个新的操作系统。他们的动机很朴素——汤普森想在这台机器上运行他编写的一款名为“太空旅行”的游戏。这个项目最初只有两个人，目标也很有限，但它后来改变了整个计算机产业。布莱恩·柯尼汉（Brian Kernighan）给它起了个名字叫 Unix——这是对他们之前参与的大型操作系统项目 Multics 的戏谑，Multics 的“multi”变成了“uni”，寓意简单。

Unix 的设计哲学可以用三句话概括：每个程序只做一件事并做好它；程序之间通过文本流互相通信；一切皆文件。这种小工具组合的哲学看似简单，却蕴含着深刻的工程智慧——它使得复杂的任务可以通过组合简单工具来完成，而不必为每个新任务开发一个庞大的专用程序。

1973 年，里奇和汤普森用 C 语言重写了 Unix。这是一个具有历史意义的决策——Unix 成为第一个用高级语言而非汇编语言编写的操作系统。这意味着 Unix 可以相对容易地移植到不同的硬件架构上，而此前每种计算机都需要从头开发自己的操作系统。

这一决策与 EDA 的发展直接相关。1970 年代，加州大学伯克利分校同时是 Unix 和 EDA 的重镇：一方面，比尔·乔伊（Bill Joy）等人在伯克利开发了 BSD Unix，成为 Unix 最有影响力的学术分支；另一方面，拉里·内格尔和唐纳德·彼德森在同一所大学开发了 SPICE。SPICE 最初就运行在伯克利的 Unix 系统上。EDA 与 Unix 从诞生之日起就共享同一个学术生态——同样的大学、同样的计算机、同样的操作系统。这不是巧合，而是共生。

1980 年代，Unix 成为工程计算的事实标准。太阳微系统（Sun Microsystems）的 SPARC 工作站运行 Solaris——一种 Unix 变体——几乎占据了 EDA 用户的整个桌面。Synopsys、Cadence、Mentor Graphics 三大 EDA 公司的产品都以 Solaris 为首选平台。工程师们在 Sun 工作站上编写 Tcl 脚本，运行综合和布局布线工具，查看版图——这一景象持续了将近二十年。

然而，1980 年代也是 Unix 世界陷入分裂的年代。AT&T 收回了 Unix 的商业权利，推出 System V；伯克利继续发展 BSD；IBM 有 AIX，HP 有 HP-UX，Sun 有 SunOS 后来演变为 Solaris——每家公司都在自己的 Unix 变体上添加私有扩展，导致软件在不同 Unix 之间移植困难重重。这场被称为“Unix 战争”的混乱直接催生了 POSIX 标准——IEEE 在 1988 年发布的一组操作系统接口规范，试图在各家 Unix 之间建立一个公

共的 API 基线。POSIX 的意义在于它确立了一个原则：**接口可以标准化，即便实现各不相同**。这一思想对 EDA 工具的跨平台移植具有直接影响——EDA 公司只需要针对 POSIX 接口编程，就可以相对容易地在不同 Unix 系统之间移植工具。

Unix 战争的另一个深远后果，是为 Linux 的诞生创造了条件。1991 年 8 月 25 日，芬兰赫尔辛基大学的二年级学生林纳斯·托瓦兹（Linus Torvalds）在 Usenet 的 comp.os.minix 新闻组上发布了一条后来载入史册的消息："我正在做一个（免费的）操作系统（只是一个爱好，不会像 GNU 那样大而专业），面向 386(486) AT 兼容机。"这个"爱好项目"就是 Linux。托瓦兹最初的动机很朴素——他买不起运行 Unix 的工作站，但想在自己的 386 PC 上使用类 Unix 的操作系统。当时已有理查德·斯托曼（Richard Stallman）的 GNU 项目提供了大量 Unix 兼容的用户空间工具——编译器 GCC、编辑器 Emacs、Shell 工具等——但缺少一个内核。托瓦兹填补了这最后一块拼图。

Linux 的成功有多重原因，但最关键的一个是时机。1990 年代初，个人计算机的性能已经足够运行一个完整的 Unix 系统，但商业 Unix 的价格对个人和小型机构来说仍然高不可攀。Linux 提供了一个免费的、开源的替代方案——而且由于全球志愿者社区的参与，它的进化速度远超任何一家公司能够单独驱动的速度。到 1990 年代末，Linux 已经在服务器市场站稳脚跟；到 2000 年代，它开始进入企业级应用领域。

转折发生在 2000 年代初。Linux 的成熟和 x86 服务器的性价比优势使得越来越多的设计团队开始从昂贵的 SPARC 工作站迁移到廉价的 Linux PC。2002 年，Red Hat Enterprise Linux 发布，提供了企业级的稳定性和支持。EDA 公司相继将产品移植到 Linux x86 平台。到 2010 年代，这一迁移已基本完成——如今你走进全球任何一家芯片设计公司的机房，看到的几乎清一色是运行 Red Hat 或 CentOS 的 x86 服务器。

*EDA 行业从 Unix 到 Linux 的迁移，是平台演进的一个缩影。迁移的根本驱动力不是技术上的优越性——Linux 在功能上与商业 Unix 大同小异——而是**经济性**和**开放性**。一台 Sun SPARC 工作站的价格可以买十台甚至二十台同等算力的 Linux PC；而 Linux 的开源特性使得 EDA 公司可以更深入地了解和定制操作系统的行为，解决性能瓶颈。这提醒我们，软件设计平台的选择不仅是技术决策，更是经济决策。*

2.2　操作系统：软件世界的基础设施

2.2.1　操作系统的角色与使命

在平台的分层结构中，操作系统占据着一个独特的位置：它是硬件之上的第一层软件，是所有应用程序的共同地基。没有操作系统的计算机，用户必须直接面对硬件资源和软件资源，手动管理每一次内存分配、每一次磁盘读写、每一次设备中断——这就像在一座没有市政管理的城市里生活，每个居民都要自己修路、自己发电、自己处理垃圾。有了操作系统，这些公共事务被统一管理，居民只需要关心自己的日常工作。

操作系统承担三项核心使命。第一是**提供接口**：它在用户与硬件之间搭建了一座桥梁，使得用户不必了解硬件的物理细节就能使用计算机。第二是**资源管理**：CPU、内存、磁盘、网络——这些硬件资源都是有限的，操作系统负责在多个程序之间公平、高效地分配这些资源。第三是**流程组织**：它提供合理的任务调度和执行机制，使得计算机系统整体上高效运转。

想象一下没有操作系统的世界：每个应用程序都必须自己管理内存分配，自己控制磁盘读写，自己处理键盘和屏幕——这些基础设施代码可能占到总代码量的 80% 以上，而真正实现应用逻辑的部分不到 20%。操作系统的存在，使得应用开发者可以站在一个更高的起点上——这又是 DRY 原则的体现：**把所有应用都需要的公共服务提取出来，由操作系统统一提供。**

2.2.2 操作系统的演进：从批处理到现代

操作系统的发展并非一蹴而就，而是经历了数十年的渐进演化，每一代都是对前一代所暴露问题的回应。

最早的操作系统是 1950 年代末出现的**批处理系统**。在批处理模式下，操作员将一批作业——通常以穿孔卡片的形式——提交给计算机，系统自动依次执行每个作业，完成后输出结果。用户无法与正在运行的程序交互——提交作业后只能等待。批处理提高了计算机的利用率——机器不再需要在两个作业之间空等操作员手工设置——但用户体验极差：一个程序如果有一个小错误，你可能要等上一天才能拿到错误信息，修改后再等一天运行第二次。

1960 年代，**分时系统**的出现彻底改变了人与计算机的交互方式。分时系统允许多个用户通过终端同时使用一台计算机，操作系统将 CPU 时间切分成极短的时间片，轮流分配给每个用户的程序。由于切换速度极快——通常在毫秒级——每个用户都感觉自己独占了一台计算机。1964 年，MIT、贝尔实验室和通用电气公司联合启动了 Multics 项目，目标是创建一个能支持数百用户同时在线的分时操作系统。Multics 在技术上雄心勃勃，引入了层次化文件系统、虚拟内存、安全环等至今仍在使用的概念。但项目的复杂度远超预期——开发进度一再延误，最终贝尔实验室退出了 Multics 项目。汤普森和里奇正是在离开 Multics 之后，才在那台 PDP-7 上开发出了 Unix。历史有时就是这样运作的：一个失败的大项目孕育了一个成功的小项目。

1980 年代，个人计算机的普及催生了**桌面操作系统**的繁荣——MS-DOS、Mac OS、Windows 相继出现。1990 年代至今，**网络化**和**移动化**成为操作系统演进的主旋律：Linux 在服务器端占据主导，Android 和 iOS 统治了移动设备，而云计算则催生了容器化和微服务架构下的新一代操作系统理念。

回顾操作系统六十多年的演进史，一个清晰的主线浮现出来：操作系统的每一次进化，都是在更大规模上实现**资源共享**和**复杂性隐藏**。批处理系统实现了作业之间的自动

切换，分时系统实现了用户之间的 *CPU* 共享，网络操作系统实现了机器之间的资源共享，云操作系统实现了数据中心之间的资源共享。抽象的层次越来越高，用户离硬件越来越远——但能做的事情也越来越多。

2.2.3　操作系统的四个基本特征

现代操作系统有四个基本特征：并发、共享、虚拟和异步。这四个特征不是独立存在的——它们相互依存、相互制约，共同定义了操作系统的行为模型。

并发与并行

在操作系统的语境下，有两个容易混淆的概念需要严格区分。**并行**（Parallelism）指的是在**同一时刻**有多个任务同时执行，这需要多个物理处理器或多核 CPU 的支持——就像多条高速公路上的车辆同时行驶。**并发**（Concurrency）指的是在**一段时间内**有多个任务交替执行，它们在宏观上看起来像是同时进行的，但在微观上其实是快速切换——就像一个厨师在几道菜之间快速切换注意力，看起来像在同时做几道菜，实际上任何时刻只在处理一道。

现代操作系统的并发能力使得一台计算机可以同时运行数十甚至数百个进程。这对 EDA 工具尤其重要：一个芯片设计工程师的工作站上通常同时运行着综合工具、波形查看器、文本编辑器、邮件客户端、版本控制系统——操作系统的进程调度器负责让这些程序和谐共存。

但并发也带来了一类极其隐蔽和危险的软件缺陷——竞态条件（race condition）。当多个进程或线程同时访问共享资源，且最终结果取决于它们的执行顺序时，就可能发生竞态条件。这类 bug 的可怕之处在于它们通常不是每次都出现——它们取决于操作系统调度器的具体行为，而调度器的行为受到系统负载、中断时机等随机因素的影响。一个程序可能在开发者的机器上运行一万次都正确，但在客户的机器上每运行一千次就崩溃一次。

Therac-25 事件是竞态条件导致灾难的经典案例。*Therac-25* 是加拿大原子能有限公司（*AECL*）在 *1982* 年推出的一款医用放射治疗设备。它有两种工作模式：低能量电子束直接照射模式和高能量 *X* 射线模式。在 *X* 射线模式下，一块金属靶板会插入束流路径，将高能电子束转换为 *X* 射线。*1985* 年至 *1987* 年间，至少有六名患者在 *Therac-25* 的治疗中受到严重的辐射过量，其中至少三人死亡。事故调查发现，问题的根源是控制软件中的一个竞态条件：当操作员在极短时间内修改治疗参数时，软件可能在靶板尚未到位的情况下就发出高能量电子束——相当于用原本应该被转换为 *X* 射线的全功率电子束直接照射患者。更令人震惊的是，*Therac-25* 的前代产品 *Therac-20* 有硬件安全联锁装置，可以在靶板未到位时物理阻断束流；但 *Therac-25* 的设计者认为软件控制已经足够可靠，移除了这些硬件联锁。事故的教训是多重的：并发编程的复杂性不可低估，软

件不能替代硬件安全联锁，防御性设计必须是多层次的。从 *EDA* 的角度看，*Therac-25* 提醒我们：*EDA* 工具的输出——芯片设计——最终可能进入安全关键系统，工具本身的正确性直接关系到最终产品的安全性。

共享

共享（Sharing）是指计算机系统中的软硬件资源可供多个用户或多个进程共同使用。操作系统必须提供两种不同形式的共享机制。**互斥共享**（Mutual Exclusion）要求在同一时间内只允许一个进程使用某一资源——就像一间只有一把钥匙的会议室，一个人使用时其他人必须等待。**同时共享**（Concurrent Access）则允许多个进程在一段时间内同时访问某一资源——就像一座图书馆，多个读者可以同时在里面阅读不同的书。

并发和共享是操作系统最基本的两个特征，它们互为条件：没有并发，就不存在多个进程竞争共享资源的问题；没有共享机制，多个并发进程就无法协调地使用系统资源。

EDA 行业有一个对“共享”体会极为深刻的场景：**软件许可管理**。商业 EDA 工具使用 FlexNet（前身为 FLEXlm）等许可管理系统来控制软件的并发使用数量。一家芯片设计公司可能购买了 10 个 Design Compiler 的浮动许可——这意味着同一时间最多有 10 名工程师可以运行这款工具。当第 11 个人试图启动时，他会看到一条令人沮丧的消息：“No license available”。许可管理系统本质上是一个**应用层面的资源管理器**，它面临的挑战——如何公平分配有限资源、如何处理等待和超时、如何防止死锁——与操作系统的资源管理问题如出一辙。

虚拟化

虚拟化（Virtualization）是操作系统最富有想象力的设计思想之一。它的核心理念是：**用软件创造出实际并不存在的“虚拟”资源。**

虚拟化有两种基本策略。**时分复用**（Time Sharing）是将一个物理资源的使用时间分割成若干时间段，让多个逻辑实体轮流使用——最典型的例子是 CPU 虚拟化：一个物理 CPU 通过快速切换，让每个进程都“以为”自己独占了一个 CPU。**空间复用**（Space Sharing）是将一个物理资源的空间分割或聚合，形成多个逻辑实体——最典型的例子是虚拟内存：操作系统让每个进程都“以为”自己拥有一大片连续的内存空间，而实际上这些虚拟内存可能分散在物理内存的不同位置，甚至有一部分被临时存放在磁盘上。

虚拟内存对 EDA 工具具有特殊的重要性。考虑一个现代 SoC 的物理设计：一个包含数十亿个晶体管的设计，其网表、约束、时序信息等数据结构在内存中的占用量可能达到数百 GB。但工程师的工作站或服务器可能只有 64GB 或 128GB 的物理内存。虚拟内存机制使得 EDA 工具可以使用远超物理内存的地址空间——操作系统会将暂时不活跃的数据页面交换到磁盘上，需要时再换回来。当然，频繁的内存交换会严重降低性能——这就是为什么 EDA 服务器通常配备尽可能多的物理内存。一台专业的 EDA 服

务器配备 512GB 甚至 1TB 的内存并不罕见。

虚拟化的哲学启示在于：它展示了抽象的力量。物理世界的资源是有限的、不连续的、异构的；虚拟化在其上创造了一个理想化的世界——无限的、连续的、统一的。这种理想世界当然不是免费的——它需要操作系统在背后做大量的翻译和管理工作——但它极大地简化了上层软件的设计。每个程序都可以假设自己独享整台计算机的资源，而不必关心其他程序的存在。这种对复杂性的封装和隐藏，是计算机科学最核心的思维方式之一。

异步性

异步性（Asynchronism）是并发环境的必然结果。当操作系统同时管理多个进程时，由于资源争用和调度策略的影响，每个进程的执行过程并非“一气呵成”，而是“走走停停”——何时执行、何时暂停、以怎样的速度推进，都是不可预知的。

异步性增加了系统行为的不确定性，但操作系统必须保证一个关键性质：**只要运行环境相同，程序经过多次运行，都会获得完全相同的结果**。换言之，尽管执行过程是不确定的，最终结果必须是确定的。这一要求看似理所当然，实际上要求操作系统在进程同步、资源分配等方面做出极为精细的设计。

1997 年的火星探路者号（Mars Pathfinder）任务提供了一个异步性导致系统故障的生动案例。探路者号的“旅居者”（Sojourner）火星车在登陆火星表面后不久就开始经历反复的系统重启。NASA 的工程师经过远程诊断，发现问题出在 VxWorks 实时操作系统的进程调度上——一种被称为**优先级反转**（Priority Inversion）的经典故障。具体来说，一个低优先级的气象数据采集任务持有一把互斥锁，而高优先级的总线管理任务需要获取这把锁才能继续工作；同时，一个中等优先级的通信任务抢占了低优先级任务的 CPU 时间，导致低优先级任务无法运行，也就无法释放锁——于是高优先级任务被间接地无限期阻塞。系统的看门狗定时器检测到高优先级任务超时，于是触发了系统重启。

这个故障在地面测试中从未出现——因为地面测试的数据负载比实际火星环境小得多，三个任务之间的时序冲突极为罕见。而在火星上，更高的数据采集频率使得冲突概率大幅增加。*NASA* 最终通过远程上传补丁，启用了 *VxWorks* 内置的“优先级继承”（*Priority Inheritance*）协议来修复这个问题——当高优先级任务等待低优先级任务持有的锁时，暂时将低优先级任务的优先级提升到与高优先级任务相同，从而避免被中等优先级任务抢占。这个案例完美地说明了异步性的危险：在并发环境中，即便每个单独的组件都是正确的，它们的组合行为也可能在特定时序条件下出错——这与第一讲中阿丽亚娜 *5* 号事故所揭示的“系统思维的缺失”一脉相承。

2.2.4 操作系统的核心功能

操作系统的核心功能可以概括为六大模块：进程管理、存储管理、设备管理、文件管理、用户界面和网络管理。这些模块并非孤立存在——它们通过系统调用（System Call）向上层应用提供服务，通过硬件抽象层向下与硬件交互，构成一个有机的整体。以 Linux 内核为例，其架构从上到下包括：用户空间的应用程序通过系统调用接口进入内核空间，内核中的进程管理、内存管理、文件系统、设备驱动和网络子系统各司其职，最终通过硬件支持层与 CPU、内存、存储设备、终端设备和网络适配器通信。

进程管理

进程（Process）是操作系统中最核心的抽象概念之一。它的正式定义是：**可并发执行的程序在一个数据集合上的运行过程**。注意“程序”和“进程”的区别——程序是静态的指令序列，存储在磁盘上；进程是动态的执行实例，存在于内存中。同一个程序可以产生多个进程，就像同一张乐谱可以被不同的乐团同时演奏。

进程管理包含四项核心任务。**进程控制**负责进程的创建和撤销、资源的分配和回收。一个进程在其生命周期中会经历多种状态：就绪态——已具备运行条件，等待 CPU 分配；执行态——正在占用 CPU 运行；封锁态——因等待某一事件而暂停运行。这三种状态之间的转换构成了进程状态机——就绪态的进程被调度器选中后进入执行态，执行态的进程因等待 I/O 或其他资源而进入封锁态，封锁态的进程在等待的事件完成后回到就绪态。**进程同步**为多个并发进程提供协调机制，确保它们不会因为竞争共享资源而产生错误。**进程通信**使得协作的进程之间可以交换信息。**进程调度**从就绪队列中选择下一个要执行的进程，分配 CPU 资源——调度算法的选择直接影响系统的响应速度和吞吐量。

与进程密切相关的一个概念是**线程**（Thread）。线程是进程的下属实体，是操作系统进行 CPU 调度的基本单位。进程有两个基本属性：它是拥有计算资源的独立单位，也是被操作系统独立调度和分派计算资源的基本单位。线程则不再拥有系统资源，与同属一个进程的其他线程共享计算资源。一个线程可以创建和撤销另一个线程；同一进程中的多个线程之间可以并发执行。线程之间的切换开销远小于进程之间的切换，因为不需要切换内存映射——这使得多线程编程成为提高程序并行度的主要手段。

在 EDA 领域，多线程已经成为提高工具性能的关键技术。以静态时序分析为例：一个大规模设计可能包含数百万条时序路径，如果用单线程逐一分析，可能需要数小时。现代 STA 工具会将这些路径分配到多个线程上并行分析——一个 32 核的服务器可以将分析时间缩短到单线程的十分之一甚至更少。但多线程 STA 的实现远非简单地“把路径平均分配给各线程”那么直接——不同路径之间可能共享公共子路径，公共时钟路径的延迟计算结果需要在线程之间同步，负载均衡也需要精心设计。这些挑战正是操作系统进程管理理论在应用层面的直接体现。

进程调度算法的选择直接影响系统的行为特性。最简单的调度策略是**先来先服务**（First Come First Served，FCFS）——按照作业到达的顺序依次执行，实现简单但可能导致短作业被长作业阻塞。**最短作业优先**（Shortest Job First，SJF）总是选择预计执行时间最短的作业优先执行，可以最小化平均等待时间，但需要预知作业的执行时间——这在实践中往往是不可能的。**时间片轮转**（Round Robin）将 CPU 时间均匀切分，每个进程轮流获得一个时间片，时间片用完就切换到下一个进程——这是分时系统的基础，保证了响应时间的公平性。**优先级调度**为每个进程分配一个优先级，高优先级的进程优先获得 CPU——实时操作系统通常采用这种策略，以保证关键任务的及时响应。

对 *EDA* 工具而言，进程调度策略的影响不容忽视。当一台服务器上同时运行多个 *EDA* 作业时——比如三个综合任务和两个布局布线任务——操作系统的调度器决定了每个任务获得多少 *CPU* 时间。如果调度器不了解这些任务的优先级关系，可能会让一个已经接近 *deadline* 的流片任务和一个例行回归测试分到同样的 *CPU* 时间。许多 *EDA* 团队通过 *Linux* 的 *cgroup* 机制和任务队列系统（如 *LSF*、*SGE*）来精细控制作业调度，本质上是在操作系统的通用调度器之上，叠加了一层领域特定的调度策略。

存储管理

存储管理是操作系统的又一核心功能，其主要职责包括**存储分配**、**内存保护**和**地址映射**。

存储分配确保每个进程在运行时有足够的内存空间来存放其代码和数据。操作系统在进程创建时为其分配内存，并在进程终止时回收。内存保护确保每个进程只能访问属于自己的内存区域——当一个进程试图访问不属于自己的地址时，操作系统会立即终止该进程并报告错误。这一机制至关重要：没有内存保护，一个有 bug 的程序可能覆盖其他程序甚至操作系统的内存，导致整个系统崩溃。

地址映射是虚拟内存的核心实现机制。应用程序使用的内存地址是“虚拟地址”——一个连续的、从零开始的地址空间。应用程序认为它拥有连续可用的内存，而实际上它通常被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。操作系统负责将虚拟地址翻译为实际的物理地址，这种翻译通过页表（Page Table）实现：内存被分成固定大小的“页”——通常 4KB——操作系统维护一张表，记录每一个虚拟页对应的物理页位置。当一个虚拟页当前不在物理内存中而在磁盘上时，访问它会触发“缺页中断”，操作系统将该页从磁盘调入内存，如有必要则将另一个不活跃的页换出到磁盘。

对 *EDA* 工具而言，内存管理几乎是性能的决定性因素。一个先进工艺节点的大规模 *SoC* 设计，其物理设计数据在内存中的峰值占用可能达到 *200GB* 甚至更多。如果物理内存不足，操作系统会频繁地将数据页面换入换出磁盘——即便使用 *SSD*，磁盘 *I/O* 的速度也比内存访问慢两到三个数量级。一旦发生大规模的页面交换，*EDA* 工具的运行

时间可能从几小时膨胀到几天甚至数周，实际上等于无法完成。这就是为什么 *EDA* 服务器的内存配置通常远高于普通服务器——一台配备 *1TB* 内存的服务器在 *EDA* 行业并不少见，甚至有些设计团队使用 *2TB* 内存的高端服务器来处理最大规模的设计。在 *EDA* 工程师的日常语言中，“*run out of memory*”是仅次于“*timing not met*”的噩梦。

设备管理与文件系统

设备管理负责协调计算机系统中的各种 I/O 设备——磁盘、显示器、键盘、网络接口等。操作系统通过**设备驱动程序**（Device Driver）来屏蔽不同硬件设备的差异，向上层提供统一的访问接口。设备管理的核心任务包括：设计分配策略，把相应的设备分配给提出请求的进程，并启动设备完成数据的传递和 I/O 操作；根据设备的性能和作用对设备分类，设计相应的驱动程序；为用户提供使用设备的界面和访问方法。驱动程序是操作系统中最容易出问题的部分——据统计，Linux 内核中超过 70% 的代码是设备驱动，Windows 系统中大部分蓝屏死机也是由有缺陷的驱动程序引起的。

文件系统是操作系统提供的另一项基础服务。它在磁盘的原始数据块之上建立了一个层次化的目录结构，使得用户可以用人类可读的名称——而非物理扇区号——来组织和访问数据。文件系统的核心功能包括：建立文件的存储检索和目录结构，管理文件的空间分配，建立文件的共享和保护机制，以及向用户提供操作文件的界面。Unix 和 Linux 的文件系统遵循一个统一的层次化结构，从根目录“/”开始展开。这一结构反映了 Unix“一切皆文件”的哲学——不仅普通数据是文件，设备也是文件，进程信息也是文件，系统配置也是文件。这种统一的抽象极大地简化了系统编程——读取一个文件和读取一个设备使用的是同一套系统调用。

用户界面：从命令行到 API

操作系统向用户提供三种层次的交互界面。**命令行界面**（Command Line Interface，CLI）是最古老也是最强大的交互方式——用户通过键入文本命令来控制计算机。CLI 的工作流程简洁而优雅：显示提示符，等待用户输入命令，解析并执行命令，显示结果，循环往复。CLI 的优势在于精确、可脚本化、可自动化——一条精心编写的 Shell 命令可以完成 GUI 中需要点击数十次的操作。

图形用户界面（Graphic User Interface，GUI）通过窗口、图标、菜单和指点设备提供直观的交互体验。GUI 降低了使用门槛——用户不需要记忆命令和参数——但在自动化和批处理方面不如 CLI 灵活。**应用程序接口**（Application Programming Interface，API）则是软件与软件之间的交互界面。操作系统通过系统调用 API 向应用程序提供服务；应用程序之间也可以通过 API 相互调用。

EDA 工具在用户界面的设计上经历了一段有趣的演变。早期的 *EDA* 工具完全是命令行驱动的——工程师编写脚本，工具在后台运行，产生报告文件。*1990* 年代，*GUI* 开

始流行，各家 *EDA* 公司为自己的工具开发了图形界面——版图编辑器有可视化的布局视图，波形查看器有交互式的信号显示。但 *EDA* 行业很快发现了一个有趣的现象：即便有了漂亮的 *GUI*，高效的工程师仍然倾向于使用命令行和脚本。原因在于芯片设计流程的重复性——一次成功的综合可能需要数十次迭代，每次迭代都需要微调参数。用 *GUI* 手动操作数十次不仅效率低下，而且容易出错；而一个参数化的 *Tcl* 脚本可以在几秒钟内启动一次新的迭代。这就是为什么几乎所有商业 *EDA* 工具都支持 *Tcl* 脚本——*GUI* 适合探索和学习，*CLI* 适合生产和自动化。两者不是替代关系，而是互补关系。

2.3　从单机到云端：计算架构的变迁

2.3.1　计算机网络：连接万物的基础设施

当芯片设计的规模超越了单台计算机的处理能力时，多台计算机的协作就成为必然。计算机网络使这种协作成为可能——它利用通信设备和线路将地理位置不同、功能独立的多个计算机系统连接起来，实现硬件、软件及资源的共享和信息的传递。

网络的**拓扑结构**——即网络中各设备相互连接的排列方式——可以是物理的即真实的结构，也可以是逻辑的即虚拟的结构。常见的拓扑包括环形、网状、星形、全连接、线形、树形和总线等。每种拓扑都有其适用场景和权衡：星形拓扑管理简单但中心节点是单点故障，网状拓扑冗余度高但成本也高，总线拓扑经济但带宽共享。

按地理覆盖范围，网络从小到大可分为个人网（Personal Area Network，PAN）、局域网（Local Area Network，LAN）、城域网（Metropolitan Area Network，MAN）、广域网（Wide Area Network，WAN）和互联网（Internet）。EDA 工程师日常使用的主要是局域网——设计团队的工作站和计算服务器通常通过高速局域网互联，共享文件服务器上的设计数据和网络许可服务器上的 EDA 许可。互联网的物质基础包括有线互连——双绞线、同轴电缆、光纤——和无线互连——微波、卫星、移动通信网、无线局域网、蓝牙等。

计算机网络的起源本身就是一个精彩的故事。1969 年 10 月 29 日，加州大学洛杉矶分校（UCLA）的研究生查利 • 克莱恩（Charley Kline）试图通过 ARPANET——美国国防部高级研究计划局资助的实验性网络——向斯坦福研究所发送第一条信息。他打算输入“LOGIN”，但只发出了“L”和“O”两个字母后系统就崩溃了。于是人类历史上通过计算机网络发送的第一条信息是“LO”。克莱恩后来说，这倒像是“Lo and behold”（你瞧！）的缩写——仿佛在宣告一个新时代的到来。ARPANET 最初只连接了四个节点：UCLA、斯坦福研究所、加州大学圣巴巴拉分校和犹他大学。从这四个节点开始，网络逐步扩展，最终演化为今天连接数十亿设备的互联网。

网络中计算机之间消息传递的规则和格式由**网络协议**定义。网络协议是计算机通信的共同语言，是一个分层的结构，高层协议建立在低层协议之上。Internet 使用的

TCP/IP 协议分为五个层次：物理层定义网络传输介质及其硬件接口的电气特性规范，以便建立、维持和拆除物理连接；网络接口层即链路层提供相邻结点间信息传输的规范；互联网层提供基本的数据封包传送功能，让每一块数据包都能够到达目的主机；传输层提供节点间的数据传送和应用程序之间的通信服务，主要功能是数据格式化、数据确认和丢失重传等；应用层提供应用程序间的通信规范，如简单电子邮件传输协议 SMTP、文件传输协议 FTP 等。这种分层设计的思想与软件设计平台的分层哲学完全一致——每一层解决一类特定的问题，向上提供明确的服务接口，向下依赖相邻层的服务。

2.3.2 分布式计算的演进

计算架构的演进史可以概括为三个阶段，每个阶段都反映了当时技术条件下对计算资源组织方式的最优选择。

第一阶段是**主机/终端模式**。所有计算在中央主机上完成，终端只是一个“哑”设备，提供简单的输入和输出功能。这种模式盛行于大型机时代——用户坐在终端前，通过串行线路连接到机房中的大型计算机。计算能力完全集中，管理简单，但灵活性极低：主机一旦宕机，所有用户都无法工作。

第二阶段是**客户机/服务器模式**。随着个人计算机的普及，应用程序开始运行在用户自己的计算机——“客户机”上，而服务器提供共享资源的访问服务，通常是文件或数据库。客户机承担大部分计算工作，因此被称为“胖客户”。这种模式在 1990 年代到 2000 年代是主流架构。EDA 行业在这一时期的典型配置是：工程师在自己的工作站上运行 EDA 工具，设计文件存储在 NFS 文件服务器上，许可通过网络许可服务器管理。

第三阶段是**云计算模式**。云计算将计算任务分布在大量计算机构成的资源池上，使各种应用系统能够根据需要获取计算力、存储空间和各种软件服务。它的核心理念是将计算、服务和应用作为一种公共设施提供给公众——就像电力一样，用户不需要自建发电站，只需要从电网中按需取电并按量付费。

2.3.3 云计算的三种服务模式

云计算按照服务的抽象层次分为三种模式。

基础架构即服务（Infrastructure as a Service，IaaS）是最底层的云服务。它是一种托管型硬件方式，用户付费使用厂商的硬件设施。厂商将内存、I/O 设备、存储和计算能力整合成一个虚拟的资源池，为客户提供所需要的存储资源和虚拟化服务器等服务。用户获得的是虚拟硬件——可以在上面安装任何操作系统和软件，拥有最大的自由度，但也需要自己管理操作系统和应用。

平台即服务（Platform as a Service，PaaS）把开发环境作为一种服务来提供。厂商提供开发环境、服务器平台、硬件资源等给客户，用户在其平台基础上定制开发自己的应用程序并通过其服务器和互联网传递给其他客户。用户不必关心底层的服务器配置和

维护，可以专注于应用开发。

软件即服务（Software as a Service，SaaS）是最上层的云服务。用户根据需求通过互联网向厂商订购应用软件服务，厂商根据客户所定软件的数量、时间的长短等因素收费。用户不需要安装和维护软件，通过浏览器或轻量级客户端即可使用。

表 2.1: 云计算三种服务模式的比较

	IaaS	**PaaS**	**SaaS**
用户管理范围	OS 及以上全部	应用程序和数据	仅使用配置
灵活性	最高	中等	最低
管理负担	最重	中等	最轻
典型场景	虚拟服务器	开发平台	在线办公
EDA 类比	租用计算集群跑 EDA	使用预配置的 EDA 环境	云端 EDA 工具

云计算正在深刻地改变 *EDA* 行业的运作方式。传统模式下，芯片设计公司需要自建大规模的计算基础设施——数百台高性能服务器、*PB* 级的存储系统、复杂的网络架构——这些基础设施的建设和维护成本高昂，而且利用率很不均匀：在项目的流片冲刺阶段，计算需求可能是平时的数十倍，但公司不可能为峰值需求常备那么多服务器。云计算提供了一种弹性的解决方案：在需求高峰期从云端临时获取额外的计算资源，高峰过后释放。*Synopsys*、*Cadence* 和 *Siemens EDA* 都已推出了云端 *EDA* 解决方案，与 *Amazon Web Services*、*Microsoft Azure* 和 *Google Cloud* 等云服务商建立了合作。但云 *EDA* 面临一个特殊的挑战：芯片设计数据涉及高度敏感的知识产权，将设计数据上传到第三方云服务器在安全性上面临巨大质疑，尤其是在当前复杂的国际形势下。这一矛盾——效率与安全之间的权衡——将在未来数年持续驱动 *EDA* 云化的演进方向。

2.3.4　YAGNI：平台设计的减法哲学

在讨论了平台从硬件到云端的层层架构之后，有必要引入一个看似简单却极为深刻的软件工程原则——YAGNI，即 You Aren't Gonna Need It，**面临确凿的需求时，才实现相应功能**。

这一原则来自极限编程（Extreme Programming）运动，由罗恩 • 杰弗里斯（Ron Jeffries）等人在 1990 年代末推广。其核心思想是：不要因为预见到将来**可能**需要某个功能就提前实现它——只在真正需要时才动手。

YAGNI 看似与“设计良好的平台”的理念矛盾——平台不就是为了未来的需求而提前准备的吗？但两者其实并不矛盾。**平台抽象的是已经确认的、反复出现的共性需求**——进程管理、内存管理、文件系统、网络通信——这些是几十年的实践证明所有应用都需要的基础服务。YAGNI 反对的是基于猜测的提前实现——“也许将来有用户需要

这个功能”和“已经有十个用户要求这个功能”是完全不同的决策依据。

在 EDA 工具的开发实践中，YAGNI 原则有着深刻的实际意义。EDA 工具的开发者经常面临这样的诱惑：客户 A 提出了一个定制需求，工程师在实现时想“不如把接口做得更通用一些，将来其他客户可能也需要类似的功能”。于是一个简单的定制功能膨胀成了一个通用框架，开发时间从两周变成两个月，引入的 bug 比解决的问题还多——而那些“将来的客户”可能永远不会出现，或者他们的需求与预想的完全不同。YAGNI 的智慧在于承认人类预测未来需求的能力是有限的，与其在猜测上浪费精力，不如把当下确定需要的东西做到极致，等真正的新需求出现时再扩展。这与第一讲讨论的 DRY 原则并不矛盾，而是互补——DRY 告诉我们“不要重复自己”，YAGNI 告诉我们“不要预支未来”。两者共同构成了务实软件工程的基石。

2.4 本讲总结

本讲从软件开发的基础设施——软件设计平台——的视角，讨论了三个层面的问题。

平台为什么是分层的？从裸机编程到操作系统、编程语言、库和框架，平台的每一层都是对下层复杂性的封装和抽象。分层的本质是关注点分离——每一层只需关心相邻层的接口，而不必了解更底层的实现。EDA 与 Unix 的共生历史表明，平台的选择不仅是技术决策，更是经济和生态的决策。

操作系统解决了什么问题？操作系统在硬件之上创造了一个有序的虚拟世界，通过并发、共享、虚拟化和异步四大机制，使得多个程序可以安全、高效地共享有限的硬件资源。操作系统的核心功能——进程管理、存储管理、设备管理、文件系统、用户界面——构成了所有应用软件的共同地基。Therac-25 和 Mars Pathfinder 的案例提醒我们：操作系统层面的设计缺陷可能导致致命后果。

计算架构如何演进？从主机/终端到客户机/服务器再到云计算，计算架构的演进反映了对计算资源组织方式的不断优化。云计算正在改变 EDA 行业的运作模式，但效率与安全之间的权衡仍是一个未解的挑战。

贯穿这三个层面的底层哲学仍然是 DRY——**不要重复自己**。操作系统将所有应用共需的资源管理提取为公共服务，库将反复使用的功能封装为可调用的模块，云平台将基础设施的建设和维护从个体企业转移到专业服务商——每一次提取和封装，都是 DRY 原则在更大尺度上的应用。而 YAGNI 原则为 DRY 划定了边界：**封装已确认的共性需求，但不预支未来的猜测。**

课后思考

1. EDA 工具几乎全部运行在 Linux 平台上。如果你要从头开发一款 EDA 工具，你会选择 Linux、Windows 还是 macOS 作为目标平台？为什么？如果需要支持多个平台，操

作系统的哪些差异会成为主要挑战？
2. 虚拟内存机制对 EDA 工具的性能至关重要。假设你的物理设计工具在处理一个大规模设计时，运行时间从预期的 8 小时突然变成了 80 小时——你如何判断是否发生了严重的内存页面交换？如果是，你会采取什么措施？
3. 云计算为 EDA 行业带来了弹性计算的可能性，但也引发了数据安全的担忧。你认为在当前的技术条件下，是否存在既能充分利用云计算的算力优势、又能保障芯片设计数据安全的方案？如果有，它的技术架构应该是什么样的？

参考文献

[1] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," *Communications of the ACM*, vol. 17, no. 7, pp. 365–375, 1974.

[2] Tanenbaum, A. S. and Bos, H., *Modern Operating Systems*, 4th ed., Pearson, 2014.

[3] Silberschatz, A., Galvin, P. B., and Gagne, G., *Operating System Concepts*, 10th ed., Wiley, 2018.

[4] Leveson, N. G. and Turner, C. S., "An Investigation of the Therac-25 Accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[5] Reeves, G. E., "What Really Happened on Mars?," *Risks Digest*, vol. 19, no. 49, 1997.

[6] Salus, P. H., *A Quarter Century of UNIX*, Addison-Wesley, 1994.

[7] Raymond, E. S., *The Art of Unix Programming*, Addison-Wesley, 2003.

[8] Torvalds, L. and Diamond, D., *Just for Fun: The Story of an Accidental Revolutionary*, HarperBusiness, 2001.

[9] Mell, P. and Grance, T., "The NIST Definition of Cloud Computing," NIST Special Publication 800-145, 2011.

[10] Jeffries, R., "You Aren't Gonna Need It," *XProgramming.com*, 1998.

[11] Kahng, A. B., Lienig, J., Markov, I. L., and Hu, J., *VLSI Physical Design: From Graph Partitioning to Timing Closure*, 2nd ed., Springer, 2022.

[12] Hunt, A. and Thomas, D., *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 1999.

[13] 赵文庆, 周学功, 《软件设计和开发》, 复旦大学出版社, 2013.

[14] 汤小丹, 梁红兵, 哲凤屏, 汤子瀛, 《计算机操作系统》, 第四版, 西安电子科技大学出版社, 2014.