运行，这意味着硬件限制了应用程序的功能。例如，以用户模式运行的应用程序通常不能发起对磁盘的 I/O 请求，不能访问任何物理内存页或在网络上发送数据包。在发起系统调用时 [通常通过一个称为陷阱（trap）的特殊硬件指令]，硬件将控制转移到预先指定的陷阱处理程序（trap handler）（即预先设置的操作系统），并同时将特权级别提升到内核模式（kernel mode）。在内核模式下，操作系统可以完全访问系统的硬件，因此可以执行诸如发起 I/O 请求或为程序提供更多内存等功能。当操作系统完成请求的服务时，它通过特殊的陷阱返回（return-from-trap）指令将控制权交还给用户，该指令返回到用户模式，同时将控制权交还给应用程序，回到应用离开的地方。

多道程序时代

操作系统的真正兴起在大主机计算时代之后，即小型机（minicomputer）时代。像数字设备公司（DEC）的 PDP 系列这样的经典机器，让计算机变得更加实惠。因此，不再是每个大型组织拥有一台主机，而是组织内的一小群人可能拥有自己的计算机。毫不奇怪，这种成本下降的主要影响之一是开发者活动的增加。更聪明的人接触到计算机，从而让计算机系统做出更有趣和漂亮的事情。

特别是，由于希望更好地利用机器资源，多道程序（multiprogramming）变得很普遍。操作系统不是一次只运行一项作业，而是将大量作业加载到内存中并在它们之间快速切换，从而提高 CPU 利用率。这种切换非常重要，因为 I/O 设备很慢。在处理 I/O 时让程序占着 CPU，浪费了 CPU 时间。那么，为什么不切换到另一份工作并运行一段时间？

在 I/O 进行和任务中断时，要支持多道程序和重叠运行。这一愿望迫使操作系统创新，沿着多个方向进行概念发展。内存保护（memory protection）等问题变得重要。我们不希望一个程序能够访问另一个程序的内存。了解如何处理多道程序引入的并发（concurrency）问题也很关键。在中断存在的情况下，确保操作系统正常运行是一个很大的挑战。我们将在本书后面研究这些问题和相关主题。

当时主要的实际进展之一是引入了 UNIX 操作系统，主要归功于贝尔实验室（电话公司）的 Ken Thompson 和 Dennis Ritchie。UNIX 从不同的操作系统获得了许多好的想法（特别是来自 Multics [O72]，还有一些来自 TENEX [B+72]和 Berkeley 分时系统[S+68]等系统），但让它们更简单易用。很快，这个团队就向世界各地的人们发送含有 UNIX 源代码的磁带，其中许多人随后参与并添加到系统中。请参阅补充了解更多细节[①]。

摩登时代

除了小型计算机之外，还有一种新型机器，便宜，速度更快，而且适用于大众：今天我们称之为个人计算机（Personal Computer，PC）。在苹果公司早期的机器（如 Apple II）和 IBM PC 的引领下，这种新机器很快就成为计算的主导力量，因为它们的低成本让每个桌子上都有一台机器，而不是每个工作小组共享一台小型机。

遗憾的是，对于操作系统来说，个人计算机起初代表了一次巨大的倒退，因为早期的系统忘

① 我们将使用补充和其他相关文本框，让你注意到不太适合文本主线的各种内容。有时候，我们甚至会用它们来开玩笑，为什么在这个过程中没有一点乐趣？是的，许多笑话都很糟糕。

记了（或从未知道）小型机时代的经验教训。例如，早期的操作系统，如 DOS（来自微软的磁盘操作系统），并不认为内存保护很重要。因此，恶意程序（或者只是一个编程不好的应用程序）可能会在整个内存中乱写乱七八糟的东西。第一代 macOS（V9 及更早版本）采取合作的方式进行作业调度。因此，意外陷入无限循环的线程可能会占用整个系统，从而导致重新启动。这一代系统中遗漏的操作系统功能造成的痛苦列表很长，太长了，因此无法在此进行全面的讨论。

幸运的是，经过一段时间的苦难后，小型计算机操作系统的老功能开始进入台式机。例如，macOS X 的核心是 UNIX，包括人们期望从这样一个成熟系统中获得的所有功能。Windows 在计算历史中同样采用了许多伟大的思想，特别是从 Windows NT 开始，这是微软操作系统技术的一次巨大飞跃。即使在今天的手机上运行的操作系统（如 Linux），也更像小型机在 20 世纪 70 年代运行的，而不像 20 世纪 80 年代 PC 运行的那种操作系统。很高兴看到在操作系统开发鼎盛时期出现的好想法已经进入现代世界。更好的是，这些想法不断发展，为用户和应用程序提供更多功能，让现代系统更加完善。

补充：UNIX 的重要性

在操作系统的历史中，UNIX 的重要性举足轻重。受早期系统（特别是 MIT 著名的 Multics 系统）的影响，UNIX 汇集了许多了不起的思想，创造了既简单又强大的系统。

最初的“贝尔实验室”UNIX 的基础是统一的原则，即构建小而强大的程序，这些程序可以连接在一起形成更大的工作流。在你输入命令的地方，shell 提供了诸如管道（pipe）之类的原语，来支持这样的元（meta-level）编程，因此很容易将程序串起来完成更大的任务。例如，要查找文本文件中包含单词“foo”的行，然后要计算存在多少行，请键入：grep foo file.txt | wc -l，从而使用 grep 和 wc （单词计数）程序来实现你的任务。

UNIX 环境对于程序员和开发人员都很友好，并为新的 C 编程语言提供了编译器。程序员很容易编写自己的程序并分享它们，这使得 UNIX 非常受欢迎。作为开放源码软件（open-source software）的早期形式，作者向所有请求的人免费提供副本，这可能帮助很大。

代码的可得性和可读性也非常重要。用 C 语言编写的美丽的小内核吸引其他人摆弄内核，添加新的、很酷的功能。例如，由 Bill Joy 领导的伯克利创业团队发布了一个非常棒的发行版（Berkeley Systems Distribution，BSD），该发行版拥有先进的虚拟内存、文件系统和网络子系统。Joy 后来与朋友共同创立了 Sun Microsystems。

遗憾的是，随着公司试图维护其所有权和利润，UNIX 的传播速度有所放慢，这是律师参与其中的不幸（但常见的）结果。许多公司都有自己的变种：Sun Microsystems 的 SunOS、IBM 的 AIX、HP 的 HPUX（又名 H-Pucks）以及 SGI 的 IRIX。AT&T/贝尔实验室和这些其他厂商之间的法律纠纷给 UNIX 带来了阴影，许多人想知道它是否能够存活下来，尤其是 Windows 推出后并占领了大部分 PC 市场……

补充：然后出现了 Linux

幸运的是，对于 UNIX 来说，一位名叫 Linus Torvalds 的年轻芬兰黑客决定编写他自己的 UNIX 版本，该版本严重依赖最初系统背后的原则和思想，但没有借用原来的代码集，从而避免了合法性问题。他征集了世界各地许多其他人的帮助，不久，Linux 就诞生了（同时也开启了现代开源软件运动）。

随着互联网时代的到来，大多数公司（如谷歌、亚马逊、Facebook 和其他公司）选择运行 Linux，因为它是免费的，可以随时修改以适应他们的需求。事实上，如果不存在这样一个系统，很难想象这些

新公司的成功。随着智能手机成为占主导地位的面向用户的平台，出于许多相同的原因，Linux 也在那里找到了用武之地（通过 Android）。史蒂夫·乔布斯将他的基于 UNIX 的 NeXTStep 操作环境带到了苹果公司，从而使得 UNIX 在台式机上非常流行（尽管很多苹果技术用户可能都不知道这一事实）。因此，UNIX 今天比以往任何时候都更加重要。如果你相信有计算之神，那么应该感谢这个美妙的结果。

2.7 小结

至此，我们介绍了操作系统。今天的操作系统使得系统相对易于使用，而且你今天使用的几乎所有操作系统都受到本章讨论的操作系统发展的影响。

由于篇幅的限制，我们在本书中将不会涉及操作系统的一些部分。例如，操作系统中有很多网络代码。我们建议你去上网络课以便更多地学习相关知识。同样，图形设备尤为重要。请参加图形课程以扩展你在这方面的知识。最后，一些操作系统书籍谈论了很多关于安全性的内容。我们会这样做，因为操作系统必须在正在运行的程序之间提供保护，并为用户提供保护文件的能力，但我们不会深入研究安全课程中可能遇到的更深层次的安全问题。

但是，我们将讨论许多重要的主题，包括 CPU 和内存虚拟化的基础知识、并发以及通过设备和文件系统的持久性。别担心！虽然有很多内容要介绍，但其中大部分都很酷。这段旅程结束时，你将会对计算机系统的真实工作方式有一个全新的认识。现在开始吧！

参考资料

[BS+09]“Tolerating File-System Mistakes with EnvyFS”
Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
USENIX ’09, San Diego, CA, June 2009
一篇有趣的文章，讲述同时使用多个文件系统以容忍其中任何一个文件系统出现错误。

[BH00]“The Evolution of Operating Systems”
P. Brinch Hansen
In Classic Operating Systems: From Batch Processing to Distributed Systems Springer-Verlag, New York, 2000
这篇文章介绍了与具有历史意义的系统相关的内容。

[B+72]“TENEX, A Paged Time Sharing System for the PDP-10”
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson CACM, Volume 15, Number 3, March 1972
TENEX 拥有现代操作系统中的许多机制。请阅读更多关于它的信息，看看在 20 世纪 70 年代早期已经有了哪些创新。

[B75]“The Mythical Man-Month”Fred Brooks
Addison-Wesley, 1975

一本关于软件工程的经典教科书，非常值得一读。

[BOH10]“Computer Systems: A Programmer’s Perspective”Randal E. Bryant and David R. O’Hallaron
Addison-Wesley, 2010
关于计算机系统工作原理的另一本卓越的图书，与本书的内容有一点点重叠——所以，如果你愿意，你可以跳过本书的最后几章，或者直接阅读它们，以获取关于某些相同材料的不同观点。毕竟，健全与完善自己知识的一个好方法，就是尽可能多地听取其他观点，然后在此问题上扩展自己的观点和想法。

[K+61]“One-Level Storage System”
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner IRE Transactions on Electronic Computers, April 1962
Atlas 开创了你在现代系统中看到的大部分概念。但是，这篇论文并不是最好的读物。如果你只读一篇文章，可以了解一下下面的历史观点[L78]。

[L78]“The Manchester Mark I and Atlas: A Historical Perspective”
S.H. Lavington
Communications of the ACM archive Volume 21, Issue 1 (January 1978), pages 4-12
关于计算机系统早期发展的历史和 Atlas 的开拓性工作。当然，我们可以自己阅读 Atlas 的论文，但是这篇论文提供了一个对计算机系统的很好的概述，并且增加了一些历史观点。

[O72]“The Multics System: An Examination of its Structure”Elliott Organick, 1972
Multics 的完美概述。这么多好的想法，但它是一个过度设计的系统，目标太多，因此从未真正按预期工作。*Fred Brooks* 是所谓的“第二系统效应”的典型例子[B75]。

[PP03]“Introduction to Computing Systems: From Bits and Gates to C and Beyond”
Yale N. Patt and Sanjay J. Patel
McGraw-Hill, 2003
我们最喜欢的计算系统图书之一。它从晶体管开始讲解，一直讲到 C。书中早期的素材特别好。

[RT74]“The UNIX Time-Sharing System”Dennis M. Ritchie and Ken Thompson
CACM, Volume 17, Number 7, July 1974, pages 365-375
关于 UNIX 的杰出总结，作者撰写此书时，UNIX 正在计算世界里占据统治地位。

[S68]“SDS 940 Time-Sharing System”Scientific Data Systems Inc.
TECHNICAL MANUAL, SDS 90 11168 August 1968
这是我们可以找到的一本不错的技术手册。阅读这些旧的系统文件，能看到在 20 世纪 60 年代后期技术发展的进程，这很有意思。伯克利时分系统（最终成为 SDS 系统）背后的核心构建者之一是 Butler Lampson，后来他因系统贡献而获得图灵奖。

[SS+10]“Membrane: Operating System Support for Restartable File Systems”Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift FAST ’10, San Jose, CA, February 2010
写自己的课程注解的好处是：你可以为自己的研究做广告。但是这篇论文实际上非常简洁。当文件系统遇到错误并崩溃时，Membrane 会自动重新启动它，所有这些都不会导致应用程序或系统的其他部分受到影响。

第 1 部分　虚拟化

第 3 章　关于虚拟化的对话

教授：现在我们开始讲操作系统 3 个部分的第 1 部分——虚拟化。

学生：尊敬的教授，什么是虚拟化？

教授：想象我们有一个桃子。

学生：桃子？（不可思议）

教授：是的，一个桃子，我们称之为物理（physical）桃子。但有很多想吃这个桃子的人，我们希望向每个想吃的人提供一个属于他的桃子，这样才能皆大欢喜。我们把给每个人的桃子称为虚拟（virtual）桃子。我们通过某种方式，从这个物理桃子创造出许多虚拟桃子。重要的是，在这种假象中，每个人看起来都有一个物理桃子，但实际上不是。

学生：所以每个人都不知道他在和别人一起分享一个桃子吗？

教授：是的。

学生：但不管怎么样，实际情况就是只有一个桃子啊。

教授：是的，所以呢？

学生：所以，如果我和别人分享同一个桃子，我一定会发现这个问题。

教授：是的！你说得没错。但吃的人多才有这样的问题。多数时间他们都在打盹或者做其他事情，所以，你可以在他们打盹的时候把他手中的桃子拿过来分给其他人，这样我们就创造了有许多虚拟桃子的假象，每人一个桃子！

学生：这听起来就像糟糕的竞选口号。教授，您是在跟我讲计算机知识吗？

教授：年轻人，看来需要给你一个更具体的例子。以最基本的计算机资源 CPU 为例，假设一个计算机只有一个 CPU（尽管现代计算机一般拥有 2 个、4 个或者更多 CPU），虚拟化要做的就是将这个 CPU 虚拟成多个虚拟 CPU 并分给每一个进程使用，因此，每个应用都以为自己在独占 CPU，但实际上只有一个 CPU。这样操作系统就创造了美丽的假象——它虚拟化了 CPU。

学生：听起来好神奇，能再多讲一些吗？它是如何工作的？

教授：问得很及时，听上去你已经做好开始学习的准备了。

学生：是的，不过我还真有点担心您又要讲桃子的事情了。

教授：不用担心，毕竟我也不喜欢吃桃子。那我们开始学习吧……

第 4 章　抽象：进程

本章讨论操作系统提供的基本的抽象——进程。进程的非正式定义非常简单：进程就是运行中的程序。程序本身是没有生命周期的，它只是存在磁盘上面的一些指令（也可能是一些静态数据）。是操作系统让这些字节运行起来，让程序发挥作用。

事实表明，人们常常希望同时运行多个程序。比如：在使用计算机或者笔记本的时候，我们会同时运行浏览器、邮件、游戏、音乐播放器，等等。实际上，一个正常的系统可能会有上百个进程同时在运行。如果能实现这样的系统，人们就不需要考虑这个时候哪一个 CPU 是可用的，使用起来非常简单。因此我们的挑战是:

关键问题：如何提供有许多 CPU 的假象？

虽然只有少量的物理 CPU 可用，但是操作系统如何提供几乎有无数个 CPU 可用的假象？

操作系统通过虚拟化（virtualizing）CPU 来提供这种假象。通过让一个进程只运行一个时间片，然后切换到其他进程，操作系统提供了存在多个虚拟 CPU 的假象。这就是时分共享（time sharing）CPU 技术，允许用户如愿运行多个并发进程。潜在的开销就是性能损失，因为如果 CPU 必须共享，每个进程的运行就会慢一点。

要实现 CPU 的虚拟化，要实现得好，操作系统就需要一些低级机制以及一些高级智能。我们将低级机制称为机制（mechanism）。机制是一些低级方法或协议，实现了所需的功能。例如，我们稍后将学习如何实现上下文切换（context switch），它让操作系统能够停止运行一个程序，并开始在给定的 CPU 上运行另一个程序。所有现代操作系统都采用了这种分时机制。

提示：使用时分共享（和空分共享）

时分共享（time sharing）是操作系统共享资源所使用的最基本的技术之一。通过允许资源由一个实体使用一小段时间，然后由另一个实体使用一小段时间，如此下去，所谓的资源（例如，CPU 或网络链接）可以被许多人共享。时分共享的自然对应技术是空分共享，资源在空间上被划分给希望使用它的人。例如，磁盘空间自然是一个空分共享资源，因为一旦将块分配给文件，在用户删除文件之前，不可能将它分配给其他文件。

在这些机制之上，操作系统中有一些智能以策略（policy）的形式存在。策略是在操作系统内做出某种决定的算法。例如，给定一组可能的程序要在 CPU 上运行，操作系统应该运行哪个程序？操作系统中的调度策略（scheduling policy）会做出这样的决定，可能利用历史信息（例如，哪个程序在最后一分钟运行得更多？）、工作负载知识（例如，运行什么类型的程序？）以及性能指标 （例如，系统是否针对交互式性能或吞吐量进行优化？）来做出决定。

4.1 抽象：进程

操作系统为正在运行的程序提供的抽象，就是所谓的进程（process）。正如我们上面所说的，一个进程只是一个正在运行的程序。在任何时刻，我们都可以清点它在执行过程中访问或影响的系统的不同部分，从而概括一个进程。

为了理解构成进程的是什么，我们必须理解它的机器状态（machine state）：程序在运行时可以读取或更新的内容。在任何时刻，机器的哪些部分对执行该程序很重要？

进程的机器状态有一个明显组成部分，就是它的内存。指令存在内存中。正在运行的程序读取和写入的数据也在内存中。因此进程可以访问的内存（称为地址空间，address space）是该进程的一部分。

进程的机器状态的另一部分是寄存器。许多指令明确地读取或更新寄存器，因此显然，它们对于执行该进程很重要。

请注意，有一些非常特殊的寄存器构成了该机器状态的一部分。例如，程序计数器（Program Counter，PC）（有时称为指令指针，Instruction Pointer 或 IP）告诉我们程序当前正在执行哪个指令；类似地，栈指针（stack pointer）和相关的帧指针（frame pointer）用于管理函数参数栈、局部变量和返回地址。

提示：分离策略和机制

在许多操作系统中，一个通用的设计范式是将高级策略与其低级机制分开[L+75]。你可以将机制看成为系统的“如何（how）”问题提供答案。例如，操作系统如何执行上下文切换？策略为“哪个（which）”问题提供答案。例如，操作系统现在应该运行哪个进程？将两者分开可以轻松地改变策略，而不必重新考虑机制，因此这是一种模块化（modularity）的形式，一种通用的软件设计原则。

最后，程序也经常访问持久存储设备。此类 I/O 信息可能包含当前打开的文件列表。

4.2 进程 API

虽然讨论真实的进程 API 将推迟到第 5 章讲解，但这里先介绍一下操作系统的所有接口必须包含哪些内容。所有现代操作系统都以某种形式提供这些 API。

- **创建（create）**：操作系统必须包含一些创建新进程的方法。在 shell 中键入命令或双击应用程序图标时，会调用操作系统来创建新进程，运行指定的程序。
- **销毁（destroy）**：由于存在创建进程的接口，因此系统还提供了一个强制销毁进程的接口。当然，很多进程会在运行完成后自行退出。但是，如果它们不退出，用户可能希望终止它们，因此停止失控进程的接口非常有用。
- **等待（wait）**：有时等待进程停止运行是有用的，因此经常提供某种等待接口。
- **其他控制（miscellaneous control）**：除了杀死或等待进程外，有时还可能有其他

控制。例如，大多数操作系统提供某种方法来暂停进程（停止运行一段时间），然后恢复（继续运行）。

- **状态（statu）**：通常也有一些接口可以获得有关进程的状态信息，例如运行了多长时间，或者处于什么状态。

4.3 进程创建：更多细节

我们应该揭开一个谜，就是程序如何转化为进程。具体来说，操作系统如何启动并运行一个程序？进程创建实际如何进行？

操作系统运行程序必须做的第一件事是将代码和所有静态数据（例如初始化变量）加载（load）到内存中，加载到进程的地址空间中。程序最初以某种可执行格式驻留在磁盘上（disk，或者在某些现代系统中，在基于闪存的 SSD 上）。因此，将程序和静态数据加载到内存中的过程，需要操作系统从磁盘读取这些字节，并将它们放在内存中的某处（见图 4.1）。

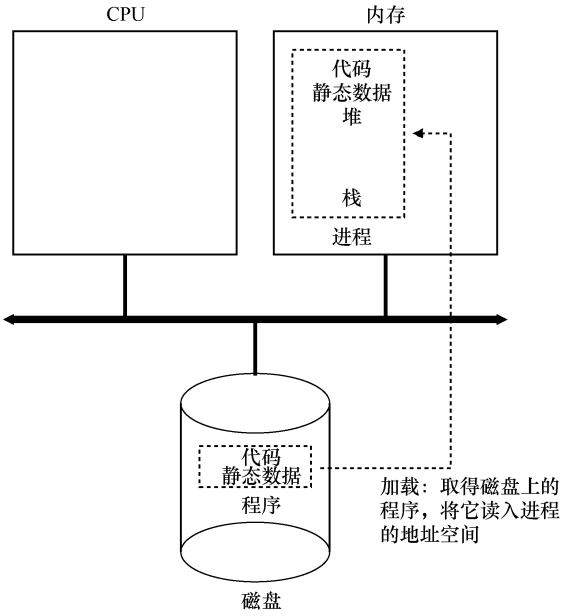

图 4.1 加载：从程序到进程

在早期的（或简单的）操作系统中，加载过程尽早（eagerly）完成，即在运行程序之前全部完成。现代操作系统惰性（lazily）执行该过程，即仅在程序执行期间需要加载的代码或数据片段，才会加载。要真正理解代码和数据的惰性加载是如何工作的，必须更多地了解分页和交换的机制，这是我们将来讨论内存虚拟化时要涉及的主题。现在，只要记住在运行任何程序之前，操作系统显然必须做一些工作，才能将重要的程序字节从磁盘读入内存。

将代码和静态数据加载到内存后，操作系统在运行此进程之前还需要执行其他一些操作。必须为程序的运行时栈（run-time stack 或 stack）分配一些内存。你可能已经知道，C

程序使用栈存放局部变量、函数参数和返回地址。操作系统分配这些内存，并提供给进程。操作系统也可能会用参数初始化栈。具体来说，它会将参数填入 main()函数，即 argc 和 argv 数组。

操作系统也可能为程序的堆（heap）分配一些内存。在 C 程序中，堆用于显式请求的动态分配数据。程序通过调用 malloc()来请求这样的空间，并通过调用 free()来明确地释放它。数据结构（如链表、散列表、树和其他有趣的数据结构）需要堆。起初堆会很小。随着程序运行，通过 malloc()库 API 请求更多内存，操作系统可能会参与分配更多内存给进程，以满足这些调用。

操作系统还将执行一些其他初始化任务，特别是与输入/输出（I/O）相关的任务。例如，在 UNIX 系统中，默认情况下每个进程都有 3 个打开的文件描述符（file descriptor），用于标准输入、输出和错误。这些描述符让程序轻松读取来自终端的输入以及打印输出到屏幕。在本书的第 3 部分关于持久性（persistence）的知识中，我们将详细了解 I/O、文件描述符等。

通过将代码和静态数据加载到内存中，通过创建和初始化栈以及执行与 I/O 设置相关的其他工作，OS 现在（终于）为程序执行搭好了舞台。然后它有最后一项任务：启动程序，在入口处运行，即 main()。通过跳转到 main()例程（第 5 章讨论的专门机制），OS 将 CPU 的控制权转移到新创建的进程中，从而程序开始执行。

4.4 进程状态

既然已经了解了进程是什么（但我们会继续改进这个概念），以及（大致）它是如何创建的，让我们来谈谈进程在给定时间可能处于的不同状态（state）。在早期的计算机系统[DV66，V+65]中，出现了一个进程可能处于这些状态之一的概念。简而言之，进程可以处于以下 3 种状态之一。

- **运行（running）**：在运行状态下，进程正在处理器上运行。这意味着它正在执行指令。
- **就绪（ready）**：在就绪状态下，进程已准备好运行，但由于某种原因，操作系统选择不在此时运行。
- **阻塞（blocked）**：在阻塞状态下，一个进程执行了某种操作，直到发生其他事件时才会准备运行。一个常见的例子是，当进程向磁盘发起 I/O 请求时，它会被阻塞，因此其他进程可以使用处理器。

如果将这些状态映射到一个图上，会得到图 4.2。如图 4.2 所示，可以根据操作系统的载量，让进程在就绪状态和运行状态之间转换。从就绪到运行意味着该进程已经被调度（scheduled）。从运行转移到就绪意味着该进程已经取消调度（descheduled）。一旦进程被阻塞（例如，通过发起 I/O 操作），OS 将保持进程的这种状态，直到发生某种事件（例如，I/O 完成）。此时，进程再次转入就绪状态（也可能立即再次运行，如果操作系统这样决定）。

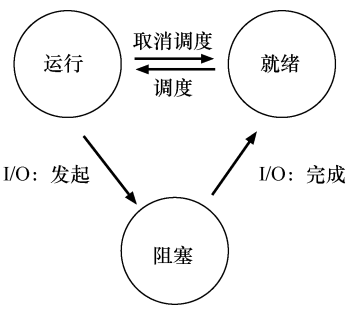

图 4.2 进程：状态转换

我们来看一个例子，看两个进程如何通过这些状态转换。首先，想象两个正在运行的进程，每个进程只使用 CPU（它们没有 I/O）。在这种情况下，每个进程的状态可能如表 4.1 所示。

表 4.1 跟踪进程状态：只看 CPU

时间	Process0	Process1	注
1	运行	就绪	
2	运行	就绪	
3	运行	就绪	
4	运行	就绪	Process0 现在完成
5	—	运行	
6	—	运行	
7	—	运行	
8	—	运行	Process1 现在完成

在下一个例子中，第一个进程在运行一段时间后发起 I/O 请求。此时，该进程被阻塞，让另一个进程有机会运行。表 4.2 展示了这种场景。

表 4.2 跟踪进程状态：CPU 和 I/O

时间	Process0	Process1	注
1	运行	就绪	
2	运行	就绪	
3	运行	就绪	Process0 发起 I/O
4	阻塞	运行	Process0 被阻塞
5	阻塞	运行	所以 Process1 运行
6	阻塞	运行	
7	就绪	运行	I/O 完成
8	就绪	运行	Process1 现在完成
9	运行	—	
10	运行	—	Process0 现在完成

更具体地说，Process0 发起 I/O 并被阻塞，等待 I/O 完成。例如，当从磁盘读取数据或等待网络数据包时，进程会被阻塞。OS 发现 Process0 不使用 CPU 并开始运行 Process1。当 Process1 运行时，I/O 完成，将 Process0 移回就绪状态。最后，Process1 结束，Process0 运行，然后完成。

请注意，即使在这个简单的例子中，操作系统也必须做出许多决定。首先，系统必须决定在 Process0 发出 I/O 时运行 Process1。这样做可以通过保持 CPU 繁忙来提高资源利用率。其次，当 I/O 完成时，系统决定不切换回 Process0。目前还不清楚这是不是一个很好的决定。你怎么看？这些类型的决策由操作系统调度程序完成，这是我们在未来几章讨论的主题。

4.5 数据结构

操作系统是一个程序，和其他程序一样，它有一些关键的数据结构来跟踪各种相关的信息。例如，为了跟踪每个进程的状态，操作系统可能会为所有就绪的进程保留某种进程列表（process list），以及跟踪当前正在运行的进程的一些附加信息。操作系统还必须以某种方式跟踪被阻塞的进程。当 I/O 事件完成时，操作系统应确保唤醒正确的进程，让它准备好再次运行。

图 4.3 展示了 OS 需要跟踪 xv6 内核中每个进程的信息类型[CK+08]。“真正的”操作系统中存在类似的进程结构，如 Linux、macOS X 或 Windows。查看它们，看看有多复杂。

从图 4.3 中可以看到，操作系统追踪进程的一些重要信息。对于停止的进程，寄存器上下文将保存其寄存器的内容。当一个进程停止时，它的寄存器将被保存到这个内存位置。通过恢复这些寄存器（将它们的值放回实际的物理寄存器中），操作系统可以恢复运行该进程。我们将在后面的章节中更多地了解这种技术，它被称为上下文切换（context switch）。

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
```

```
  struct inode *cwd;           // Current directory
  struct context context;      // Switch here to run process
  struct trapframe *tf;        // Trap frame for the
                               // current interrupt
};
```

图 4.3　xv6 的 proc 结构

从图 4.3 中还可以看到，除了运行、就绪和阻塞之外，还有其他一些进程可以处于的状态。有时候系统会有一个初始（initial）状态，表示进程在创建时处于的状态。另外，一个进程可以处于已退出但尚未清理的最终（final）状态（在基于 UNIX 的系统中，这称为僵尸状态[①]）。这个最终状态非常有用，因为它允许其他进程（通常是创建进程的父进程）检查进程的返回代码，并查看刚刚完成的进程是否成功执行（通常，在基于 UNIX 的系统中，程序成功完成任务时返回零，否则返回非零）。完成后，父进程将进行最后一次调用（例如，wait()），以等待子进程的完成，并告诉操作系统它可以清理这个正在结束的进程的所有相关数据结构。

补充：数据结构——进程列表

操作系统充满了我们将在这些讲义中讨论的各种重要数据结构（data structure）。进程列表（process list）是第一个这样的结构。这是比较简单的一种，但是，任何能够同时运行多个程序的操作系统当然都会有类似这种结构的东西，以便跟踪系统中正在运行的所有程序。有时候人们会将存储关于进程的信息的个体结构称为进程控制块（Process Control Block，PCB），这是谈论包含每个进程信息的 C 结构的一种方式。

4.6　小结

我们已经介绍了操作系统的最基本抽象：进程。它很简单地被视为一个正在运行的程序。有了这个概念，接下来将继续讨论具体细节：实现进程所需的低级机制和以智能方式调度这些进程所需的高级策略。结合机制和策略，我们将加深对操作系统如何虚拟化 CPU 的理解。

参考资料

[BH70]“The Nucleus of a Multiprogramming System”Per Brinch Hansen
Communications of the ACM, Volume 13, Number 4, April 1970
本文介绍了 Nucleus，它是操作系统历史上的第一批微内核（microkernel）之一。体积更小、系统更小的想法，在操作系统历史上是不断重复的主题。这一切都始于 Brinch Hansen 在这里描述的工作。

① 是的，僵尸状态。 就像真正的僵尸一样，这些“僵尸”相对容易杀死。 但是，通常建议使用不同的技术。

[CK+08]“The xv6 Operating System”
Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich
xv6 是世界上颇有魅力的、真实的小型操作系统。请下载并利用它来了解更多关于操作系统实际工作方式的细节。

[DV66]“Programming Semantics for Multiprogrammed Computations”Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
本文定义了构建多道程序系统的许多早期术语和概念。

[L+75]“Policy/mechanism separation in Hydra”
R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf SOSP 1975
一篇关于如何在名为 Hydra 的研究操作系统中构建一些操作系统的早期论文。虽然 Hydra 从未成为主流操作系统，但它的一些想法影响了操作系统设计人员。

[V+65]“Structure of the Multics Supervisor”
V.A. Vyssotsky, F. J. Corbato, R. M. Graham Fall Joint Computer Conference, 1965
一篇关于 Multics 的早期论文，描述了我们在现代系统中看到的许多基本概念和术语。计算作为实用工具，这背后的一些愿景终于在现代云系统中得以实现。

作业

补充：模拟作业

模拟作业以模拟器的形式出现，你运行它以确保理解某些内容。模拟器通常是 Python 程序，它们让你能够生成不同的问题（使用不同的随机种子），也让程序为你解决问题（带-c 标志），以便你检查答案。使用-h 或--help 标志运行任何模拟器，将提供有关模拟器所有选项的更多信息。

每个模拟器附带的 README 文件提供了有关如何运行它的更多详细信息，其中详细描述了每个标志。

程序 process-run.py 让你查看程序运行时进程状态如何改变，是在使用 CPU（例如，执行相加指令）还是执行 I/O（例如，向磁盘发送请求并等待它完成）。详情请参阅 README 文件。

问题

1．用以下标志运行程序：./process-run.py -l 5:100,5:100。CPU 利用率（CPU 使用时间的百分比）应该是多少？为什么你知道这一点？利用 -c 标记查看你的答案是否正确。

2．现在用这些标志运行：./process-run.py -l 4:100,1:0。这些标志指定了一个包含 4 条指

令的进程（都要使用 CPU），并且只是简单地发出 I/O 并等待它完成。完成这两个进程需要多长时间？利用-c 检查你的答案是否正确。

3．现在交换进程的顺序：./process-run.py -l 1:0,4:100。现在发生了什么？交换顺序是否重要？为什么？同样，用-c 看看你的答案是否正确。

4．现在探索另一些标志。一个重要的标志是-S，它决定了当进程发出 I/O 时系统如何反应。将标志设置为 SWITCH_ON_END，在进程进行 I/O 操作时，系统将不会切换到另一个进程，而是等待进程完成。当你运行以下两个进程时，会发生什么情况？一个执行 I/O，另一个执行 CPU 工作。（-l 1:0,4:100 -c -S SWITCH_ON_END）

5．现在，运行相同的进程，但切换行为设置，在等待 I/O 时切换到另一个进程（-l 1:0,4:100 -c -S SWITCH_ON_IO）。现在会发生什么？利用-c 来确认你的答案是否正确。

6．另一个重要的行为是 I/O 完成时要做什么。利用-I IO_RUN_LATER，当 I/O 完成时，发出它的进程不一定马上运行。相反，当时运行的进程一直运行。当你运行这个进程组合时会发生什么？（./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p）系统资源是否被有效利用？

7．现在运行相同的进程，但使用-I IO_RUN_IMMEDIATE 设置，该设置立即运行发出 I/O 的进程。这种行为有何不同？为什么运行一个刚刚完成 I/O 的进程会是一个好主意？

8．现在运行一些随机生成的进程，例如-s 1 -l 3:50,3:50, -s 2 -l 3:50,3:50, -s 3 -l 3:50,3:50。看看你是否能预测追踪记录会如何变化？当你使用-I IO_RUN_IMMEDIATE 与-I IO_RUN_LATER 时会发生什么？当你使用-S SWITCH_ON_IO 与-S SWITCH_ON_END 时会发生什么？