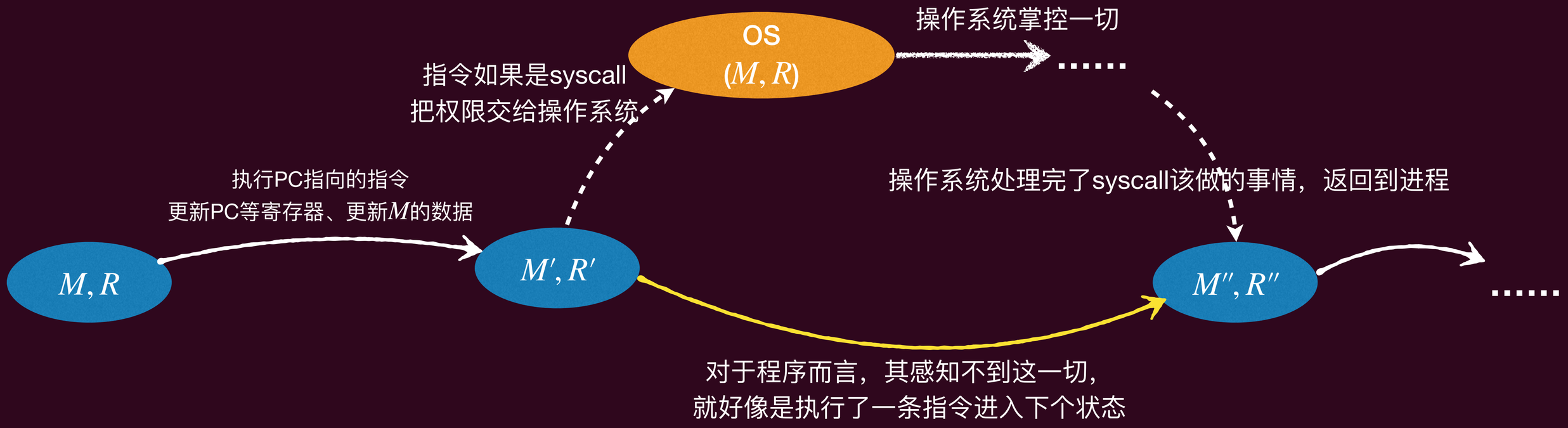
课程回顾
Course Review

钮鑫涛
南京大学
2025春

本门课由蒋炎岩和钮鑫涛共同构建

一切皆是状态机

- 进程是一个状态机，内存和寄存器代表其状态，指令（无论是直接的低级机器指令还是操作系统“虚拟”出来的高级指令－系统调用）的执行迁移了状态



一切皆是状态机

- 操作系统本身也是状态机

初始化阶段

Bootloader transfer to load kernel

initialize and configures memory, processors, I/O, and storage devices, sets up interrupt handling (IRQs).

Mount the initrd and load essential drivers then Unmount initrd

Mount the root filesystem

init

Start the first user-level process: **init** (the newest version of **init** in linux is **systemd**)

Many user processes will start from here!

yield cpu

形成这样一个状态机

一切皆是状态机

- 操作系统本身也是状态机

运行阶段

用户进程1　用户进程2　…

自陷(trap)

返回用户空间，用户进程继续

syscall API

执行相应系统调用代码

执行相中断处理程序

Meta(进程1)	Meta(进程2)	…	内核堆	内核栈1	内核栈2	…	内核代码区域
Meta(文件1)	Meta(文件2)	…					
Meta(地址1)	Meta(地址2)	…					

s

Meta(进程1)	Meta(进程2)	…	内核堆	内核栈1	内核栈2	…	内核代码区域
Meta(文件1)	Meta(文件2)	…					
Meta(地址1)	Meta(地址2)	…					

s'

Meta(进程1)	Meta(进程2)	…	内核堆	内核栈1	内核栈2	…
Meta(文件1)	Meta(文件2)	…				
Meta(地址1)	Meta(地址2)	…				

s''

可能需要硬件（如disk）辅助

发生某些中断事件（如硬盘读写完成）

硬件1　硬件2　…

Meta为元信息，比如对于进程而言，操作系统只在内核中维护进程的元信息（进程控制表，存放如进程id、进程栈指针、PC…）

一切皆是状态机

- 操作系统作为一个状态机的特殊性在于：其可以管理其他状态机
- 从另一个角度来说明这件事就是：操作系统本身是一个状态机，那么其他状态机就是运行在这个状态机上的状态机
- 因此，某种程度上来说，操作系统是一个“通用机”
  - 其可以运行其他所有图灵机
- 通用机本身也是图灵机，其也可以跑在其他“通用机”里
  - 虚拟机—操作系统的“操作系统”

抽象

- 得到这样的简洁、漂亮的“状态机”模型的关键来自“抽象”技术！
- 操作系统为上层应用可以得到一个干净简洁的接口，而不是直面硬件
  - 地址：无限，独占的连续的存储空间
    - 而不是实际的“坑坑洼洼”的内存
  - 文件：名称，读写操作完成所有的持久化操作
    - 而不是各种不同的磁盘设备，各种繁琐的操作

抽象

- 能够提供这些干净接口的能力还是来自抽象！

⬍ `int file_descriptor, char* filename`

⬍ `struct file`

⬍ `struct inode`

⬍ `struct buf`

⬍ `struct buf`

⬍ `struct buf`

⬍ `in/out device registers`

每一层都是仅使用其下层的 API 实现
每一层的 API 必须设计为直接满足其上层的需求

抽象

- 抽象的另一个好处在于进程运行在“抽象”的资源上，而不是具体的资源，其对具体的资源是不可见的
- 这些虚拟资源时操作系统提供的，这就使的操作系统可以“欺骗”进程
  - 内存是虚拟的，并不是实际的内存，进程可能都没有装入内存，而是在磁盘上, 只有在需要时才被装入内存（Demand paging）
  - 进程之间彼此共享了一整个大的物理内存，但由于“不可见”物理内存，所以相安无事
  - 这就是虚拟化，每个进程都有无限连续空间

抽象

- 抽象的另一个好处在于进程运行在“抽象”的资源上，而不是具体的资源，其对具体的资源是不可见的
- 这些虚拟资源时操作系统提供的，这就使的操作系统可以“欺骗”进程
  - CPU也不是独占的，而是操作系统配合硬件中断形成“分时”的共享CPU
  - 但由于操作系统霸占中断，进程对这一切完全无知
  - 每个进程都感觉自己独占了整个CPU！

保护

- 然而这些虚像和物理资源往往是“少”对“多”的对应关系
  - 本质上操作系统在做一些“拆东墙补西墙”的工作
    - 这个时候如何避免“拆”掉关键的承重墙呢?
  - 操作系统在实现这些虚像时的一个重要责任即为保护
    - 保护进程（线程）之间没有冲突

保护

- 比如在内存的分配上，不能出现两个进程分配的内存出现重叠（共享内存除外）
- 比如在使用CPU上，操作系统必须保证分时的正确实现，即上下文切换，使的一个进程的运行时寄存器可以安全的先切到内存上，然后再切回来，整个过程对于进程而言是没有感知的
- 比如文件系统上的各种读、写、用户组的权限，必须使其没有发生越权的行为
  - 权限的定义在内存中也有类似设置（只读、只写、执行...）
  - CPU的执行也定义了一些特权指令，使的用户进程禁止直行这些指令，只有通过操作系统才能执行

保护

- 并发编程的一个关键部分就是保护，即不能违背程序的“安全性”
  - 互斥的作用在于让多线程不能无节制访问共享内存，否则会出现“竞态条件”从而导致结果失效
  - 如何实现互斥是操作的重要部分，在现有体系架构下（编译器、处理器乱序）已经难以用软件实现互斥了
    - 因此需要软硬件结合的方式
    - 高效的实现需要考虑现实的workload
      - fast path (自旋)
      - slow path （sleep and wake up ）

保护

- 并发编程的另一大问题是同步，即如何控制多个线程达成一个既定目标
- 条件变量、信号量
- 然而并发还是困难的，因为极容易出错，比如死锁
- 因此一个好的保护机制不仅仅在源头避免，还需要在现实中设计很多“防御性编程”
  - 了解问题的安全（活性）性质，多写assert吧！

性能

- 一个奇慢无比的操作系统是没人用的
- 调度本质上就是为了性能！
  - 页缓存调度策略
  - 进程调度
  - 磁盘I/O调度

性能

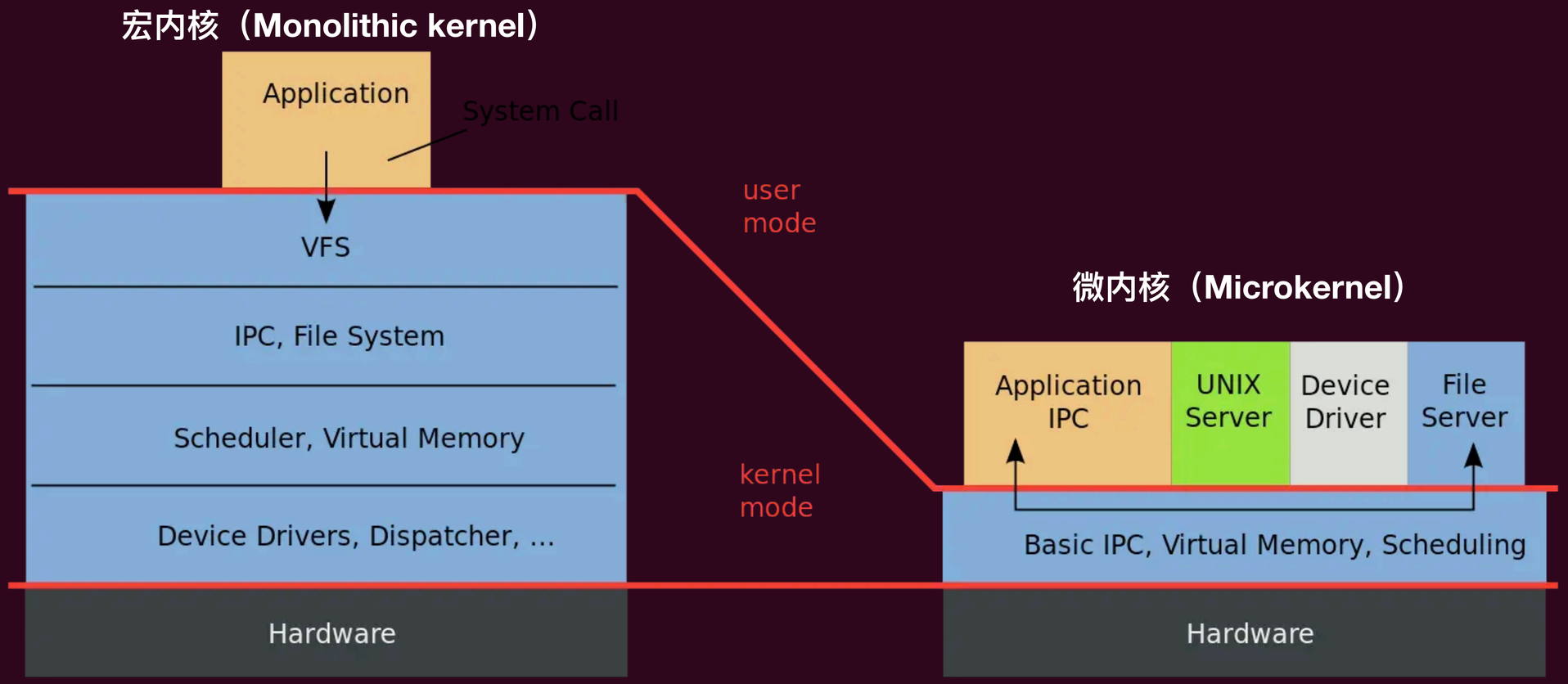
- 在工程实现上，为了达到很好的性能，需要充分考虑现实的workload
- 这是“现实”和“理论”的界限
- 比如之前的futex、读写锁的设计、Read-copy-update
- 比如内存分配的fast（slab）和slow（buddy）
- 比如文件系统的索引结构，为了现实中大部分存在的都是“小”文件

性能

- 各种缓存层的存在也是为了提高性能（cache, LTB, page buffer, buffer cache）
- 抽象层次的过多会导致性能直接下降
  - 比如linux就有直接I/O的访问
  - 比如进程的受限直接执行（Limited Direct Execution）策略，并不是完全跑在操作系统上，而是有一部分是直接在CPU上

复杂VS性能

- 复杂和性能似乎是天生的死敌
- 这在操作系统本身的系统设计上也有体现



谢谢