虚拟化：进程抽象

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组

进程 (Process)

- 定义
  - A process is a running program.

- 计算机专业的两大哲学问题
  - 什么是程序？ (什么是计算？)
  - 什么是程序的运行？ (什么是计算机？)

什么是程序?

什么是程序?

- A computer program is a collection of instructions that performs a specific task when executed by a computer.

  -- Wikipedia

- 无论是C, Java, Go, Scala, JavaScript程序，最终都成为了指令在计算机上执行(因为狭义的计算机是执行指令的机器)

- 程序实现了纯粹的计算，以及通过特殊的指令与外界交互
  - 计算：内存无限大时其能力等同于一切可计算物(Turing Machine)
  - 交互：通过物理/机械手段将结果输出

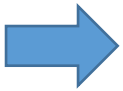
指令集

- 指令集定义了程序(指令)执行的环境和行为
  - 通常的执行环境(状态)：内存M；寄存器R
  - 例如$M[\texttt{0x8048000}] = \texttt{0x4f}$, $R[\texttt{eax}] = \texttt{0x00001234}$
  - 行为：确定(deterministic)的语义
    - 区别于non-deterministic

旧状态(M, R)

$R[\texttt{eip}] = x \wedge M[x] = \texttt{0xc3}$

(旧状态满足的条件)



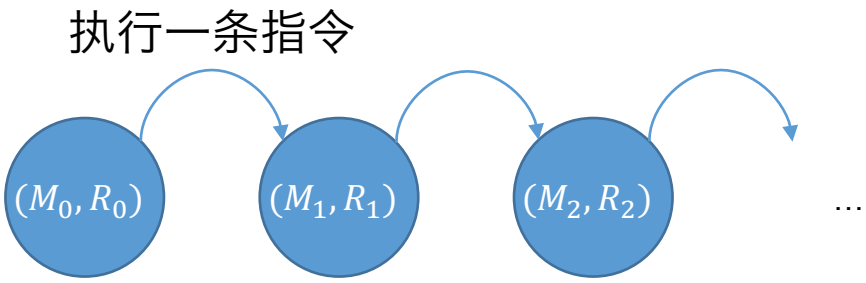
新状态$(M', R') = (M, R)$除:

$R'[\texttt{eip}] = M[R[\texttt{esp}]];$
$R'[\texttt{esp}] = R[\texttt{esp}] + 4;$

(新状态满足的条件)

程序的执行

- 随着时间的推进，根据指令集的语义不断改变状态(M, R)
  - 计算机 = 状态机
  - NEMU恰好模拟了这个过程



(程序执行示意图)

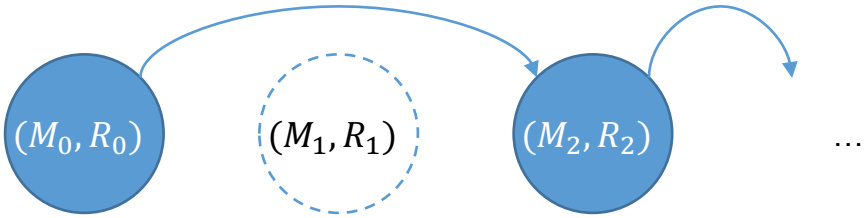
[号外] 为什么“指令”会是这样的形式?

- 指令是用来描述解决任务的步骤的
  - 硬件(数字逻辑电路)能力是有限的
  - CISC指令：一条指令可以描述一个工作(字符串拷贝，如movsb)
  - RISC指令：访问0/1个内存字、读写若干寄存器

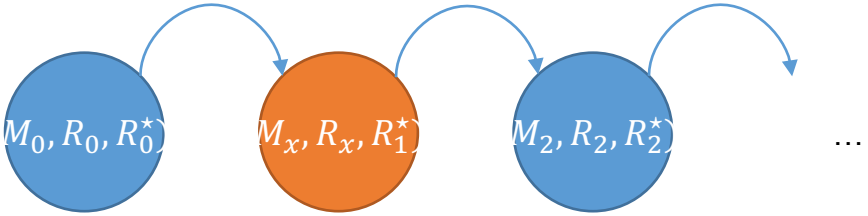
- 指令的设计
  - 考虑了存储器访问的速度特性：内存/寄存器要分开
    - 一旦设计，难以更改，只能补充(x86)
  - 指令有很多种其他设计：向量计算机、GPU、……
    - 针对特定领域有很大的设计空间(DianNao系列；寒武纪)

为什么“指令”会是这样的形式？(cont'd)

- 指令只是定义了程序的语义，并没有规定指令实际在计算机中的执行
  - 在有一个周期执行两条指令，有可能看不到(M_1, R_1)，因此模拟器(QEMU)的行为和真机有出入



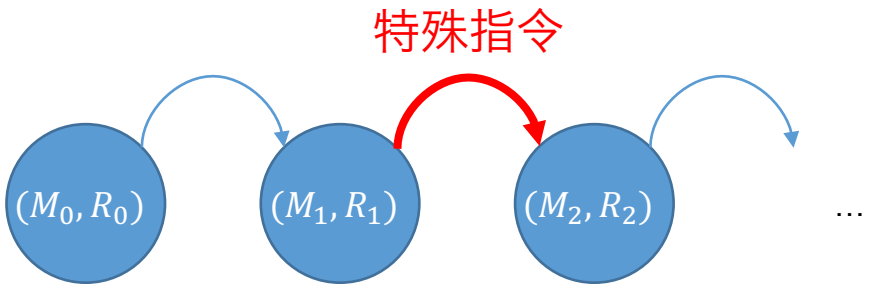
  - 有可能处理器有内部的状态$R^\star$(如缓存)



  - 如果有来自外星的神秘计算机，可能能直接计算出执行的结果

与外界交互

- 可以通过特殊的指令与外界交互
  - 特殊指令可能不改变自身的状态(输出)
  - 也可能改变内存/寄存器的值(系统调用可以看作特殊指令)

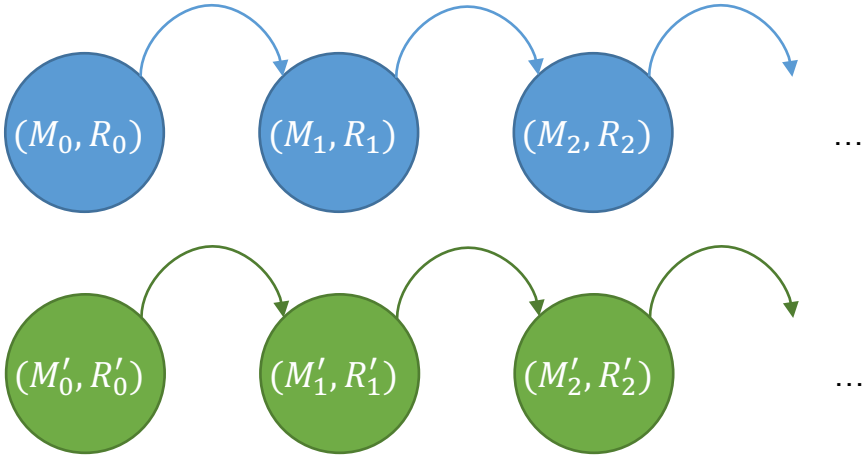


- 在计算机只能执行一个程序的时代，这基本准确地反映了计算机的工作方式

进程：运行的程序

人类总是不容易满足的……

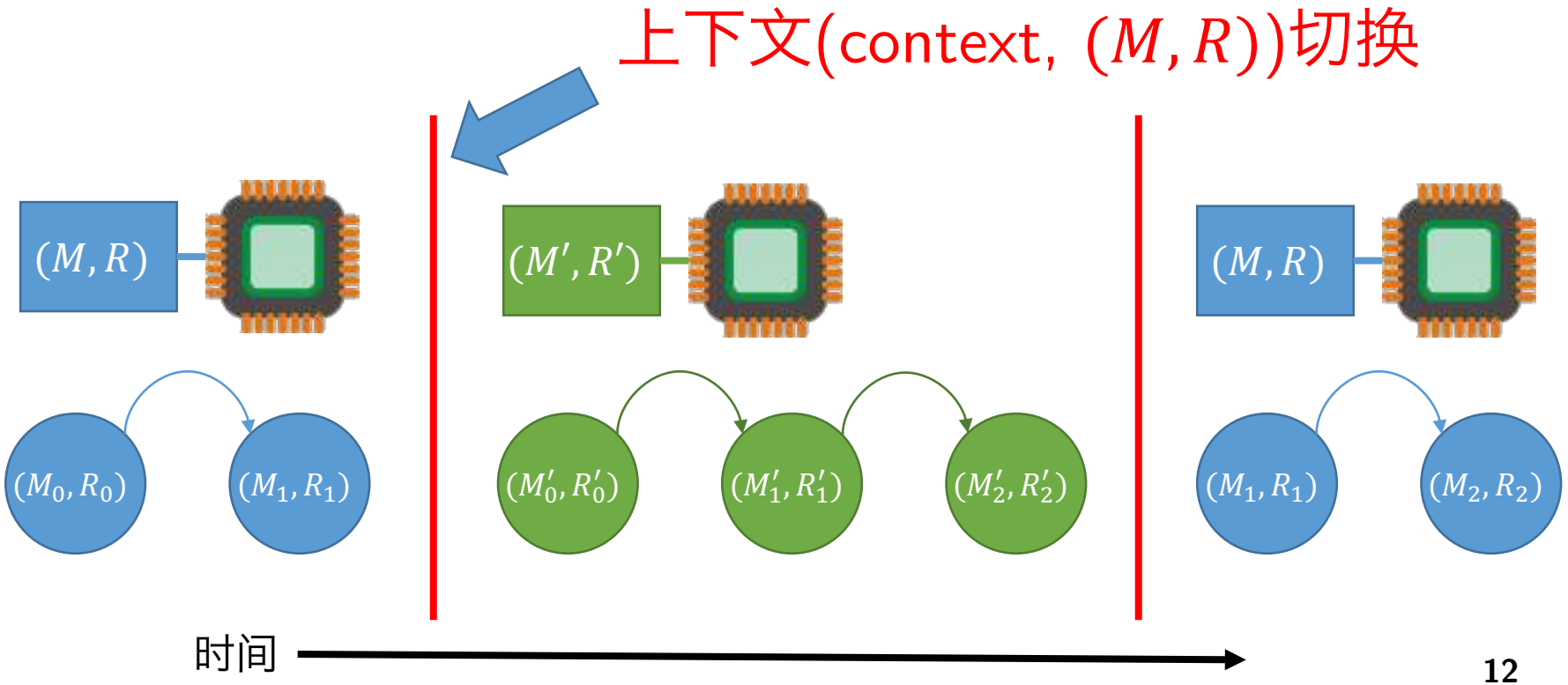
- 凭什么不能一边看电影一边写代码?
  - 如果要运行两个程序，准备两份(M, R)，弄两个处理器就行了



- 但要运行10,000个程序，这好像不太行得通
  - 分时共享一个处理器、一份内存、一组寄存器

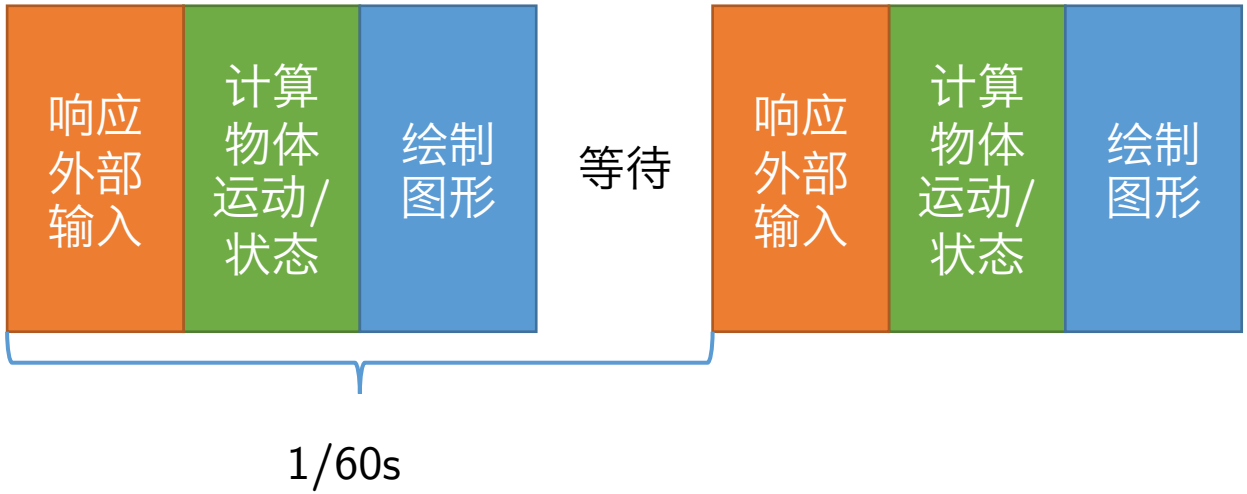
分时共享处理器

- 先假设给每个程序独立的内存和寄存器
  - 处理器 = 执行指令的部件(箭头)
  - 连上(M, R)执行就行了



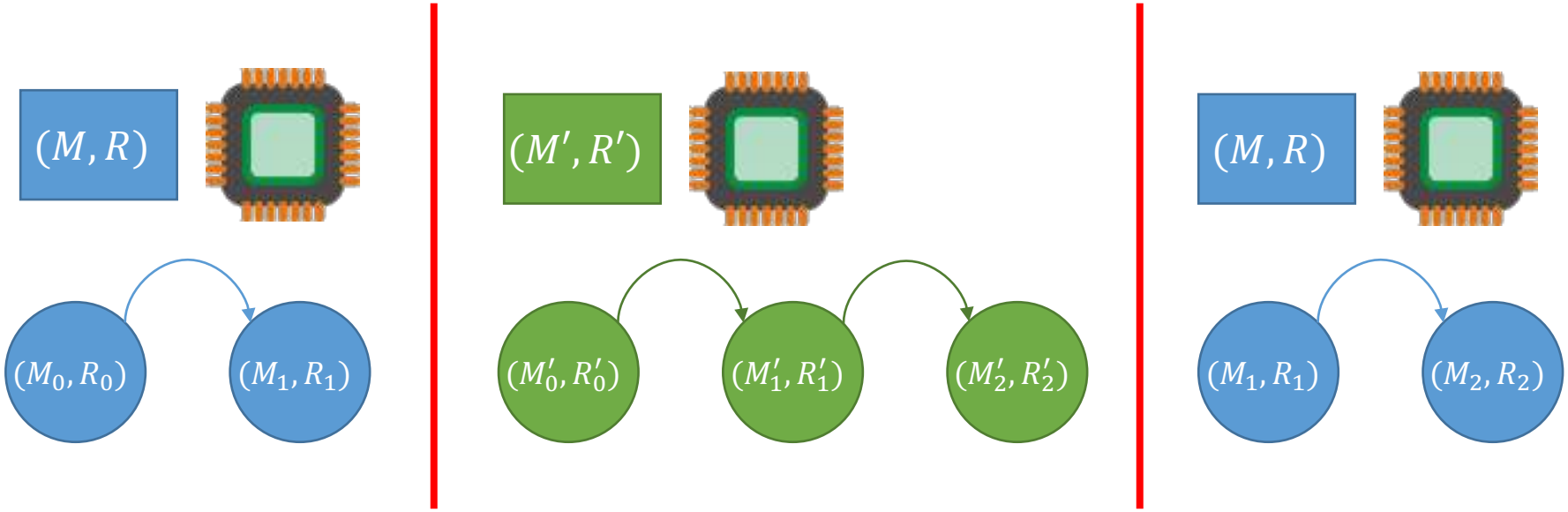
分时共享处理器的例子

- 分时无处不在，一个程序内也可以分时执行不同的功能



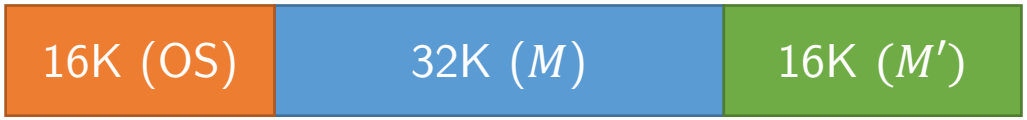
分时共享物理内存

- 我们把物理内存根据程序的大小分成两份(比如16KB和48KB)
- 设置计算机系统的状态，使它访问M的行为发生改变
  - 根据当前运行的程序
  - 决定$M[x]$访问$PM[x+0]$还是$PM[x+16384]$ (PM是物理内存)



例子：分段式的内存管理

- 只允许访问一段物理内存：$[\ell, r)$
  - [16K,48K), [48K, 64K)

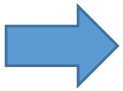


- CPU执行指令的时候的语义作出相应调整

旧状态(M, R)

$R[\texttt{eip}] = x \wedge M[\ell + x] = \texttt{0xc3}$

(旧状态满足的条件)



新状态$(M', R') = (M, R)$除：

$R'[\texttt{eip}] = M[\ell + R[\texttt{esp}]];$
$R'[\texttt{esp}] = R[\texttt{esp}] + 4;$

(新状态满足的条件)

内存管理：理论与实践

- 为每个进程维护映射$VM(x)$将
  - 进程视角的地址(虚拟地址)映射到物理地址
    - 访问$M[x]$实际访问物理内存$PM[VM(x)]$
  - 分段：$VM(x) = x + \ell_x$

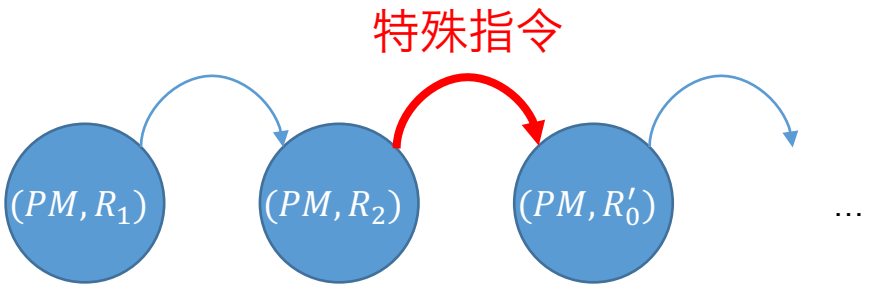
- $VM(x)$体现了设计上的权衡
  - 内存数量很大，但$VM(x)$内存占用要小、查找速度要快
  - 讨论：如何设计?

分时共享寄存器

- 寄存器本质上和内存一样，都是数据
- 平时将CPU让给程序执行$(M_i, R_i) \Rightarrow (M_{i+1}, R_{i+1})$

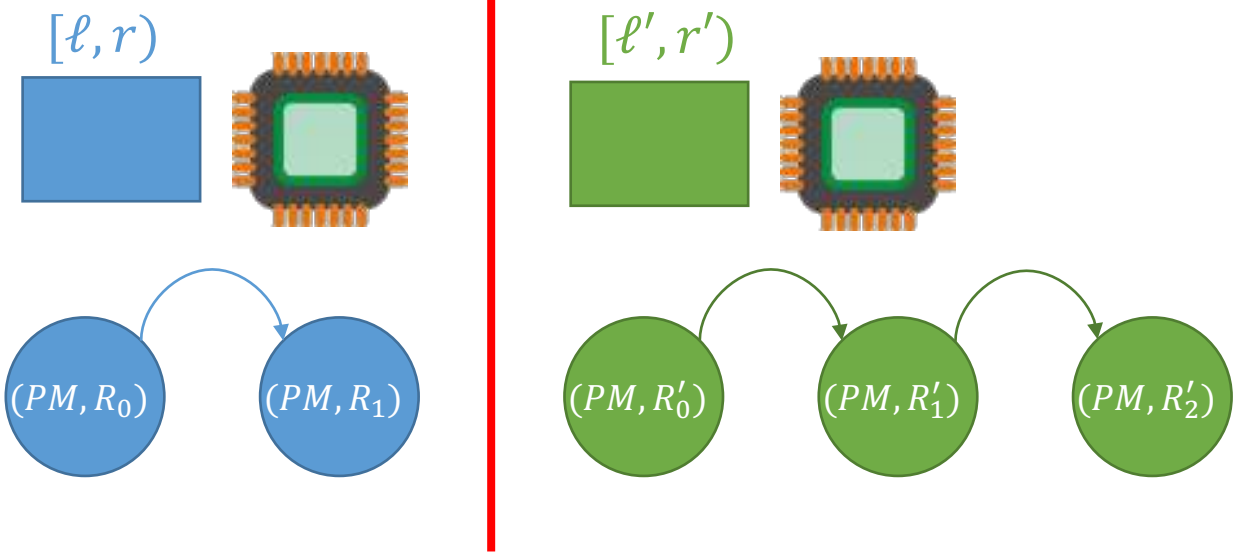
- 在必要的时候“上位”——操作系统代码执行
  - 把当前的R保存到操作系统预留的内存里
  - 找到保存的另一个进程的R' (包含内存描述$[\ell', r')$)，切换到R'

- 操作系统的上位方式
  - 执行一条特殊的跳转



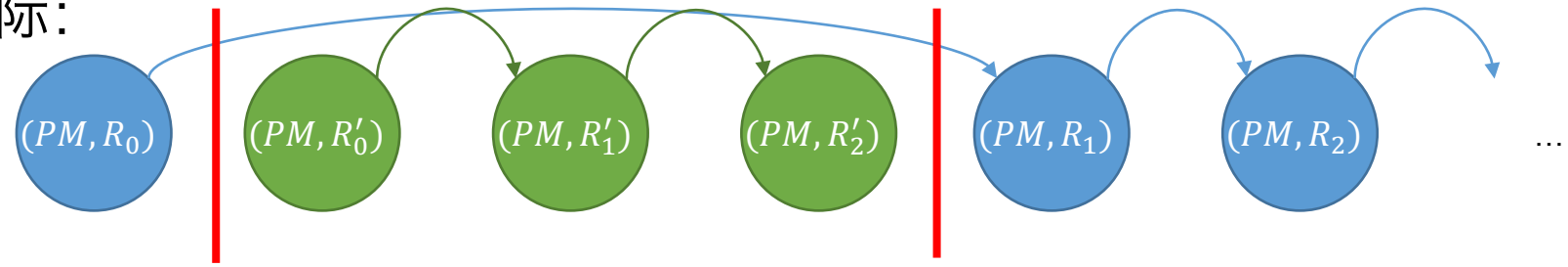
操作系统：上下文切换实现的分时共享

- 在红色的“切换”部分
  - M无需切换，只需切换处理器的“视角”：VM (例如$[\ell, r)$)
  - R需要切换；VM其实是R的一部分
  - R决定了处理器执行的指令(箭头)，最终实现处理器分时共享

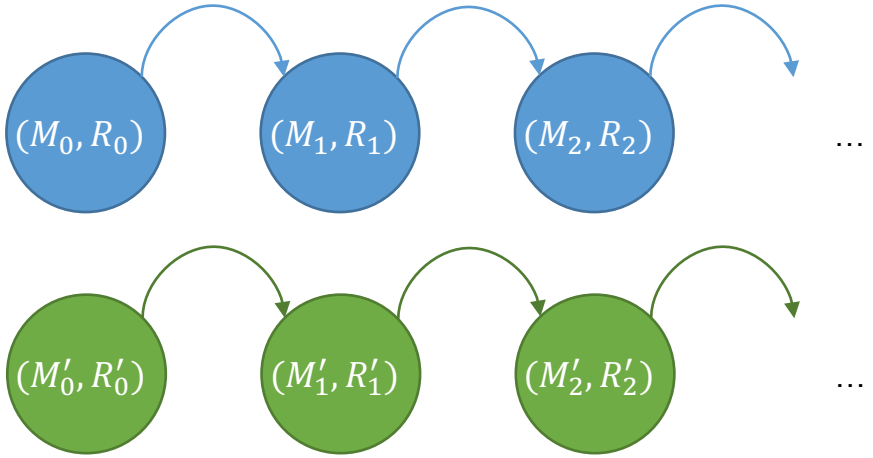


分时共享：并发执行

- 实际：



- 外界看起来：

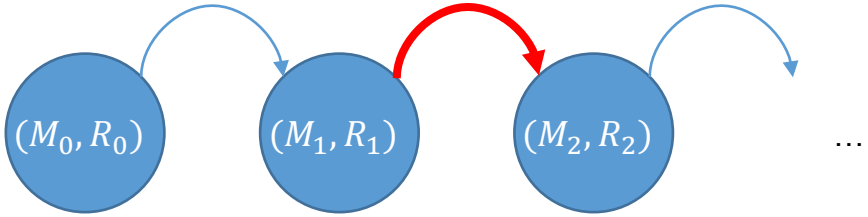


- 一学期同时在上多门课，但同一时间只能上一门课

虚拟化：进程抽象

- 每个进程有自己被保护的虚拟内存，共享一套系统调用
  - 进程看不到$[\ell, r)$，好像自己拥有连续的内存

- 系统里能同时运行多个进程
  - 系统调用由特殊指令完成(系统调用可以看做是一次状态的转换)



  - 跳转至操作系统有时被强制执行(借助中断)