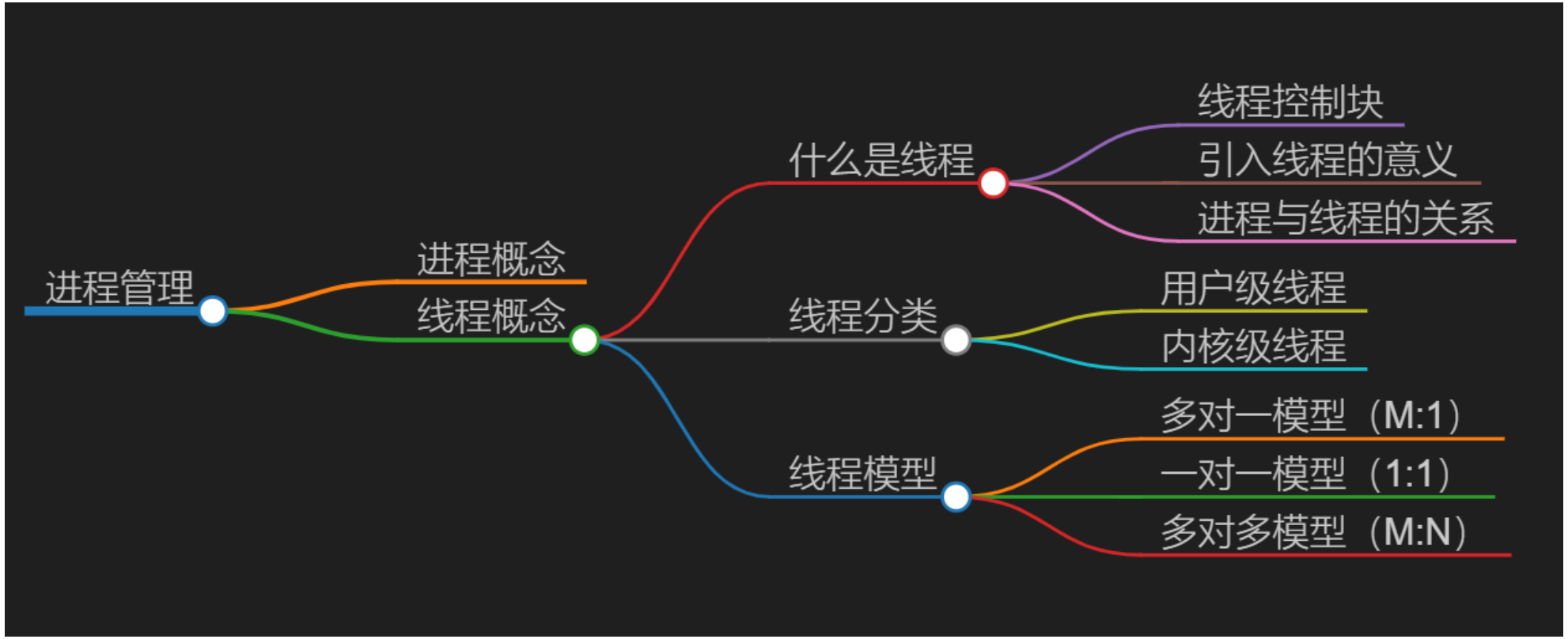
操作系统

L06 CPU调度算法1

胡燕
大连理工大学 软件学院

6.0-前情回顾

线程


进程管理
进程概念
线程概念
什么是线程
线程控制块
引入线程的意义
进程与线程的关系
线程分类
用户级线程
内核级线程
线程模型
多对一模型（M:1）
一对一模型（1:1）
多对多模型（M:N）

CPU调度

CPU Scheduling

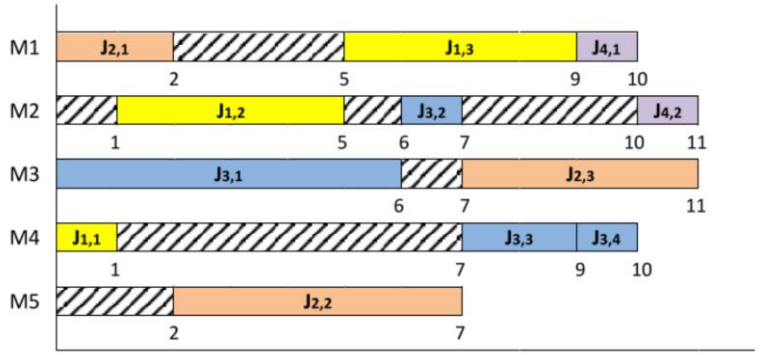
01

为什么进行CPU调度

调度

Scheduling

为什么进行CPU调度



Job Shop Scheduling

Boarding Gate Scheduling

Digital Scheduling

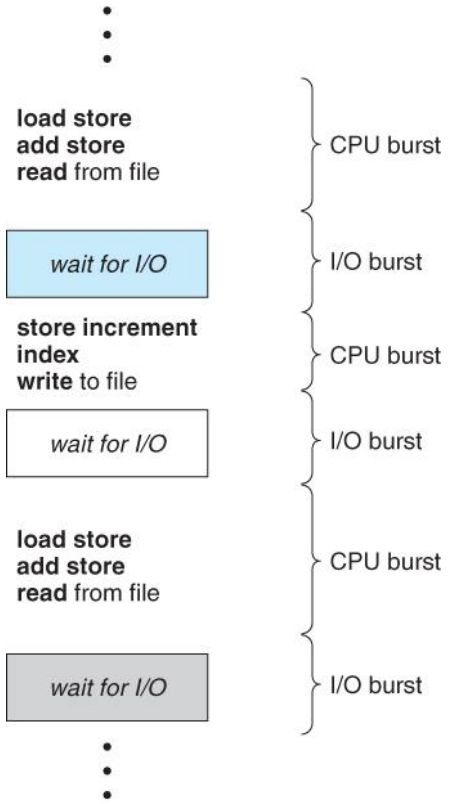
调度在生活中无处不在

为什么进行CPU调度

调度的核心目标： 效率

为什么进行CPU调度

程序执行特点：CPU周期-I/O周期交替



- 任务串行执行：效率低
  - CPU轻松，干一阵休一阵 （利用率不高）

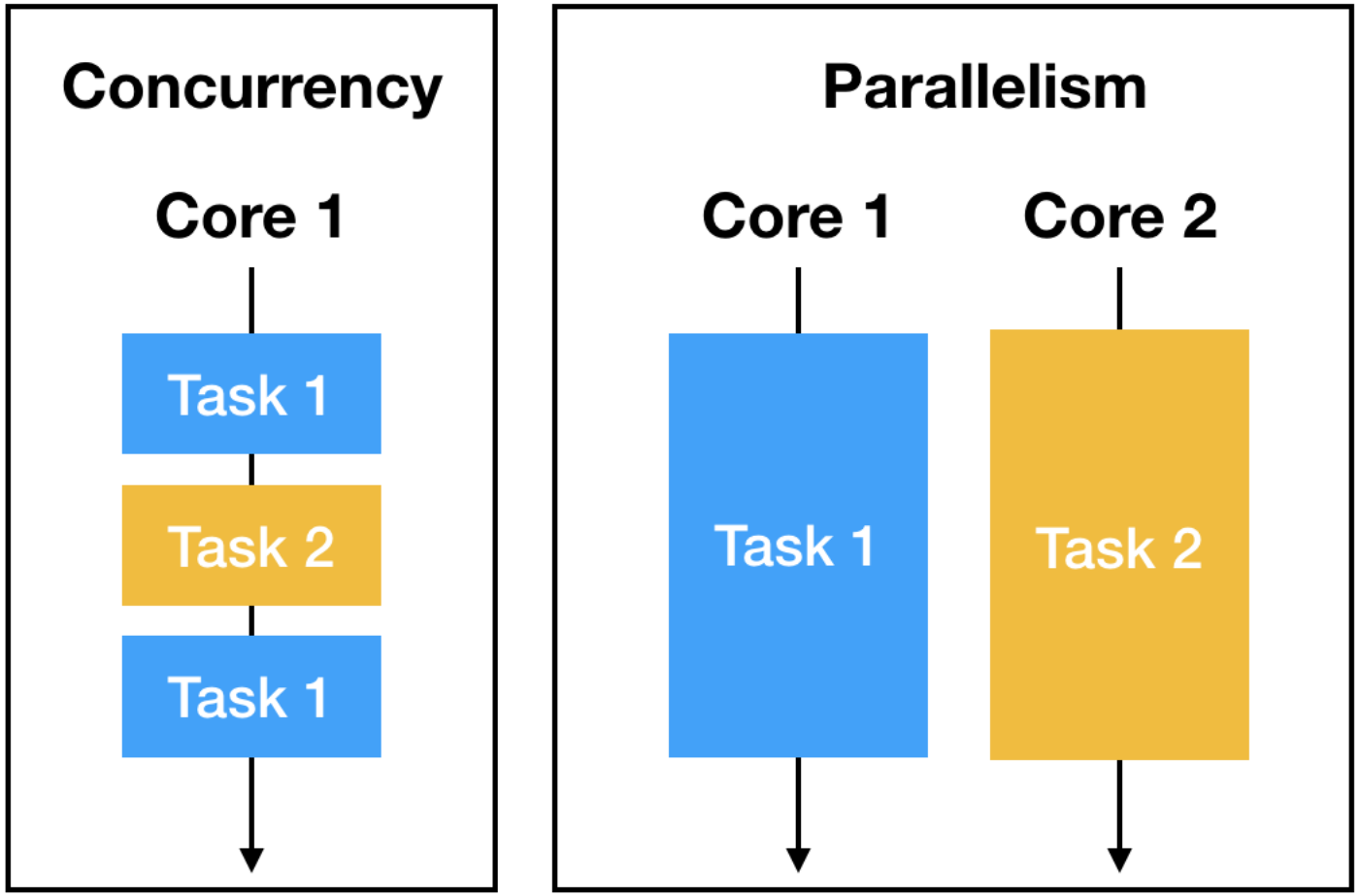
三天打鱼，两天晒网：不可取！

正确做法：

见缝插针： to make use of every bit of time

为什么进行CPU调度

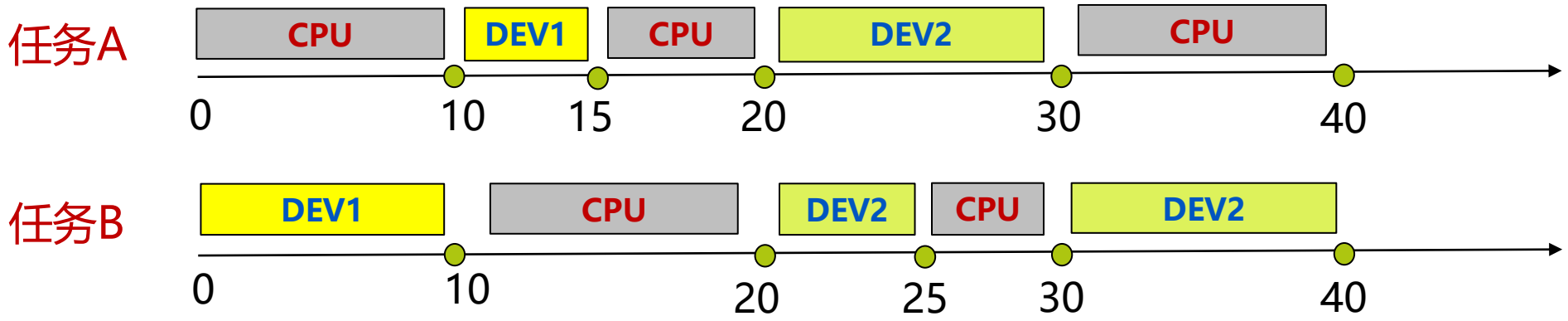
调度与并发



为什么进行CPU调度

调度与并发

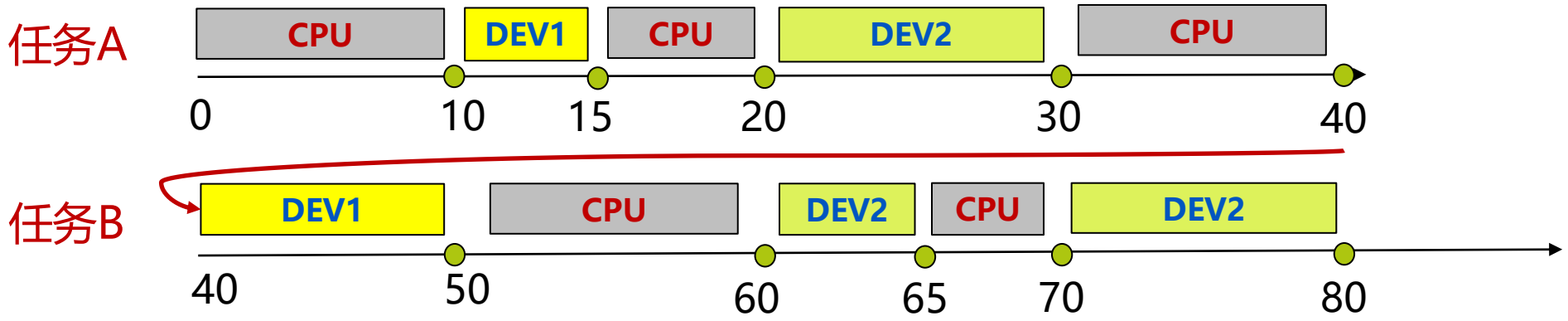
单核CPU下的程序A和程序B的执行



为什么进行CPU调度

调度与并发

单核CPU下的程序A和程序B的执行



sequential execution（串行执行结果）：

Cumulative Time 周转时间：80

CPU Efficiency：40/80 = 50%

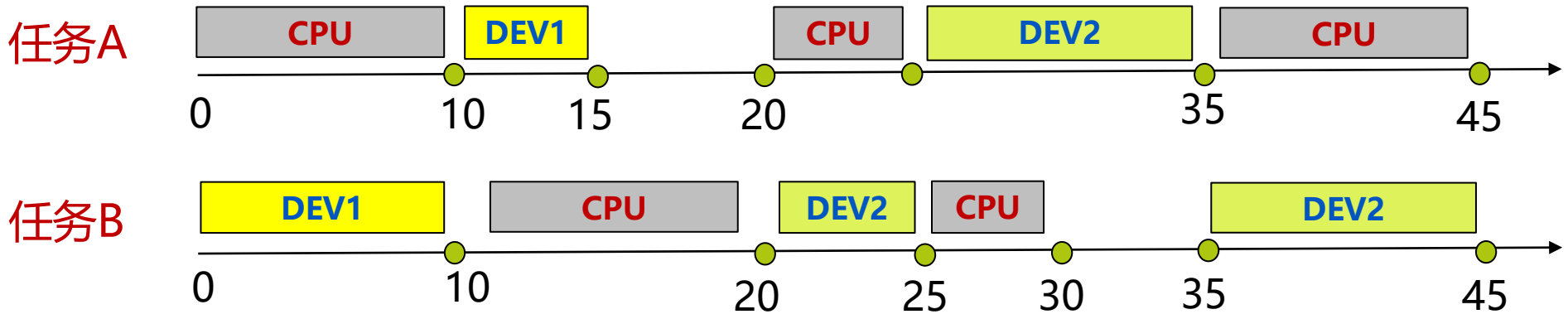
DEV1 Efficiency：15 / 80 = 18.75%

DEV2 Efficiency：25 / 80 = 31.25%

为什么进行CPU调度

调度与并发

单核CPU下的程序A和程序B的执行



Concurrency（并发执行结果）

Cumulative Time 周转时间：45

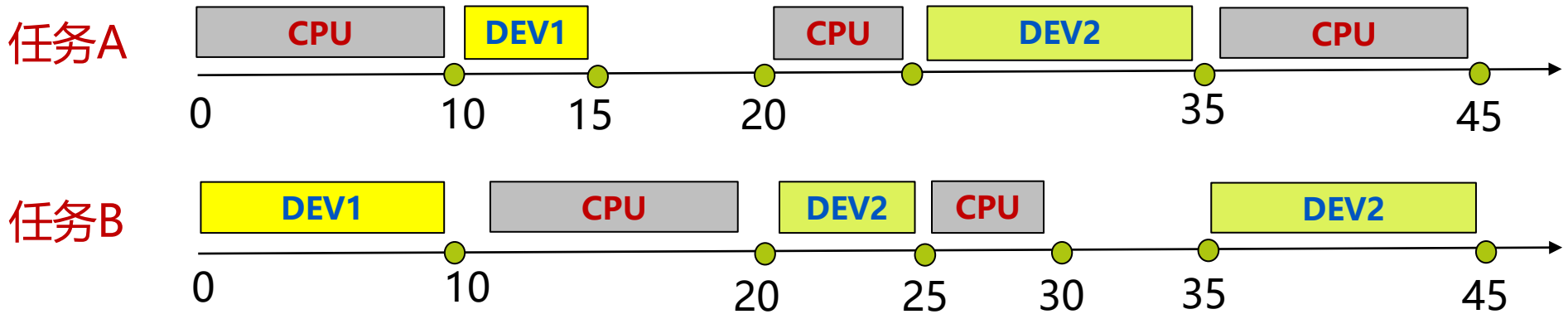
CPU Efficiency： 40 / 45 = 89%

DEV1 Efficiency： 15 / 45 = 33%

DEV2 Efficiency： 25 / 45 = 55.6%

填空题 2分

设置

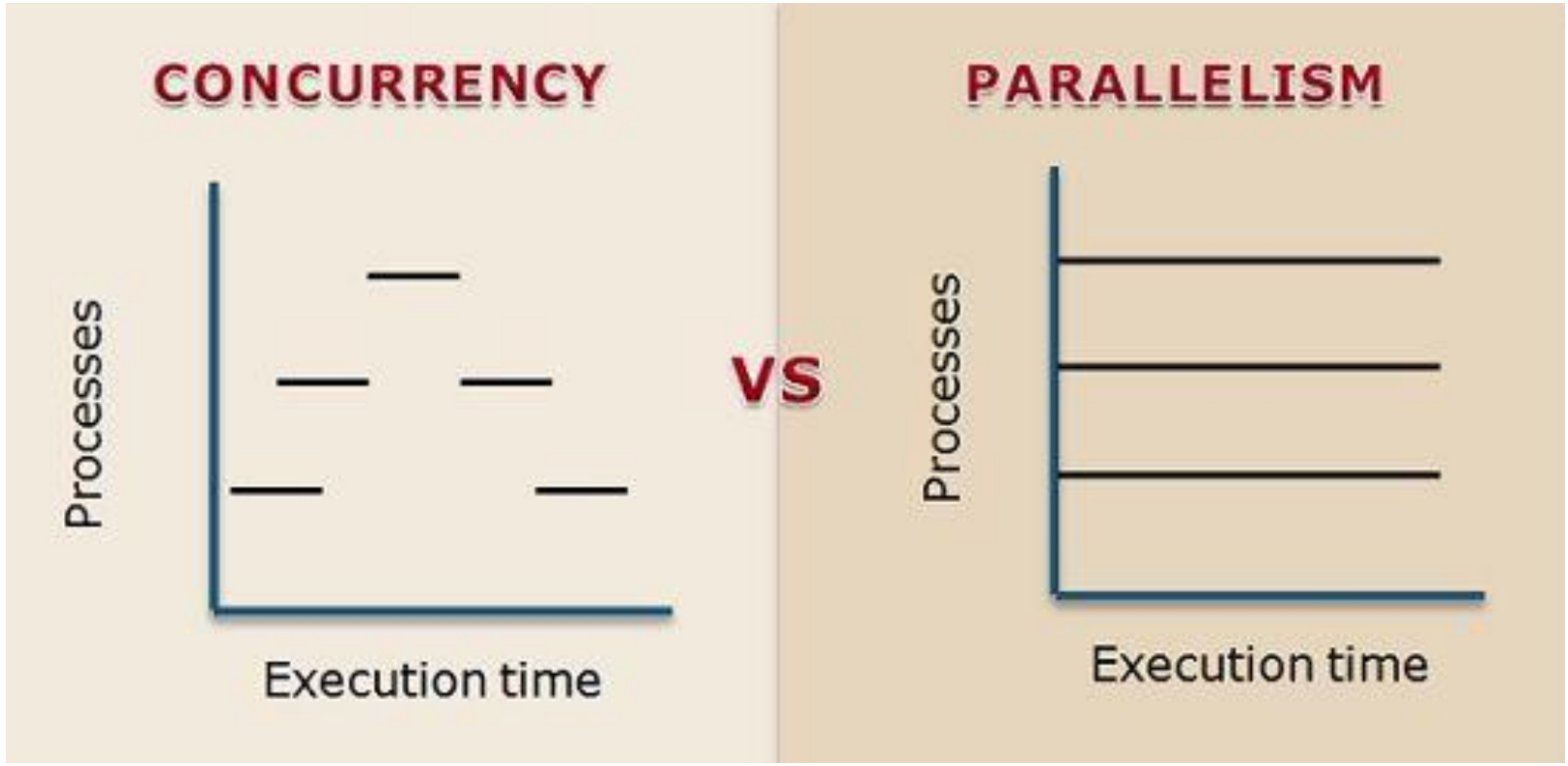


在并发执行模式下，进程A在15-20这个时间段内，处于（ [填空1] ）状态。

作答

为什么进行CPU调度

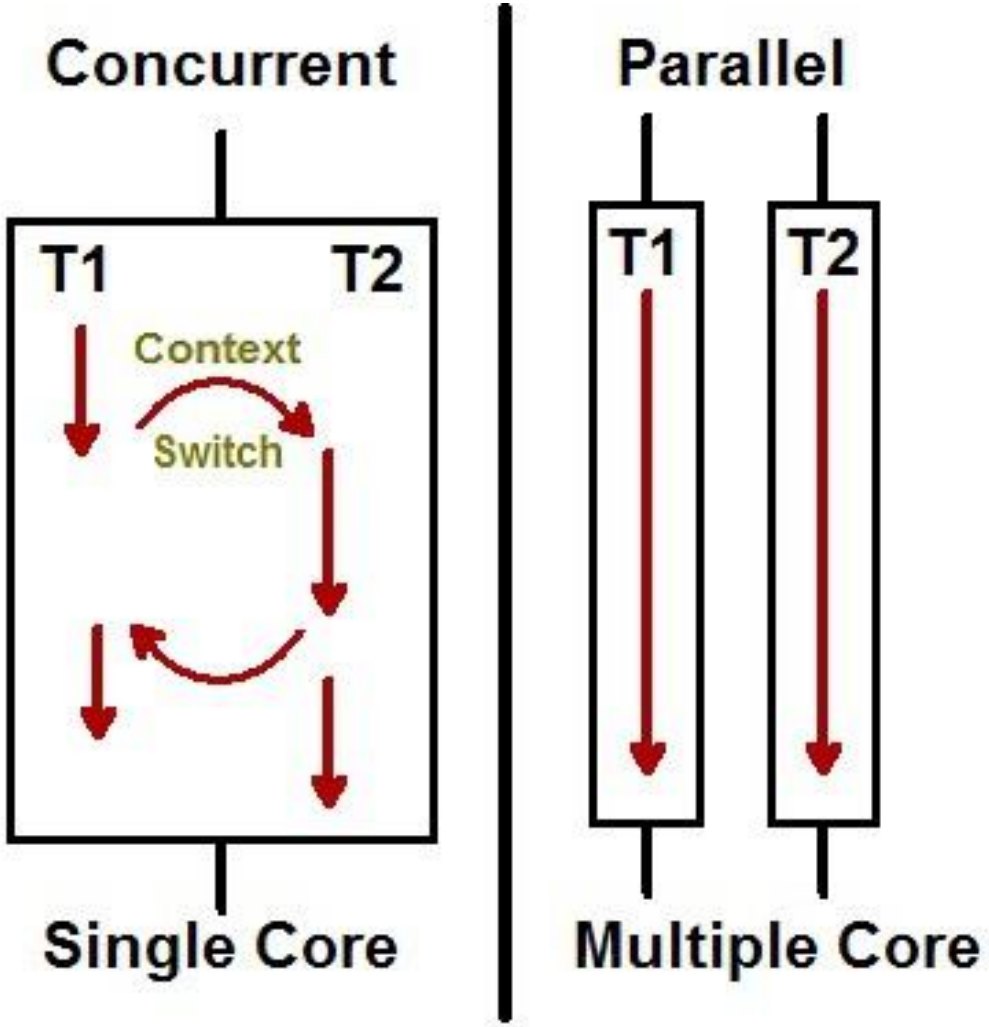
并发 v.s. 并行



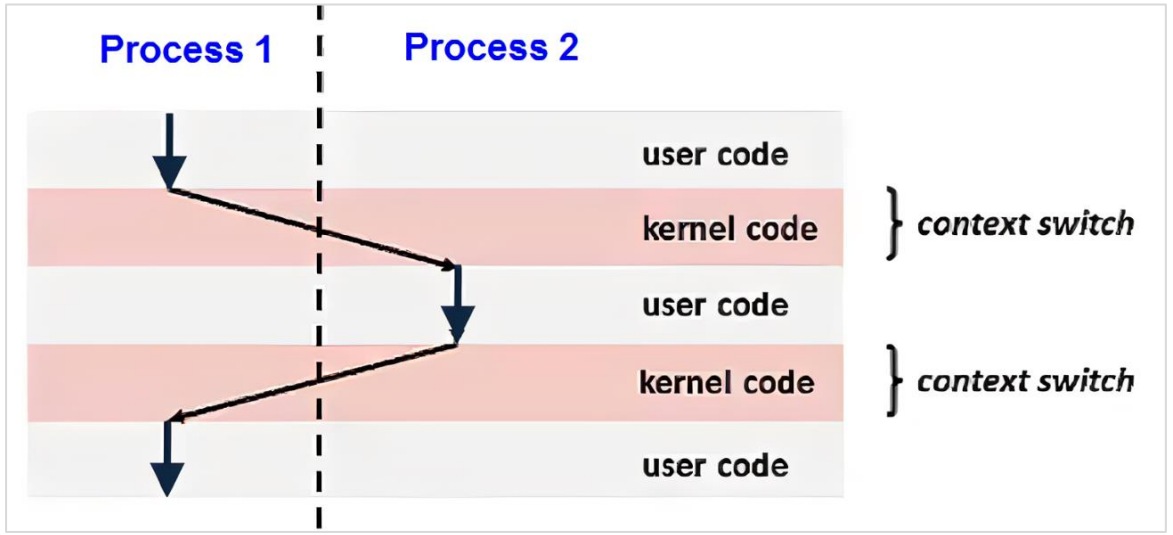
图片来源：https://techdifferences.com/difference-between-concurrency-and-parallelism.html

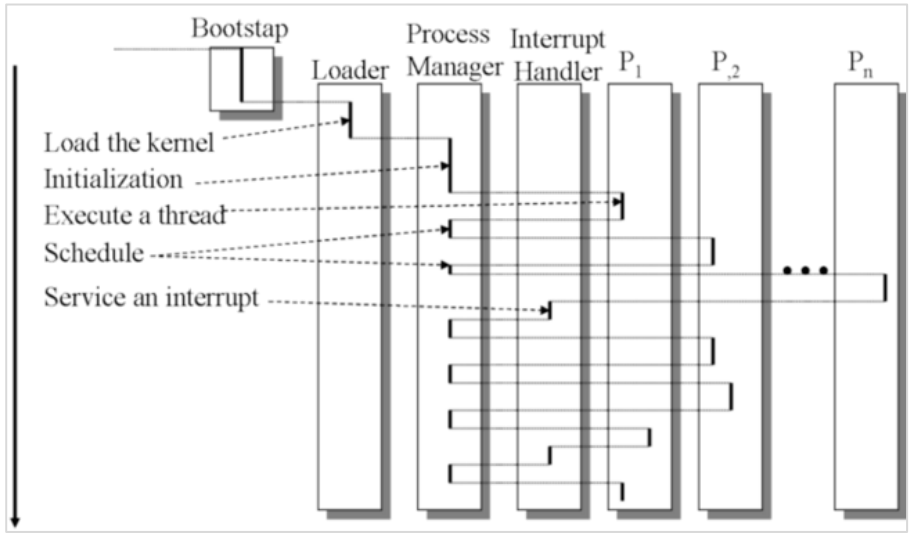
为什么进行CPU调度

并发 v.s. 并行



Context Switch





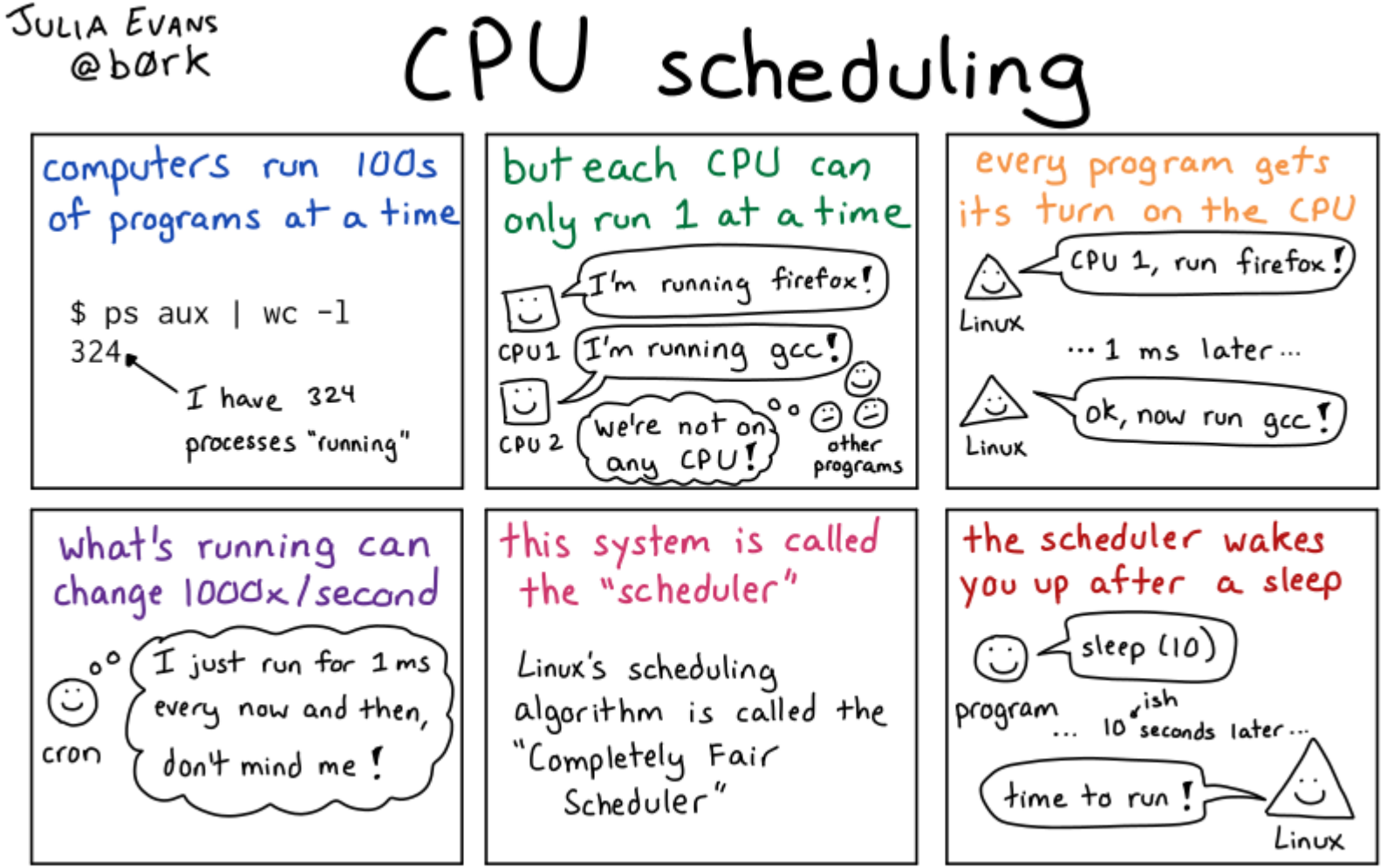
Context switch in OS without thread support

Context switch in OS with thread support

当OS调度器觉得必要时，会执行上下文切换

调度的实质：对CPU资源的虚拟化

调度基本概念


Julia Evans
@b0rk
CPU scheduling
computers run 100s of programs at a time
$ ps aux | wc -l
324
I have 324 processes "running"
but each CPU can only run 1 at a time
I'm running firefox!
CPU 1
I'm running gcc!
CPU 2
We're not on any CPU!
other programs
every program gets its turn on the CPU
CPU 1, run firefox!
Linux
... 1 ms later ...
ok, now run gcc!
Linux
what's running can change 1000x/second
I just run for 1 ms every now and then, don't mind me!
cron
this system is called the "scheduler"
Linux's scheduling algorithm is called the "Completely Fair Scheduler"
the scheduler wakes you up after a sleep
sleep (10)
program
... 10ish seconds later ...
time to run!
Linux

调度开销

调度

Schedule:
从当前就绪队列中选择一个最合适的任务，将其状态转为运行态

CPU调度（短程调度）

派遣

(1) 切换上下文 Switching Context

(2) 将程序从核心态切换回用户态

(3) 跳转到刚转入运行态的任务的下条指令，开始执行

调度基本概念

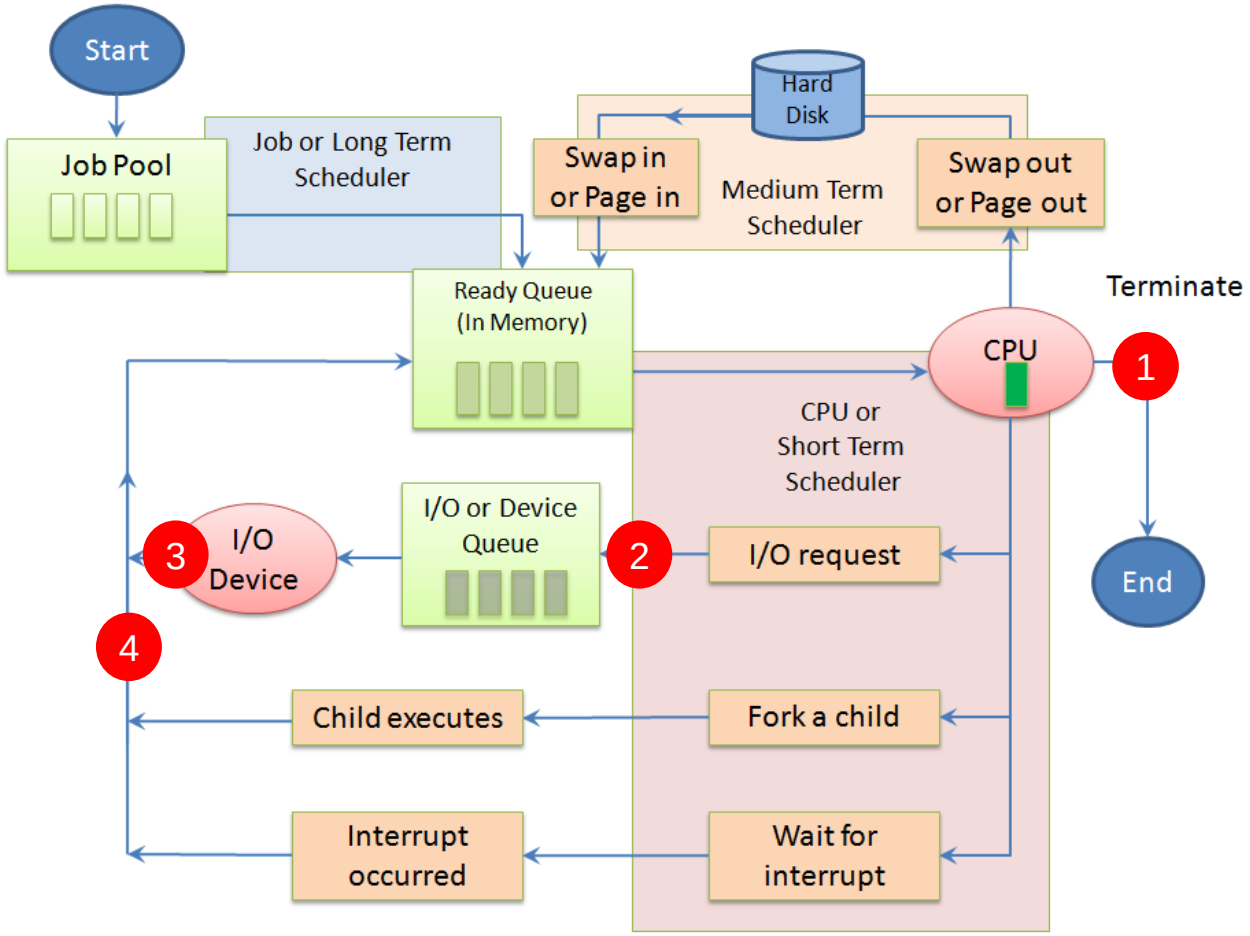
调度开销

目标：调度开销尽量最小化

调度时机

1.进程转入终止态
2.进程阻塞
3.进程被唤醒
4.进程被中断（如定时器中断）

其中，
1，2: 运行态进程主动放弃CPU（yield），
因此属于非抢占点（non-preemptive）
3，4: 属于因中断等事件产生，而可能剥夺
运行进程的CPU时间，因此属于可抢占点
（preemptive）



调度算法评价指标

Evaluation Metrics for CPU Scheduling Algorithms

02

几种典型评价标准

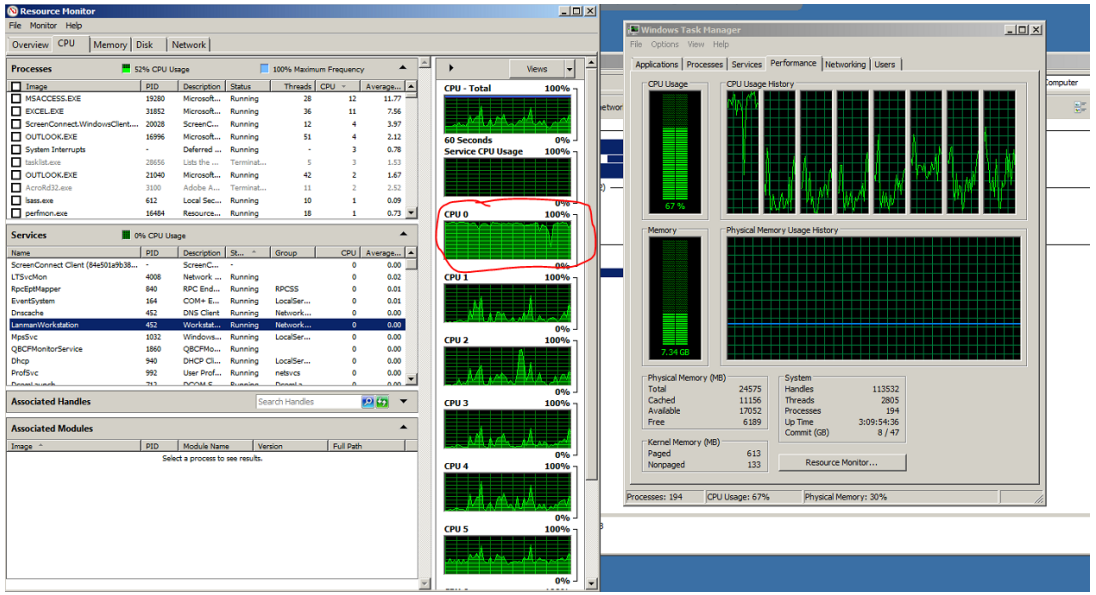
- **软件定义一切的时代，操作系统已成为智能化系统的底座**
  - 无处不在的操作系统，其类型也日益多样
  - 支撑不同应用场景的操作系统，对任务的调度要求也不尽相同

调度评价标准：

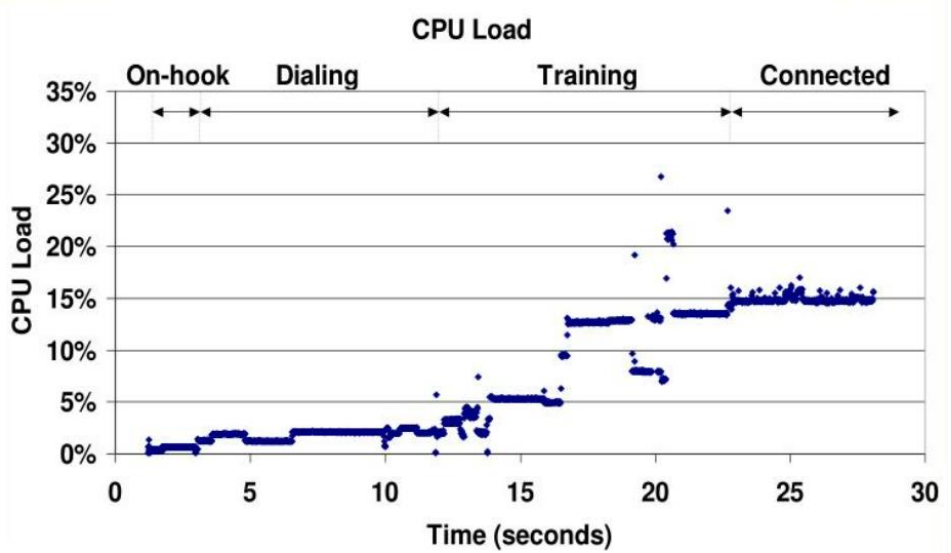
- CPU利用率
- 吞吐率
- 周转时间
- 等待时间
- 响应时间

CPU利用率

- **CPU Utilization (CPU利用率)**
  - 尽量减少CPU空闲的时间
  - 每当有CPU空闲时，尽量都让另一个就绪任务顶上



High CPU load at CPU0 (VMWare task)



CPU Utilization (Soft Modem)

吞吐率

- **Throughput (吞吐率)**
  - 单位时间内完成的进程（任务）数
  - 在这种标准下，调度算法优先选择短进程，保证完成任务的数量
  - 算法实例：短作业优先调度算法（后续有专题具体讨论）

$$Throughput = \frac{number\ of\ processes\ finished}{total\ execution\ time}$$

例如：某计算机完成了10道作业，共用了100秒，
则系统吞吐量为 10/100 = 0.1道/秒

类比：竞赛时，相同时间内，哪个队完成题目数量多，谁就更胜一筹
所以，竞赛队伍在进行任务调度时，以什么为目标?
=> 吞吐率

周转时间

- **周转时间 （Turnaround Time）**
  - 进程从被创建，到执行完毕退出的时间跨度长度
  - 若以此为标准，调度算法会考虑进程的平均周转时间
  - 平均周转时间越短，说明进程的总体执行效率较高，也就表明调度算法较优

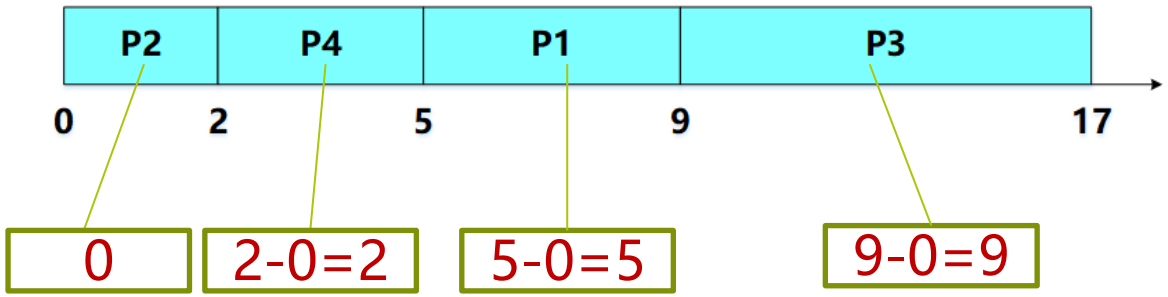
周转时间 = 进程完成时间 – 进程提交时间

等待时间

- **等待时间 （Waiting Time）**
  - 进程建立后等待被服务的时间
  - 调度算法可以将进程平均等待时间作为标准
  - 等待时间越短，调度算法效果越好

How much time processes spend in the ready queue waiting their turn to get on the CPU

- 调度甘特图示例：P1,P2,P3,P4进程，到达时间均为0

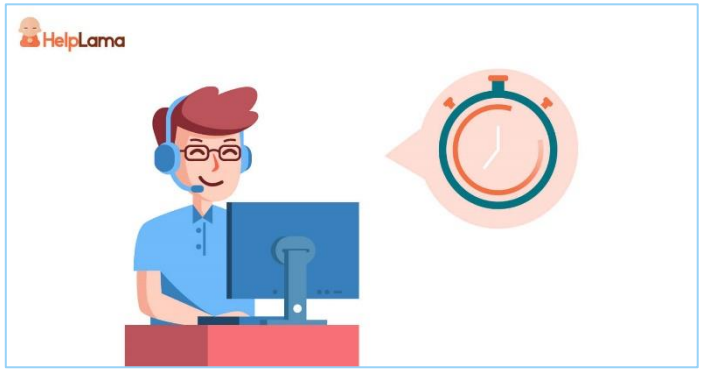


响应时间

- **响应时间 （Response Time）**
  - 从发出服务请求，到请求被满足的时间跨度
  - 响应时间是分时系统中的关键调度指标
  - 通过控制响应时间在较小的范围内，从而保证不同用户的交互操作能够及时得到响应，保证用户公平地分享分时操作系统所管理的软硬件资源

例如：浏览器中输入一个网址后，等了1秒钟，网页内容渲染完毕用户可以开始浏览，系统对此浏览事件的响应时间=1秒

Email Response Time



CPU调度算法1

CPU Scheduling Algorithms: Part 1

03

4种调度算法

一、FCFS调度算法

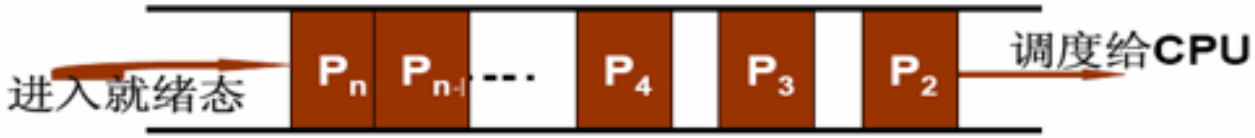
二、SJF

三、SRTF

四、优先级调度

FCFS算法

- **先来先服务（First Come, First Serve）**
- FCFS调度实现方式
  - 用一个FIFO队列来维护就绪进程
  - 每次从FIFO队列取队首进程，将其投入运行
  - 新进入就绪态的进程放在队尾



P1运行完毕后，将P2调度到CPU上，
新的进程P_{n+1}进入队列尾部

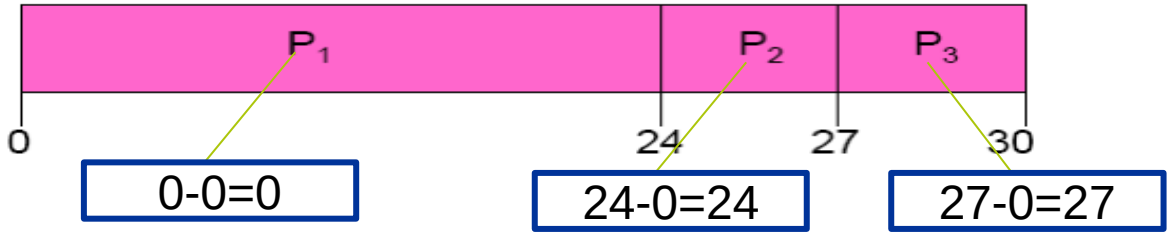
FCFS算法

- **FCFS算法调度示例**

进程P1,P2,P3的顺序在时刻0依次到达

进程	CPU Burst time
P1	24
P2	3
P3	3

通过FCFS算法得到的Gantt图



平均等待时间: (0 + 24 + 27)/3 = 17

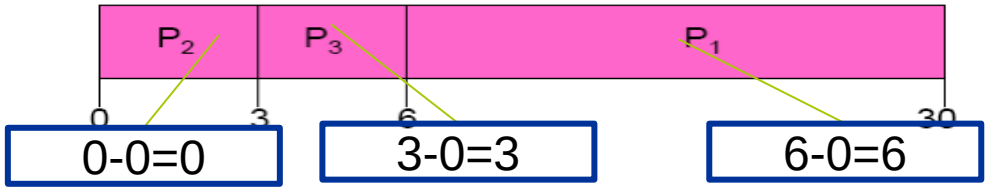
FCFS算法

- FCFS算法调度示例

进程	CPU Burst time
P1	24
P2	3
P3	3

假设3个进程的到达顺序改为P2,P3,P1(时刻0依次到达)

通过FCFS算法得到的Gantt图



平均等待时间: 6 + 0 + 3)/3 = 3

Convoy Effect:
FCFS算法不稳定，长进程先于短进程到达，会导致平均等待时间拉长

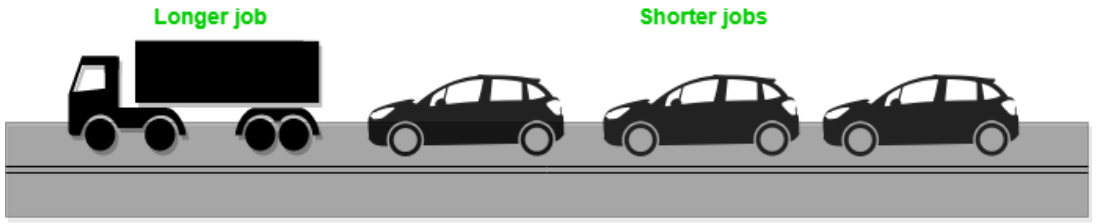
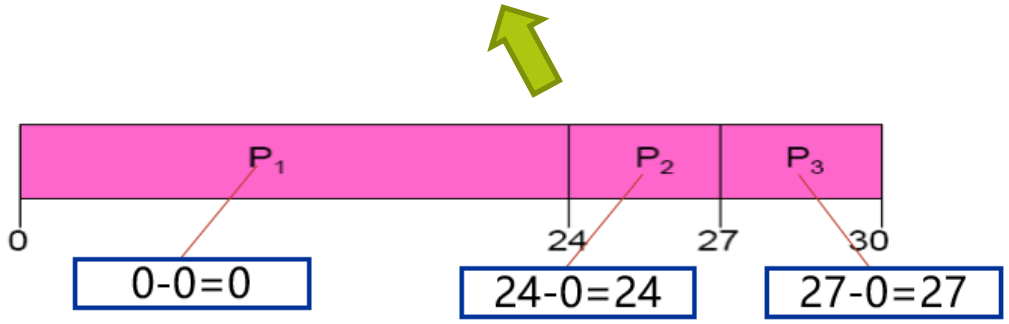


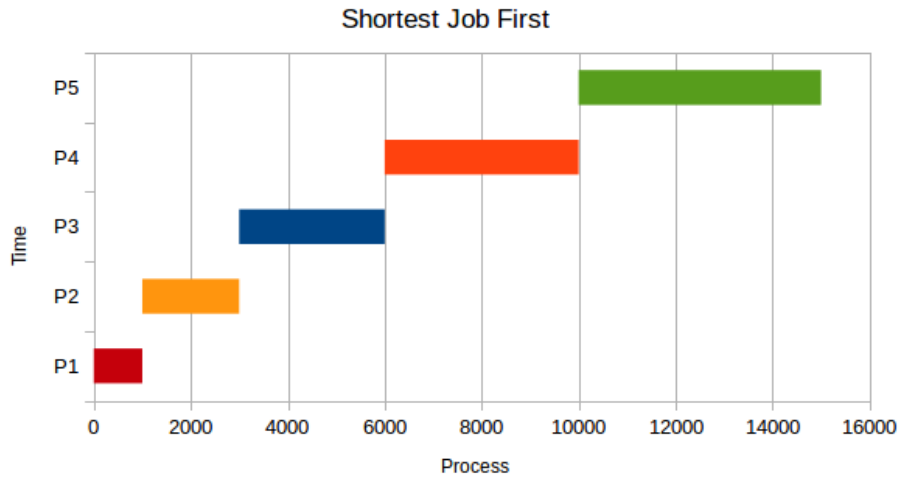
Figure - The Convey Effect, Visualized



平均等待时间: (0 + 24 + 27)/3 = 17

SJF调度算法

- **短作业优先（Shortest Job First）**
  - 每次进行调度时，优先选择下一个CPU周期最短的进程
  - 调度重要信息：每个进程的下一个CPU周期长度
  - 以平均等待时间为指标，SJF是最优的调度



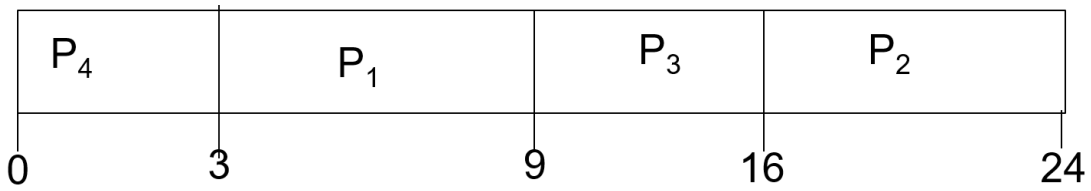
SJF调度目标：maximize throughput

SJF调度算法

- **实例:**

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

- SJF调度甘特图



平均等待时间 = (3 + 16 + 9 + 0) / 4 = 7

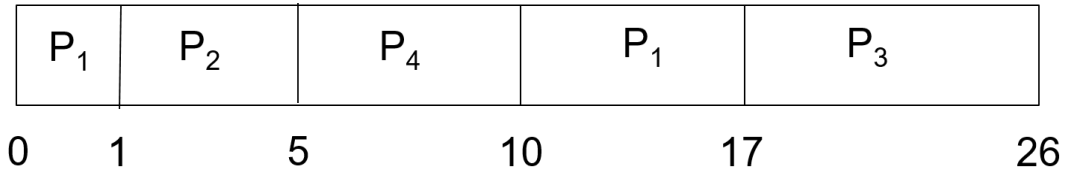
思考并抢答：SJF调度存在的主要问题是什么?

SRTF调度算法

- **最短剩余时间优先（Shortest Remaining Time First）**

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- SRTF的调度甘特图



平均等待时间= [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5

SRTF调度可被视为SJF的抢占式版本

优先级调度算法

- 优先级调度：
  - 每个进程被赋予一个优先数（Priority Number）
  - 每次调度时，优先级最高的任务被选中执行

概念辨析

优先级与优先数的关系视系统而定：
有的系统（如Linux）优先数越小，优先级越高；
有的系统优先数越大，优先级越高

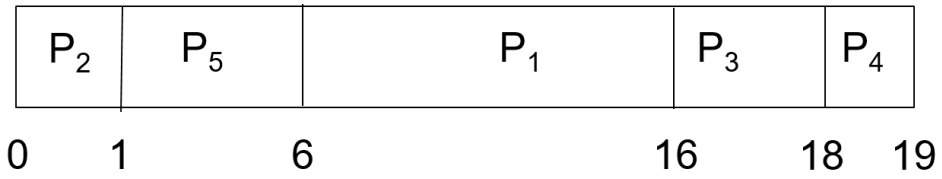
优先级调度又分为：
- 抢占式优先级调度（Preemptive priority scheduling）
- 非抢占式优先级调度（Non-preemptive priority scheduling）

优先级调度算法

- **实例**

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

- **优先级调度的甘特图**



思考并抢答：优先级调度存在的主要问题是什么？

Summary

小结：

- CPU调度概念
- 调度算法评价指标
- 调度算法1：FCFS、SJF、SRTF、优先级调度

讨论

SJF调度算法的背景

批处理操作系统

单选题 2分

设置

下列选项中，降低进程优先级的合理时机是（ ）。

A 进程时间片用完

B 进程刚完成I/O，进入就绪队列

C 进程长期处于就绪队列

D 进程从就绪态转为运行态

提交

单选题 3分

设置

以下有关抢占式调度的论述，错误的是（ ）。

A 可防止单一进程长时间独占CPU

B 进程切换频繁

C 系统开销小

D 调度程序可根据某种原则暂停某个正在执行的进程，将已分配给它的CPU重新分配给另一进程

提交

单选题 3分

设置

下列选项中，满足短任务优先且不会发生饥饿现象的调度算法是（ ）。

A 先来先服务

B 高响应比优先

C 时间片轮转

D 非抢占式短任务优先

提交

进程、线程与CPU调度的关系。

讨论

优先级调度的饥饿问题

优先级调度中的饥饿问题如何解决。

SRTF示例

SRTF调度示例中的流程（带队列的视角）。

- **最短剩余时间优先（Shortest Remaining Time First）**

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- SRTF的调度甘特图

平均等待时间= [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5

SRTF调度可被视为SJF的抢占式版本

谢 谢!
Thank you!