

Process Scheduling — Chapter 5

2022年10月



薛哲

School of Computer Science (National Pilot Software Engineering School)



北京邮电大学

CHAPTER OBJECTIVES

- To introduce CPU-scheduling, which is the basis for multiprogrammed operating systems.
- To describe various CPU-scheduling algorithms.
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system.
- To examine the scheduling algorithms of several operating systems.

Chapter 5 Process scheduling

I. 5.1 Basic Concept (resource allocation: why, who, when)

5.2 Scheduling criteria:

CPU utilization,
throughput,
turnaround time,
waiting time,
respond time,
(fairness, deadline, ...)

II. How to evaluate scheduling algorithms/methods (good or bad ?)

5.8 Algorithm evaluation

Appendix 5-1
Scheduling in Linux
(O(1), CFS)

VII. Operating-system examples

5.7 Unix, Windows

VIII. Examples and exercises:
操作系统概念（英文）——5-Process
Scheduling-例题与作业

多核多线程编程及性能对比分析

IX. Linux Experiments

III. *5.3 Scheduling_ algorithms*

First-Come First-Served(FCFS)

Shortest-Job-First(SJR)

Priority
Scheduling

preemptive

non-preemptive

Round Robin(RR)

Multiple Queue, Mutiple Feedback Queue

Highest Response Rate First(HRRF)

IV. 5.4 Thread scheduling

two-level sheduling for user threads

V. 5.5 Multiple-processor scheduling (two-level scheduling)

homogeneous, asymmetric,
processor affinity, load balancing

scheduling for multicore processor

VI. 5.6 Real-time CPU scheduling

priority-based and preemptive scheduling

rate monotonic scheduling,
earliest deadline first scheduling(EDF)
proportional share scheduling

5.1 Basic Concepts

- Why scheduling needed
 - ❑ multiple processes concurrently run in systems, a process is not allowed to monopolize (独占) the CPU and other resources such as I/O devices
 - ❑ processes have to multiplex one or more CPU (CPU cores), and occupy CPU for execution in order
 - also for other resources
 - ❑ thus process scheduling, i.e. CPU scheduling and IO scheduling are needed
- CPU scheduling
 - ❑ OS **selects** running entities in main memory (i.e., processes or threads in the ready queue) according to some criteria, **allocates** CPU to the selected running entities, and then **enables** them to run on CPU
 - ❑ selecting + allocating + enabling, in kernel mode
 - ❑ known as short-term scheduling

Basic Concepts

- Process execution cycle
 - ▣ process execution consists of a cycle of CPU execution and I/O wait, i.e., CPU burst and I/O burst
- When a CPU burst occurs, the process is in the **running** state;
- When a I/O burst occurs, the process is in the **waiting** state
- CPU-burst times distribution, Fig.5.2

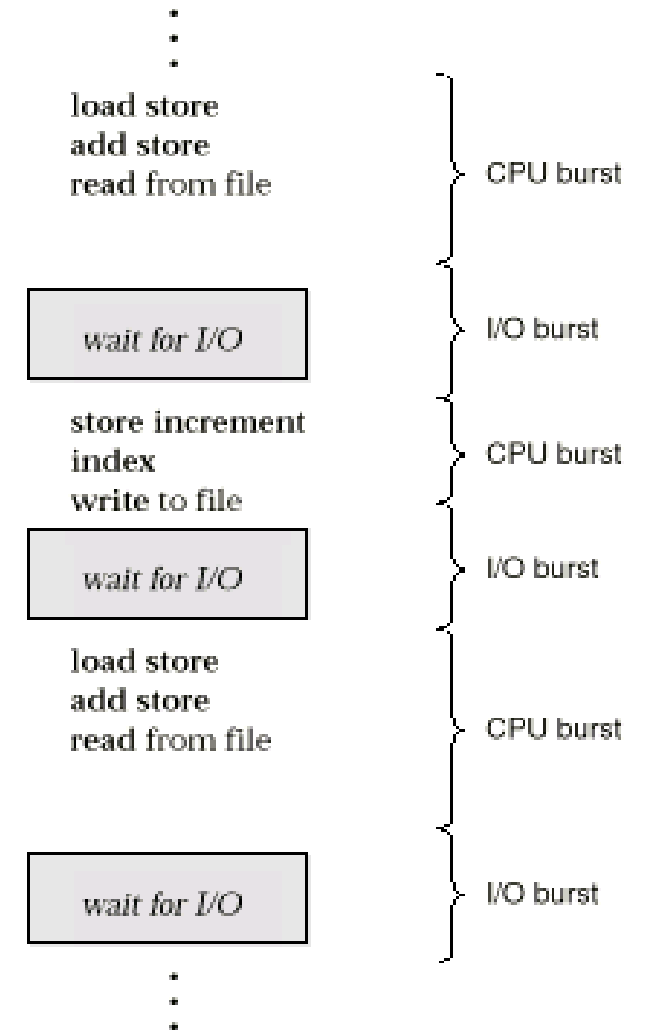


Fig. 5.1 Alternating Sequence of CPU And I/O Bursts

Basic Concepts

- CPU-bound process might have a few very long CPU bursts
 - ▣ computation-intensive, 计算密集型
- When a I/O-bound process might have many very short CPU bursts and some long I/O bursts
 - ▣ I/O密集型, I/O intensive, e.g. DB access

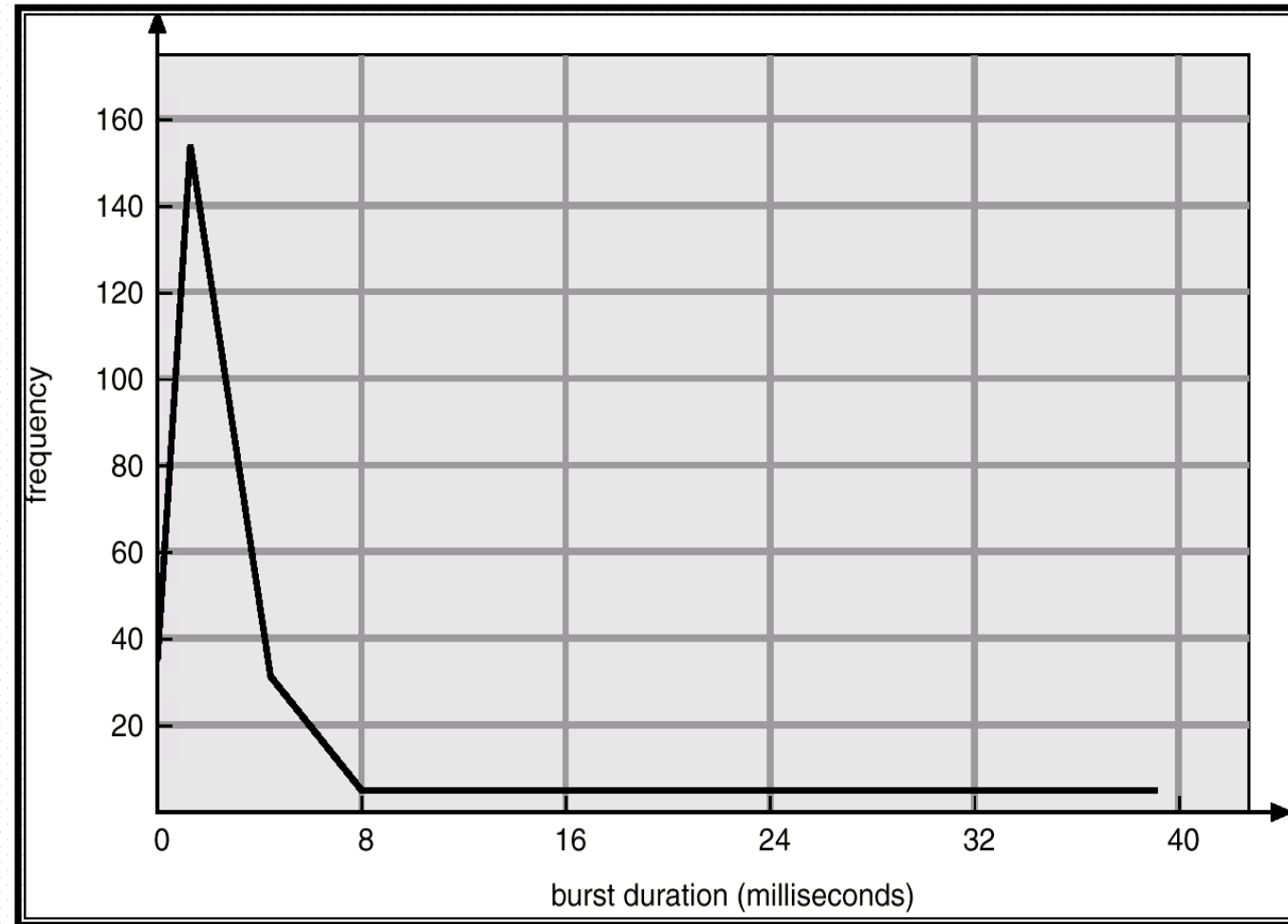



Fig. 5.2 CPU burst distribution
— histogram of CPU- burst Times

Who conduct CPU Scheduling

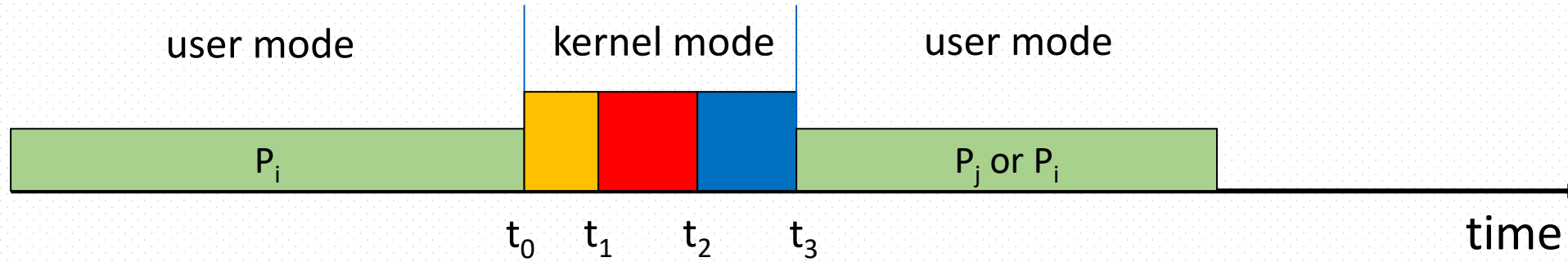
- As components of OS, scheduler (§5.1.2, 调度器) and dispatcher (§ 5.1.4, 分配器) are responsible for CPU scheduling
 - ▣ Linux: kernel/sched.c
- The (short-term) **scheduler** is responsible for CPU scheduling
 - ▣ selects from the processes in memory that are ready to execute
 - using **scheduling algorithms**
 - ▣ allocates the CPU to one of them
 - *组织和维护(就绪)进程/线程队列
- The **dispatcher** gives control of the CPU to the process selected by the scheduler, i.e. starts the selected process
 - ▣ process context switching
 - ▣ switching to user mode
 - ▣ jumping to the proper location in the user program to restart that program

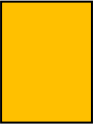
When scheduling occurs

- When CPU is **idle** or some **events occurs** in systems (e.g. real-time system), the OS scheduler should take the control of CPU , and then make scheduling decisions
 - process terminates
 - the running process switches from running to waiting state (for example, I/O requests)
 - the running process switches from running to ready state (e.g., interrupts occurs, or the timeslot is out)
 - (maybe) switches from waiting to ready (e.g. , completion of I/O)
 - Scheduling under condition 1 and 2 is nonpreemptive (非抢占式), because the CPU becomes in idle
 - All other scheduling is preemptive (抢占式) , often used in real-time systems
- 

CPU mode/process context switching in scheduling

- CPU scheduling results in CPU switching between user mode and kernel mode



 : system calls, interrupt handling ,etc., resulting in process context saving
, under four circumstances (as described in the textbook)

 : scheduler,  : dispatcher

Fig. 5.0.1 Illustration of **mode switching** in CPU scheduling

CPU mode/process context switching in scheduling

- Dispatch latency (调度等待时间)
 - ▣ time it takes for the dispatcher and scheduler to stop one process and start another running, i.e. $[t_0, t_3]$ in the figure
 - ▣ in kernel mode
- 资料：Linux 的调度延迟 - 原理与观测
 - ▣ <https://zhuanlan.zhihu.com/p/462728452>
- 资料：Linux任务调度延时分析工具getdelays
 - ▣ <https://www.cnblogs.com/liuhailong0112/p/15379397.html>

 **Linux 的调度延迟 - 原理与观测 - 知乎**

<https://zhuanlan.zhihu.com/p/462728452> ▼

2022-3-5 · 所谓「调度延迟」，是指一个任务具备运行的条件（进入 CPU 的 runqueue），到真正执行（获得 CPU 的执行权）的这段时间。那为什么会有调度延迟呢？因为 CPU 还被其他任务...

 **[linux][bcc]使用runqslower发现调度延迟问题 - 云+社区 - 腾讯**

<https://cloud.tencent.com/developer/article/1871858> ▼

2021-9-2 · 分析 调度延迟 在前文《[Linux][kernel] sched delay和steal time的原理分析以及atop的监控改进》中分析过Linux中如何计算一个task的run delay：即一个task希望运行，但是得不到运...

 **Linux任务调度延时分析工具getdelays - 温暖的电波 - 博客园**

<https://www.cnblogs.com/liuhailong0112/p/15379397.html> ▼


2021-10-10 · 因此，从原理上来说task->utime和task->stime实际上是对任务运行时间的一个采样统计方式。virtual total：来自于内核中任务调度实体的task->se.sum_exec_runtime字段，这个...

5.2 Scheduling Criteria

- Scheduler aims to optimize criteria

- maximize **CPU utilization**
 - maximize **throughput (吞吐量)**
- 

- system level criteria, specific to the whole processes in system

- minimize **turnaround time (周转时间)**
 - minimize **waiting time**
 - minimize **response time**
- 

- process level criteria, specific to a particular process

- Others

- 公平性, 资源利用率, 实时性, 能耗

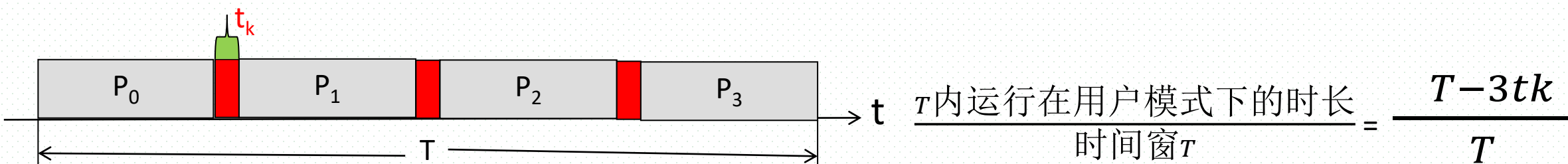
CPU utilization

■ CPU utilization

- keep the CPU as busy as possible
- keeping the CPU in user mode as possible, refer to Fig. 5.0.1, CPU运行在用户模式下的时间占比

$$\frac{\text{amount of time during which \textbf{user} processes run on CPU}}{\text{amount of time during which \textbf{user} and \textbf{OS} processes run on CPU}} \times 100\%$$

用户应用程序, 非OS的系统程序 (e.g. 编译程序)



例题：CPU utilization

- 在长度为100ms的时段T内，CPU分别运行
 - ▣ 用户应用进程 P_1 45ms, 用户应用进程 P_2 35ms
 - ▣ 编译进程 P_3 12ms, 链接/装载进程 P_4 5ms,
 - ▣ OS 调度进程 P_5 2ms, OS内存分配进程 P_6 1ms

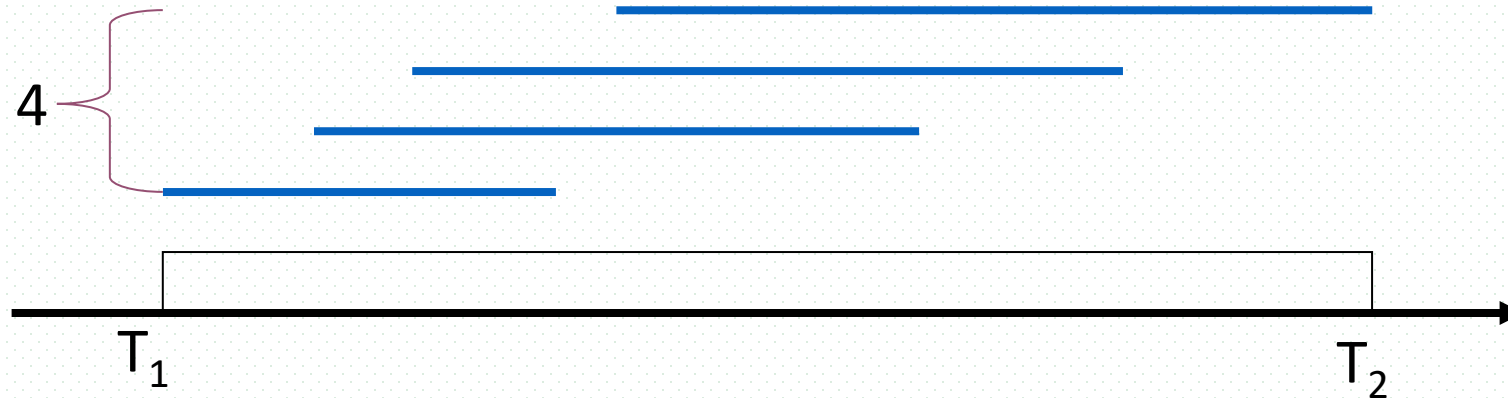
则时间段T内的CPU利用率为：

- ▣ A. 80% B. 97% C. 98 % D. 99%

- ▣ 答案： B

Throughput

- Throughput (吞吐量)
 - ▣ the number of processes that complete their execution **per time unit**
- E.g.
throughput
= number of processes terminated in $[T_1, T_2]$, e.g. 4
/ length of $[T_1, T_2]$



Turnaround Time, Waiting Time

- Turnaround time (周转时间)
 - ▣ amount of time taken to execute/complete a particular process
 - ▣ i.e. the length of the process lifetime cycle, the time the process resides in ready, running, waiting, andstates
- Waiting time
 - ▣ the sum of the periods spent waiting in the ready queue
 - ▣ i.e. amount of time a process has been waiting in the ready queue

Turnaround Time, Waiting Time

- **turnaround time:** $[t_0, t_8]$
- **waiting time:** $[t_0, t_1] + [t_3, t_4] + [t_6, t_7]$

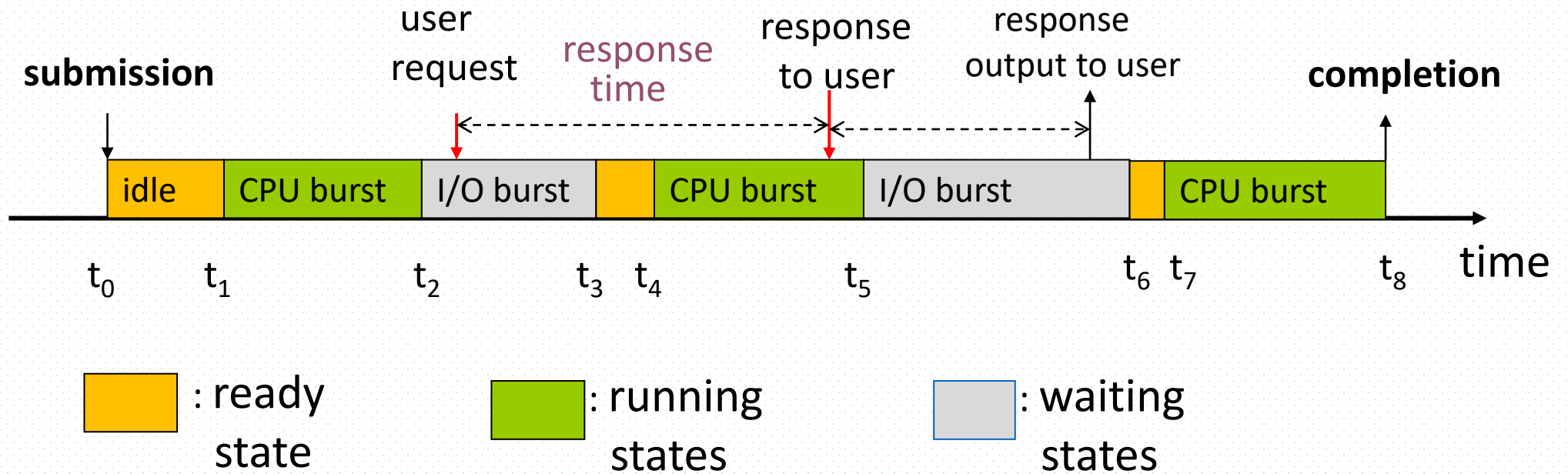
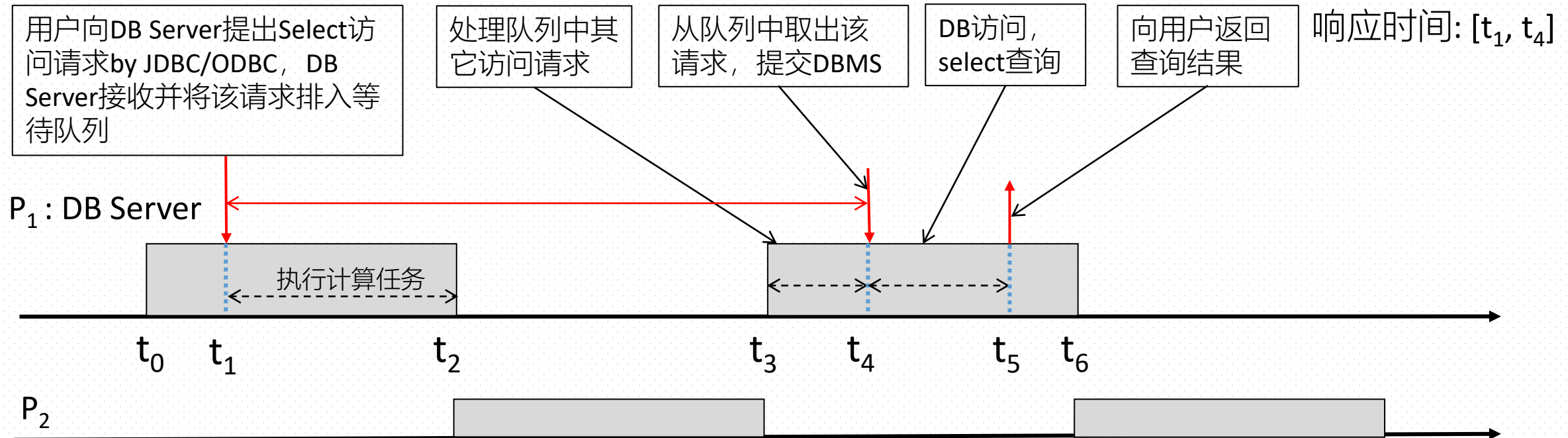


Fig.5.0.2 lifetime cycle of a process

Response Time

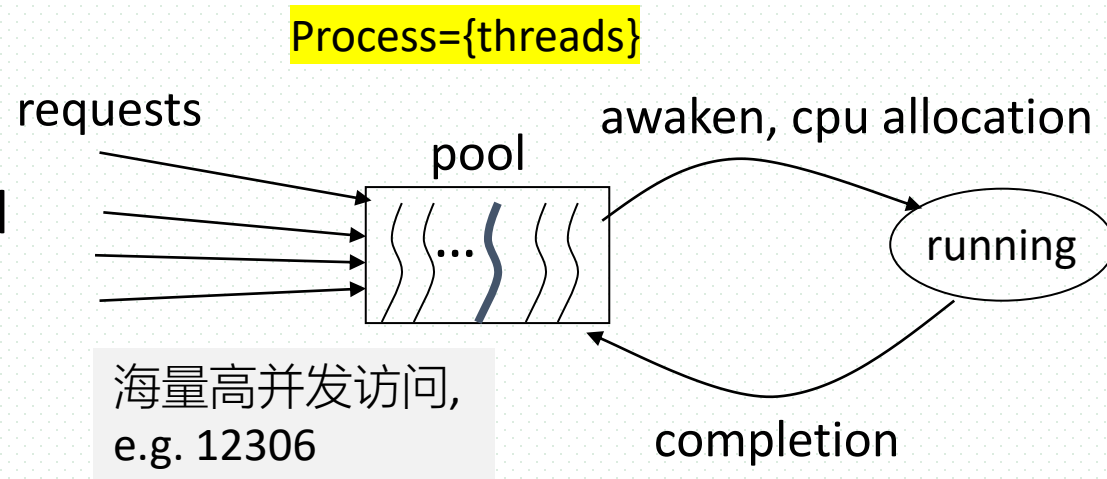
■ Response time (响应时间)

- ❑ in interactive systems, the time from the submission of a request until the first response is produced
- ❑ the time it takes to **start responding**, not the time it takes to output the response
- ❑ E.g. 1 query request in DBS



Response Time

- E.g. 2 A service request arrives at the thread pool
 - ▣ the response time is the interval from the time when the request arrives to that at which a thread is selected to serve the request



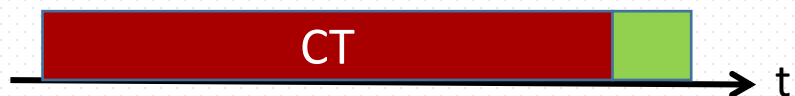
Response Time: threads in pool

当 $L > M$ 时, e.g. $L=5, M=4$, 部分用户请求request的响应时间较大

- 根据服务器端的CPU核的数目、线程需要执行的任务/进程类型, 确定线程数目
- 任务类型

- CPU密集型, CPU-intensive, CPU-bound

- e.g. 科学计算, 矩阵计算

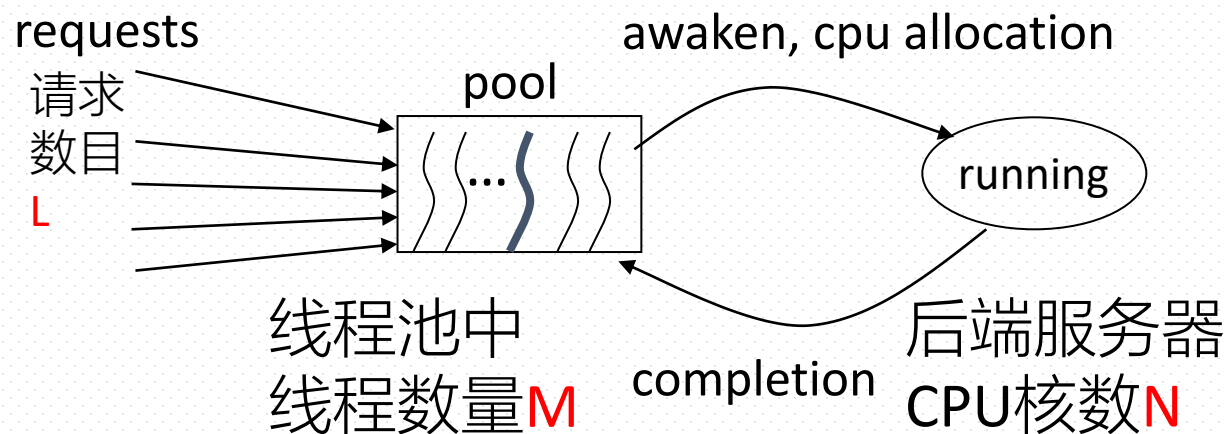


- I/O密集型, I/O-intensive, I/O-bound

- e.g. 数据库访问磁盘I/O, 网络I/O



海量高并发访问



M设置方式

- CPU密集型任务:

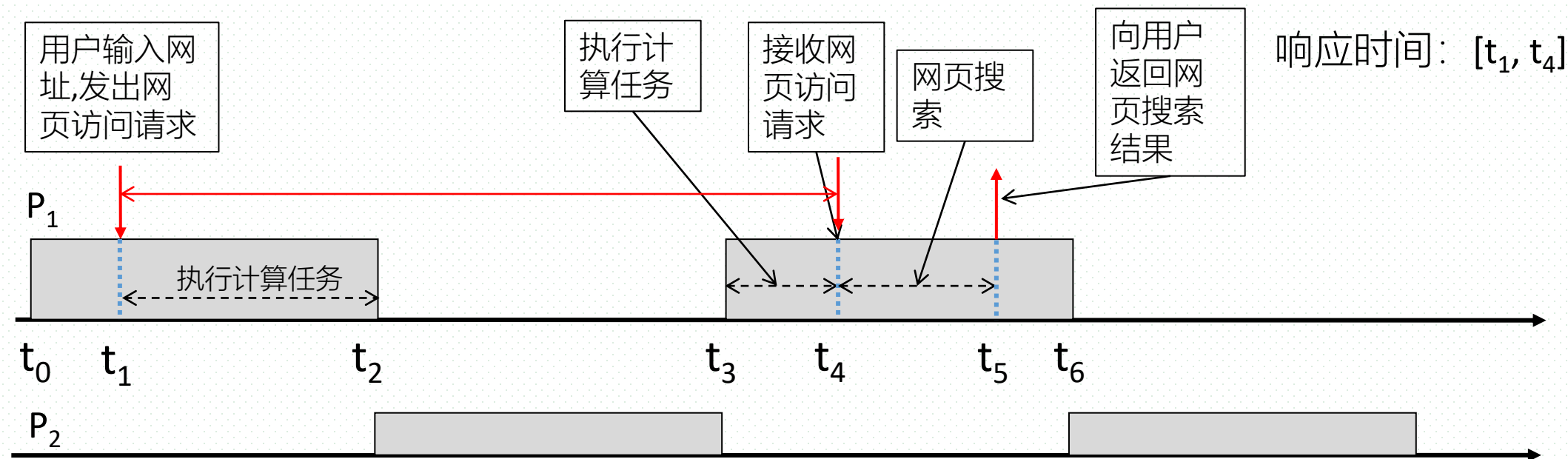
$$M=N$$

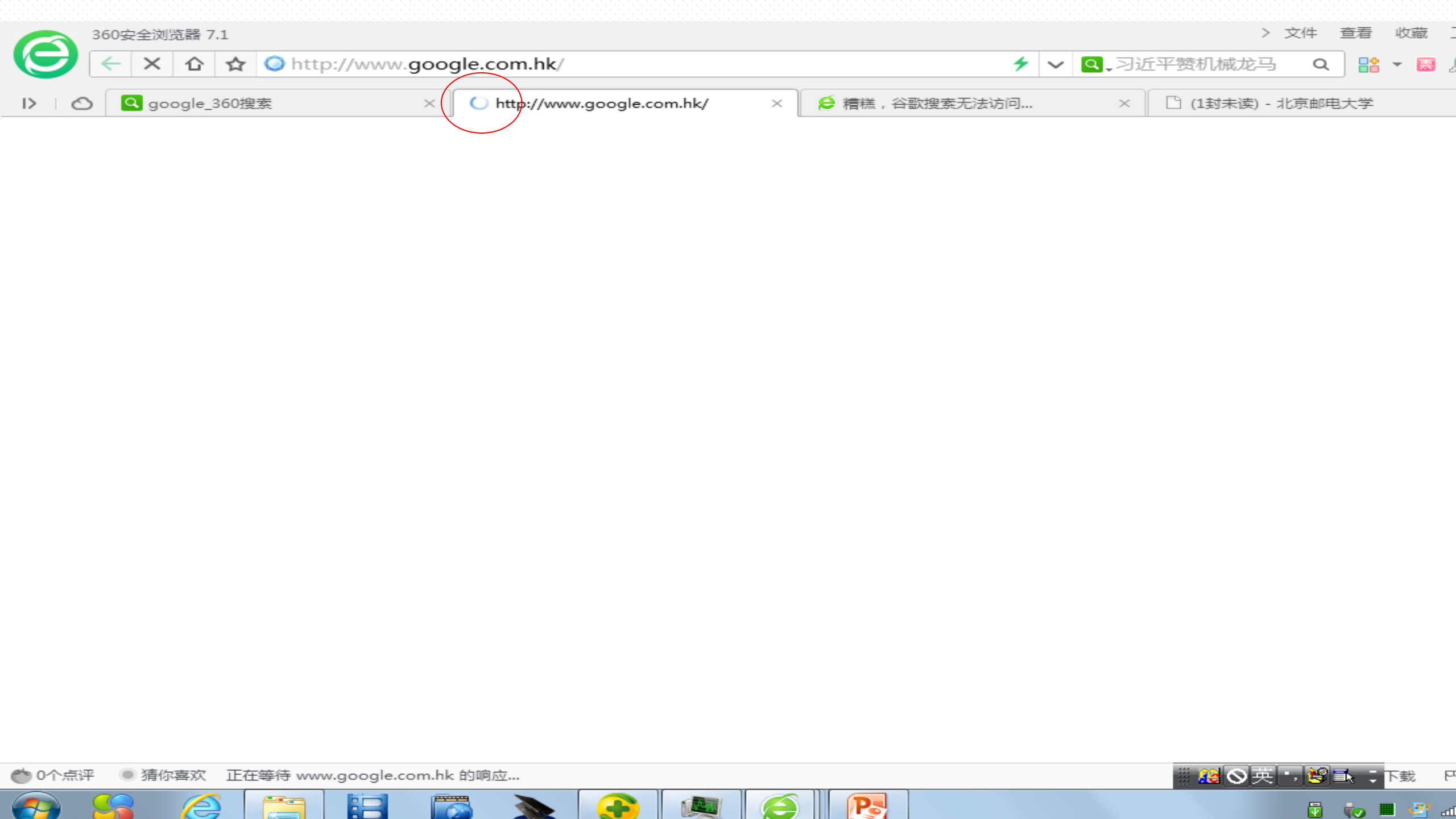
- I/O密集型任务:

$$M = N * (1 + WT/CT)$$
$$= \text{e.g. } 4 * (1 + 75/25)$$

Response Time

■ E.g. 3 Web search





糟糕!谷歌搜索无法访问...

请您稍后再试，或选择切换到其他搜索引擎：

切换到：

360 搜索+

Bai 百度

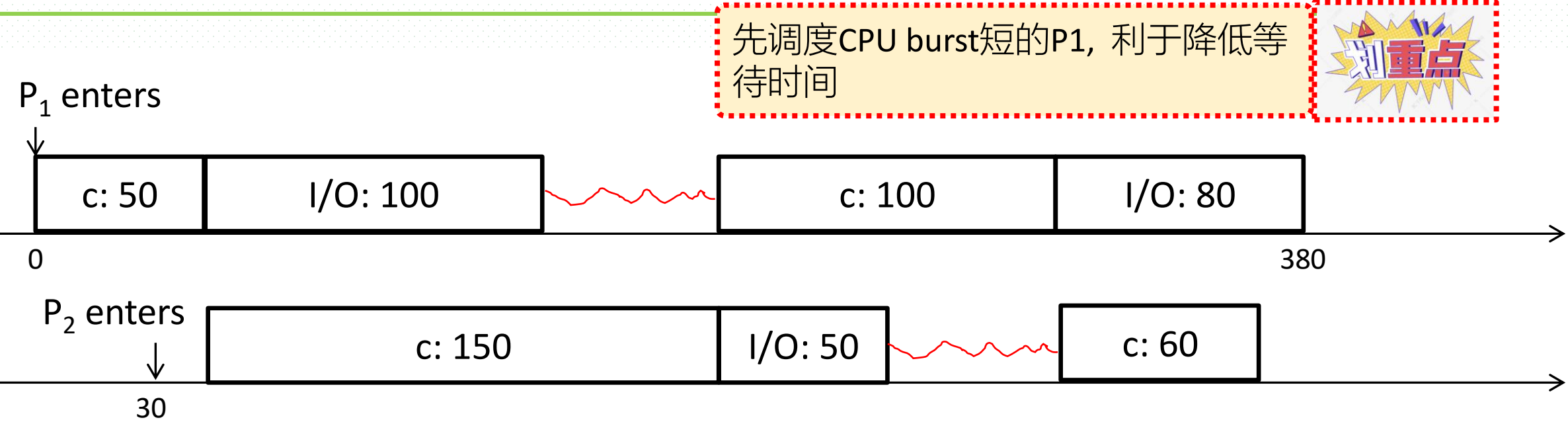
必应



Example

- In a multiprogramming batch system, there are two processes P_1 and P_2 concurrently running on a single CPU. P_2 enters into the system 30ms later than P_1 , and their executing traces, i.e. alternating sequences of CPU bursts and I/O bursts, are as follows
 - ▣ P_1 : computing, 50ms \rightarrow I/O operation, 100ms \rightarrow computing, 100ms
 \rightarrow I/O operation, 80ms
 - ▣ P_2 : computing, 150ms \rightarrow I/O operation, 50ms \rightarrow computing, 60ms
- It is assumed that time costs of CPU scheduling and process switch are omitted, illustrate the executing traces of P_1 and P_2 by charts, and compute
 - ▣ 1) the maximal throughput for completing these two processes
 - ▣ 2) the waiting times for P_1 and P_2 respectively

Example



- $\text{throughput} = 2 / (50 + 150 + 100 + 80) = 2 / 380 \text{ms}$
- **waiting** time for P₁ : 50ms
- **waiting** time for P₂ : $(50 - 30) + 50 = 70 \text{ms}$

扩展：多核多线程， 3个进程， 2个CPU

5.3 Scheduling Algorithms

- First-Come First-Served (FCFS)
- Shortest-Job-First (SJR)
- Priority scheduling
- preemptive, nonpreemptive
- Round Robin (RR, 时间片轮转)
- Multilevel queue
- Multilevel feedback queue

- Highest Response Rate First (HRRF)
- Scheduling in real-time system

First-Come First-Served (FCFS)

- Principle

- ▣ the process that requests the CPU first is allocated the CPU first

- Implementation

- ▣ the new created process is inserted into a **FIFO ready queue**, i.e. inserted into the tail of the ready queue
- ▣ scheduler selects **the first** process in the ready queue

Example

- There are three processes P_1 , P_2 , P_3 , the CPU bursts of each process are

Process (CPU and I/O) Burst Time

P_1 24

P_2 3

P_3 3

- suppose that the processes arrive in the order: P_1 , P_2 , P_3

The **Gantt Chart** for the schedule is:



- waiting time for $P_1 = 0$, $P_2 = 24$, $P_3 = 27$
- average waiting time: $(0 + 24 + 27)/3 = 17$

Example

- suppose that the processes arrive in the order

P_2, P_3, P_1 .

- the Gantt chart for the schedule is:



- waiting time for $P_1 = 6$, $P_2 = 0$, $P_3 = 3$
- average waiting time: $(6 + 0 + 3)/3 = 3$
 - much better than previous case, because the processes with shorter bursts run at first

First-Come First-Served (FCFS)

- FCFS Features

- FCFS is nonpreemptive
- the average waiting time under FCFS is generally not minimal, and may vary substantially if the process CPU-burst times vary greatly
- FCFS **prefers to** the processes with shorter CPU-burst
 - 优先调度the processes with shorter CPU-burst将获得较好的性能

Shortest-Job-First (SJR) scheduling

■ Principles

- ❑ associate with each process the length of its **next CPU burst**
- ❑ use these lengths to schedule the process with the shortest time, i.e., allocate CPU to the process with the **smallest** next CPU burst
- ❑ if two processes have the same next CPU bursts, FCFS scheduling is used.

■ Implementation

- ❑ Nonpreemptive
 - once CPU is given to the process, it cannot be preempted until completes its CPU burst
- ❑ Preemptive
 - if a new processs (**as an external event**) arrives with CPU burst length less than remaining time of current executing process, preempting and scheduling occurs, according to the remaining burst of the preempted process

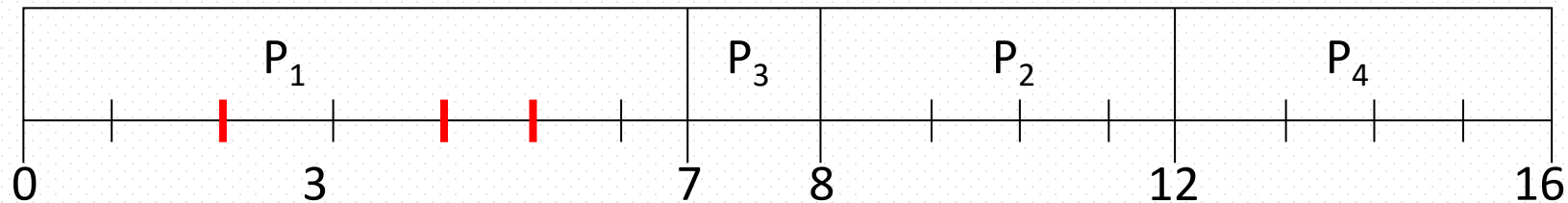
Shortest-Job-First (SJF) scheduling

- Preemptive SJF is known as the Shortest-Remaining-Time-First (SRTF)
- Feature
 - with respect to **minimum average waiting time** for a given set of processes, SJF is optimal

Example

- E.g. SJF (non-preemptive)

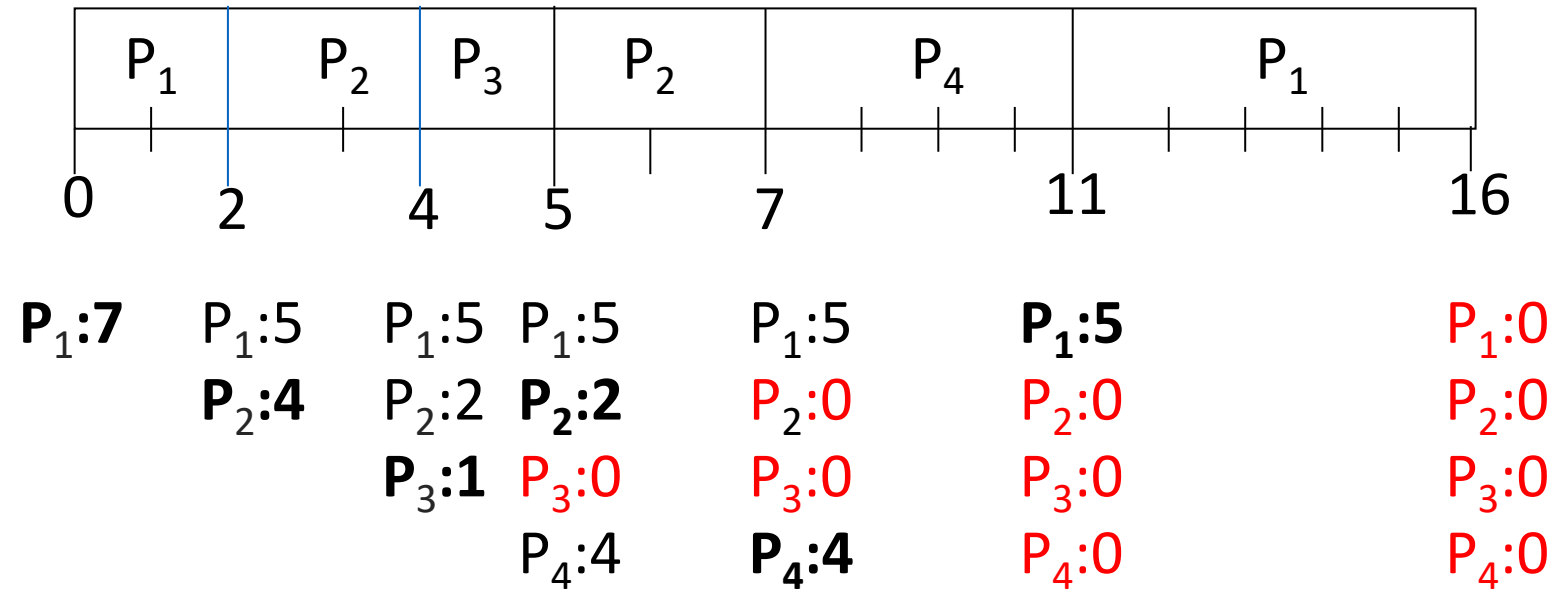
<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P ₁	0.0	7
P ₂	2.0	4
P ₃	4.0	1
P ₄	5.0	4



□ average waiting time = $(0 + 6 + 3 + 7)/4 = 4$

■ E.g. SJF (preemptive)

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P ₁	0.0	7
P ₂	2.0	4
P ₃	4.0	1
P ₄	5.0	4



□ average waiting time = $(9 + 1 + 0 + 2)/4 = 3$

Shortest-Job-First (SJR) scheduling

- How to determining the length of next CPU burst
 - ▣ can only estimate the length
 - ▣ can be predicted by using the length of previous CPU bursts, using exponential averaging, using the formula given in the textbook
 - ▣ for more details about exponential averaging prediction(指数平均预测), refer to
 - Appendix 5-1 基于指数平均的CPU burst预测 ▶

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$

Priority scheduling

■ Principles

- a priority number (integer) is associated with each process the CPU is allocated to the process with the highest priority
- often, smallest integer \equiv highest priority



■ Implementation

- Preemptive
 - preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process
- Nonpreemptive
 - insert the new process into the head of the ready queue, and wait for CPU scheduling on the basis of priorities

Windows 任务管理器

文件(F) 选项(O) 查看(V) 关机(U) 帮助(H)

应用程序 进程 性能 联网 用户

映像名称	PID	用户名	CPU	内存使用	高峰内存使用	虚拟内存大小	I/O 写入
taskmgr.exe	8108	yewenbupt	05	3,528 K	6,828 K	3,056 K	5
cidaemon.exe	7588	SYSTEM	50	812 K	6,212 K	1,820 K	3
SGTool.exe	6908	yewenbupt	00	13,560 K	13,660 K	9,748 K	36
ieexplore.exe	6828	yewenbupt	00	8,880 K	82,816 K	68,716 K	860
ieexplore.exe	6620	yewenbupt	00	8 K	8 K	78,796 K	2,869
WINWORD.EXE	6556	yewenbupt	00	8 K	8 K	38,680 K	210
MDM.EXE	5992	SYSTEM	00	8 K	8 K	1,304 K	4
SogouCloud.exe	5452	yewenbupt	00	8 K	8 K	6,936 K	374
msmsgs.exe	5340	yewenbupt	00	8 K	8 K	3,556 K	81
SGImeGuard.exe	5244	yewenbupt	00	8 K	8 K	8 K	1
360rp.exe	5200	yewenbupt	00	8 K	8 K	8 K	18,569
POWERPNT.EXE	5120	yewenbupt	00	8 K	8 K	8 K	13
SvcGuiHlpr.exe	4524	SYSTEM	00	8 K	8 K	8 K	8
Acrobat.exe	4484	yewenbupt	00	4,784 K	36,880 K	8 K	548
ieexplore.exe	4172	yewenbupt	00	3,684 K	46,440 K	8 K	969
SUService.exe	4092	SYSTEM	00	192 K	11,160 K	8 K	9
SQLAGENT90.EXE	4036	SYSTEM	00	608 K	12,616 K	8 K	76
msftesql.exe	4000	SYSTEM	00	692 K	4,720 K	8 K	3
tvtsched.exe	3892	SYSTEM	00	664 K	5,460 K	8 K	15
alg.exe	3840	LOCAL SERVICE	00	84 K	3,876 K	8 K	4
rrservice.exe	3808	SYSTEM	00	1,176 K	8,760 K	4,396 K	537
rrpservice.exe	3736	SYSTEM	00	28 K	3,312 K	1,100 K	4
tvttcsd.exe	3684	NETWORK SERVICE	00	28 K	2,272 K	708 K	3
TPHDEXLG.exe	3620	SYSTEM	00	256 K	1,892 K	672 K	20
tvt_reg_monitor_sv...	3476	SYSTEM	00	176 K	3,948 K	1,500 K	2,471
svchost.exe	3456	SYSTEM	00	2,124 K	4,740 K	2,932 K	15
sqlwriter.exe	3404	SYSTEM	00	28 K	3,832 K	1,092 K	15
sqlbrowser.exe	3356	NETWORK SERVICE	00	1,672 K	8,424 K	38,120 K	8
RIService.exe	3324	SYSTEM	00	144 K	2,352 K	612 K	3
RegSrvc.exe	3308	SYSTEM	00	28 K	3,344 K	1,032 K	4
sqlservr.exe	2992	SYSTEM	00	6,460 K	76,076 K	52,092 K	308
sqlservr.exe	2948	NETWORK SERVICE	00	620 K	27,784 K	34,404 K	146
MsDtsSrvr.exe	2768	SYSTEM	00	124 K	15,540 K	17,568 K	7
EvtEng.exe	2700	SYSTEM	00	348 K	13,640 K	9,372 K	9
DkService.exe	2648	SYSTEM	00	1,160 K	10,416 K	6,808 K	13
WPSERVICE.exe	2608	SYSTEM	00	760 K	6,560 K	1,740 K	12
cisvc.exe	2580	SYSTEM	00	968 K	35,988 K	5,792 K	99,990
CAJSHost.exe	2564	SYSTEM	00	28 K	2,620 K	812 K	2
arr_srvs.exe	2528	SYSTEM	00	124 K	20,332 K	3,748 K	7
AcPrfMgrSvc.exe	2480	SYSTEM	00	1,696 K	5,964 K	2,216 K	10
arr_isrv.exe	2448	SYSTEM	00	360 K	4,672 K	2,632 K	5
SoftMgrLite.exe	2224	yewenbupt	00	1,556 K	21,036 K	17,964 K	112

结束进程(E)
结束进程树(T)
调试(D)

设置优先级(P)
关系设置(A)...

实时(R)
高(H)
高于标准(A)
• 标准(N)
低于标准(B)
低(L)

☒ 显示所有用户的进程(S)

Example




敲黑板，重点到了

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u> Number
P ₁	10	3
P ₂	1	1
P ₃	2	4
P ₄	1	5
P ₅	5	2


Smaller priority numbers imply higher scheduling priorities



Priority Scheduling

- Types of priority
 - ▣ static priority
 - the priority remains unchanged during process lifetime
 - ▣ dynamic priority
 - the priority may vary, e.g. in Linux
- How to determine the priority
 - ▣ defined internally
 - set by OS, according to process properties
 - ▣ defined externally
 - can be set and changed by users 
- FCFS and SJF can be viewed as priority-based scheduling
 - ▣ in FCFS, the priority is the waiting time of the process from its being created
 - ▣ in SJF, the priority is the predicted next CPU burst time

Priority Scheduling

- Features
 - ▣ commonly used in modern operating systems
 - ▣ may be indefinite blocking/starvation
- Starvation Problem
 - ▣ low priority processes may never execute
 - ▣ solution: aging
 - as time progresses, increase the priority of the process, i.e. dynamic priority is adopted
- E.g 1. Dynamic priority-based scheduling in Unix
 - ▣ refer to Appendix B Priority-based scheduling in Unix 
- E.g 2. CPU scheduling in Linux
 - ▣ refer to Linux as Case Studies-6-Process scheduling
 - $O(1)$, CFS

Round Robin (轮转调度)

■ Principles

- ❑ CPU time is divided into a series of small time units, called time quantum(量子) or time slice (时间片)
- ❑ each process in ready queues is allocated a time slice, usually 10-100 milliseconds, and begins to run on CPU
- ❑ scheduler selects **the first process (head process)** in the ready queue, this process is allocated a CPU time slice
- ❑ after this time slice has elapsed, the process is preempted and added to **the tail** of the ready queue, and the CPU is allocated to the next process in the ready queue

■ Features

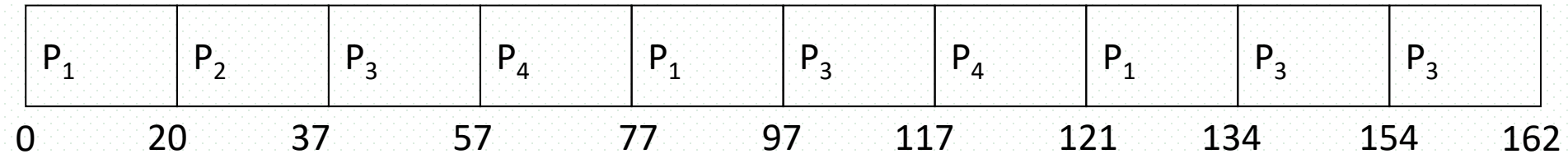
- ❑ typically, higher average **waiting time** than SJF, but better **response time**
- ❑ commonly used in **time-sharing** systems

Example

- E.g. RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P ₁	53
P ₂	17
P ₃	68
P ₄	24

- The Gantt chart is



Time Quantum vs Performance: context switch time

- q large \Rightarrow FCFS
- q small \Rightarrow the number of context switch increase
 - so q must be large with respect to context switch, otherwise overhead is too high

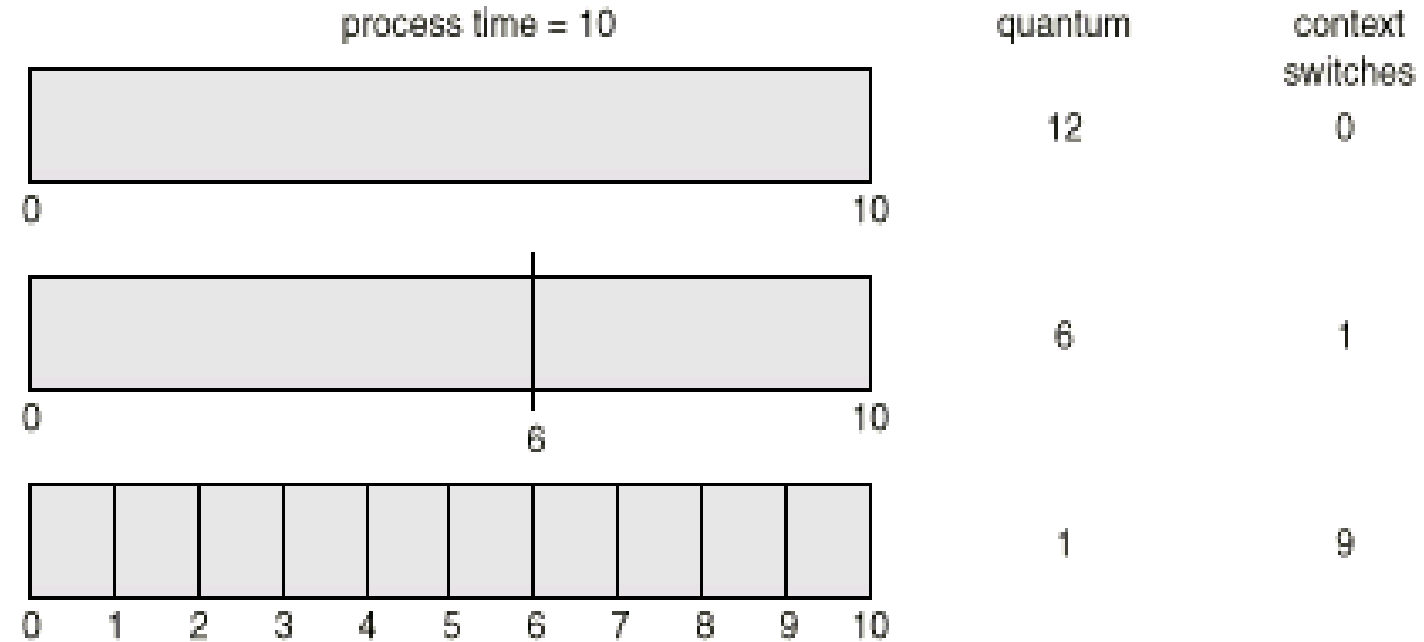
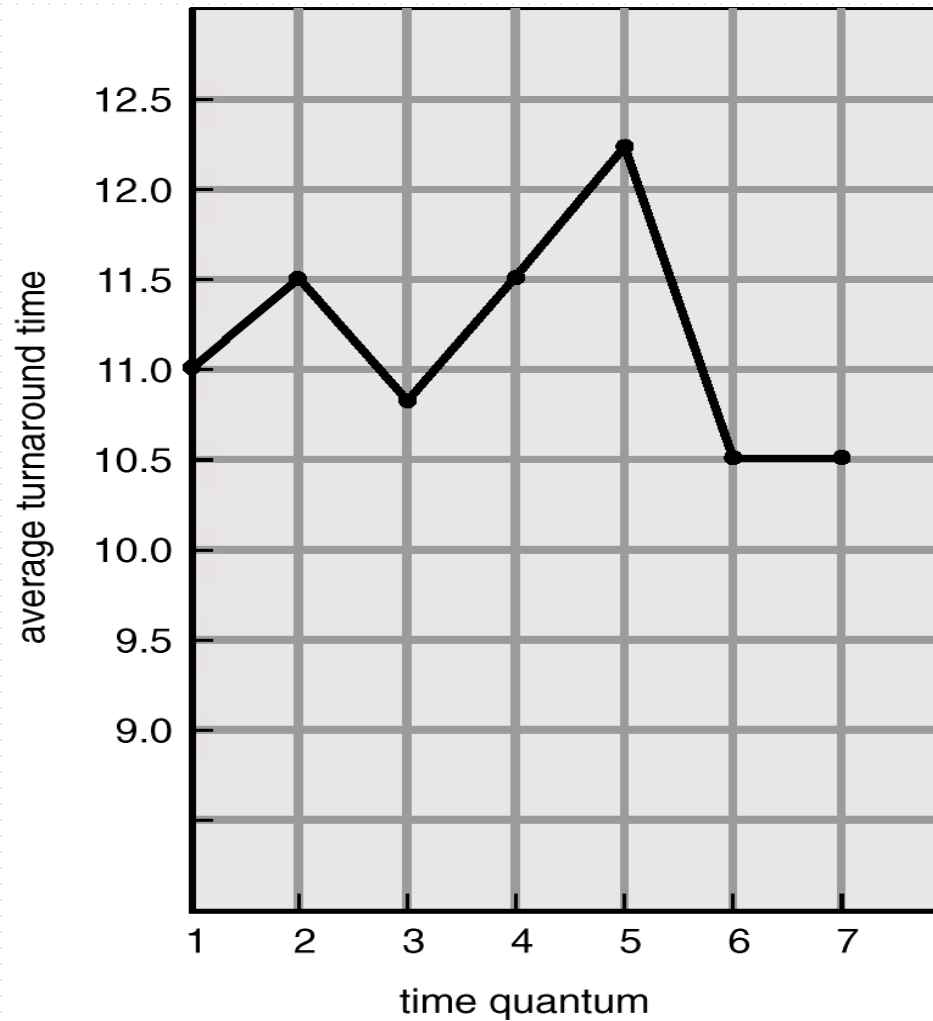


Fig. 5.4 Time Quantum and Context Switch Time

Time Quantum vs Performance: turnaround time



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Fig. 5.5 Turnaround Time Varies With The Time Quantum

Time quantum vs Performance

- If there are n processes in the ready queue and the time quantum is q , then
 - ▣ each process gets $1/n$ of the CPU time in chunks of **at most q time units** at once
 - ▣ no process waits more than $(n-1)q$ time units
- 问题：
 - n 个进程，要求每个进程的**响应时间**（？）至多为 m ，则时间片 q 至多应是多少？
 - ▣ 未必能保证，**必要但非充分条件**
 - ▣ 响应时间不仅仅取决于时间片，还取决于进程内部业务逻辑

Multilevel Queue

■ Principles

two-level scheduling as an example

1. 进程分门别类，不同类型进程位于不同队列，具有不同的调度机会/优先级
2. 队列间调度 + 队列内调度

- ❑ the ready queue is partitioned into separate queues, for example
 - foreground (interactive)
 - background (batch)
- ❑ each queue takes its own scheduling algorithm,
 - foreground – RR
 - background – FCFS
- ❑ scheduling is done among the queues at first, the selected queue is then scheduled among its processes

Multilevel Queue

■ Implementation

- ❑ fixed priority scheduling for scheduling among queues
 - e.g., serve all from foreground then from background)
 - possibility of starvation
- ❑ time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes
 - e.g. , 80% to foreground in RR, 20% to background in FCFS

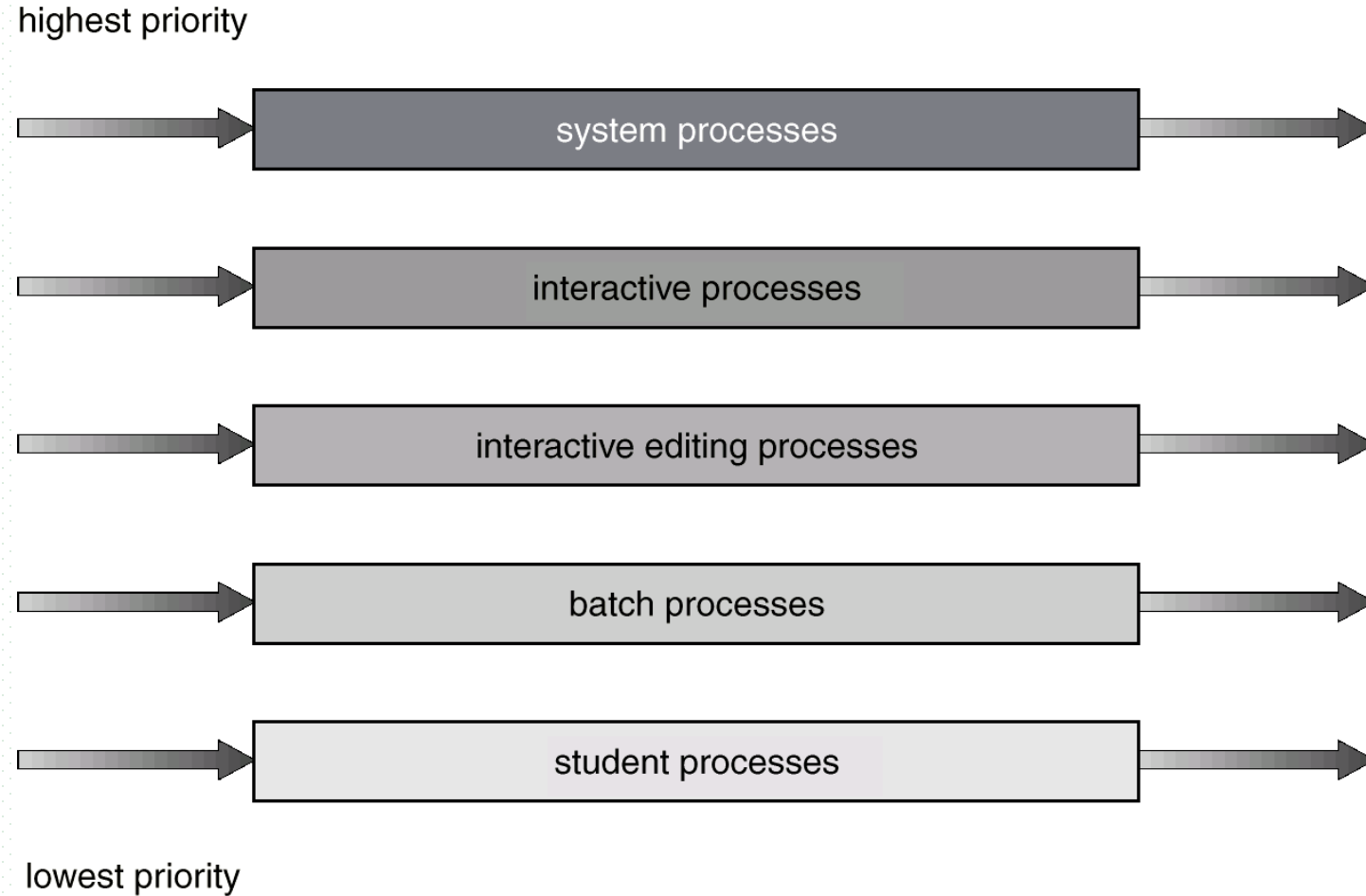


Fig. 5.6 Multilevel Queue Scheduling

Multilevel Feedback Queue

■ Principles

- ❑ a process can move between the various queues, and aging can be implemented in this way
- ❑ multilevel-feedback-queue scheduler defined by the following parameters
 - the number of queues
 - scheduling algorithms for each queue
 - the method used to determine when to **upgrade (升级)** a process
 - the method used to determine when to **demote (降级)** a process
 - the method used to determine which queue a process will enter

Example 1

a process with burst length of (8+16+9)

■ Three queues

- ❑ Q_0 – time quantum 8 milliseconds
- ❑ Q_1 – time quantum 16 milliseconds
- ❑ Q_2 – FCFS

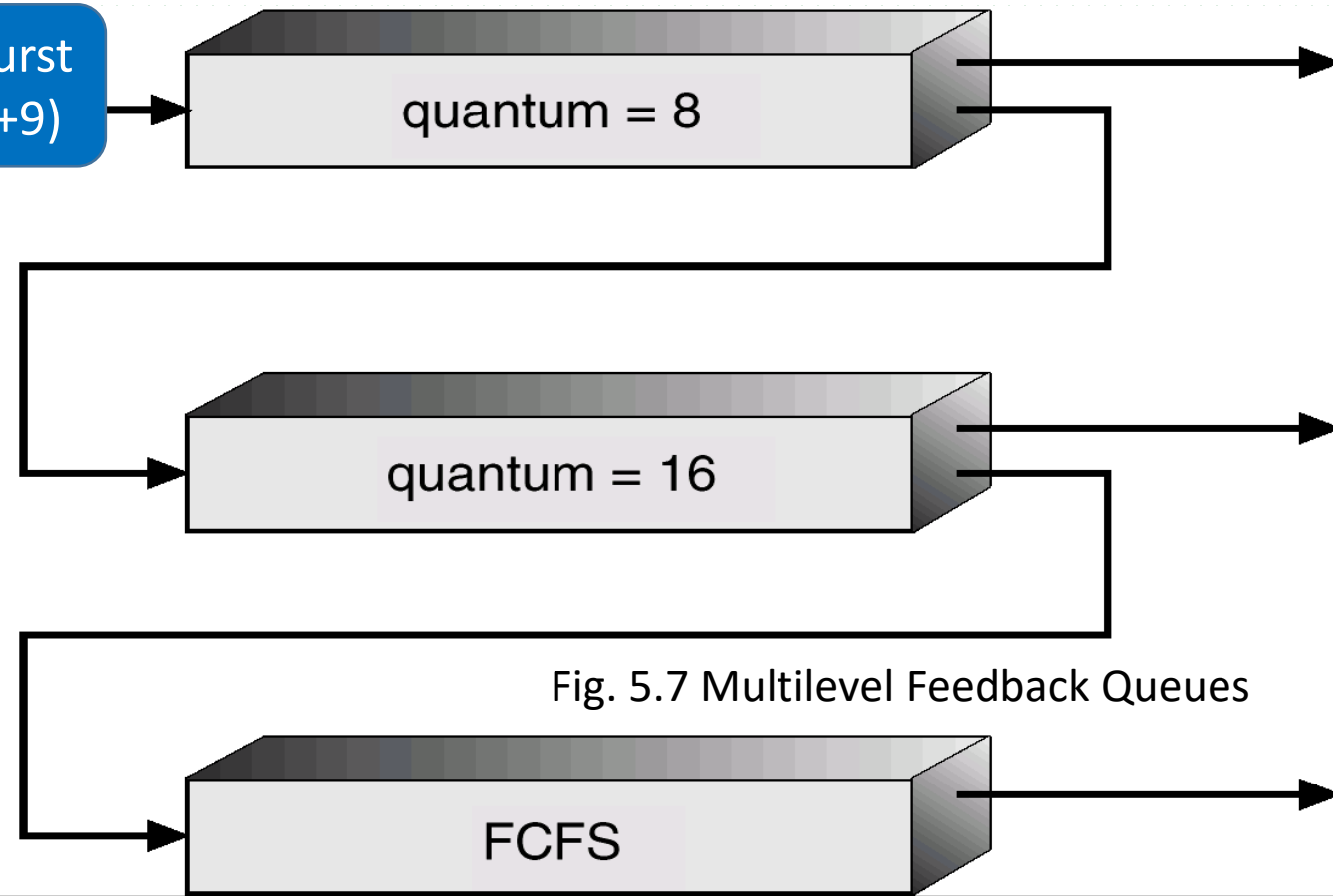


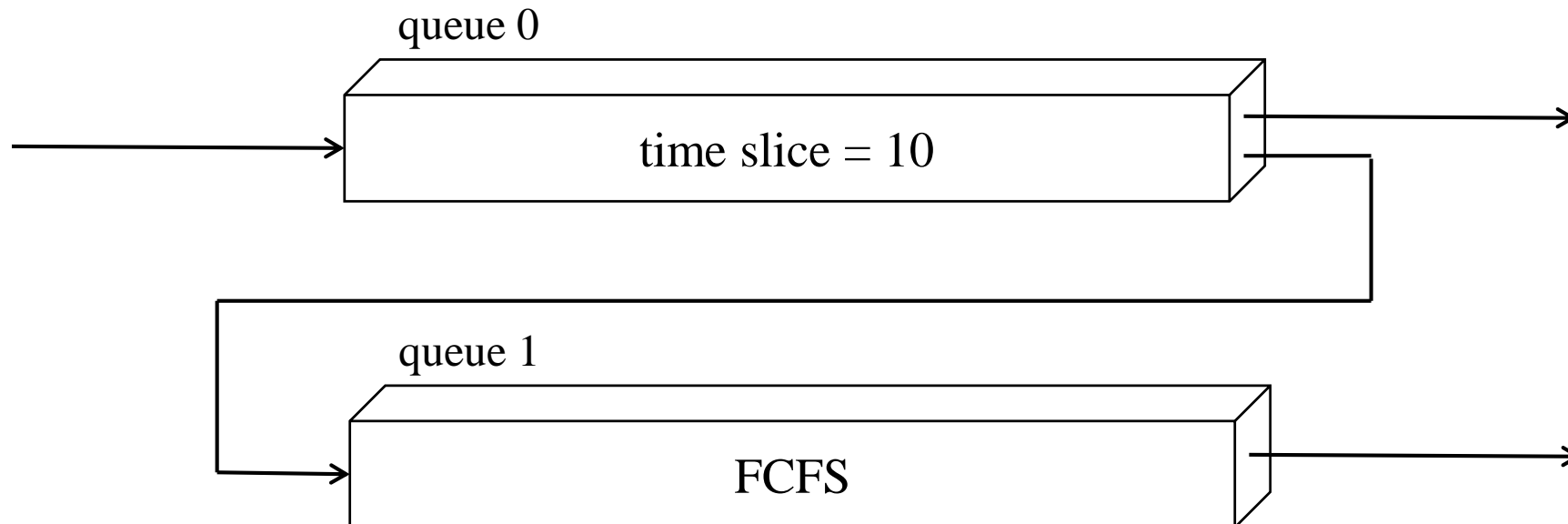
Fig. 5.7 Multilevel Feedback Queues

■ Scheduling

- ❑ a new job enters queue Q_0 which is served **FCFS**. When it gains CPU, the job receives 8 milliseconds. If it does not finish in 8 milliseconds, the job is moved to queue Q_1
- ❑ at Q_1 , job is again served **FCFS** and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2

Example 2

- As shown below, OS takes a two-level feedback-queue scheme to allocate CPU for concurrent processes
- A process entering the system is at first put in queue 0, and sequentially given a CPU time slice of 10 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. Processes in queue 1 run on FCFS scheduling, but are permitted to run only when there is no process in queue 0. When a process P_i in queue 1 is running on CPU and a new process P_j enters the system, P_j will preempt the CPU occupied by P_i



Example 2

- Consider the processes P_0, P_1, P_2, P_3 . For $0 \leq i \leq 3$, the arrival time, the length of the CPU burst time, and the priority of each P_i are given as below

Process	Arrival time	Burst Time	Priority
P_0	0.0	6.0	10
P_1	4.0	15.0	8
P_2	8.0	4.0	6
P_3	12.0	13.0	4

无用、
干扰信息

- For the snapshot shown above, suppose that two-level feedback-queue scheduling is employed
 - ▣ (1) Draw the Gantt chart that illustrates the execution of these processes.
 - ▣ (2) What are the turnaround times for the four processes?

Highest Response Rate First (HRRF)

- HRRF

- ▣ Highest Response Rate First, 响应比最高者优先

- 引入

- ▣ FCFS: 只考虑作业等待时间, 利于减少waiting time
 - ▣ SJF: 只考虑估计的作业运行时间(burst time)

将上述2算法结合, 既照顾短作业, 又不会使长作业长时间等待, 改善调度性能

- 响应时间 = 进程进入系统后的等待时间 + 估计计算时间(CPU burst)

- 响应比 = 响应时间/估计计算时间
= 1 + 等待时间/估计计算时间



HRRF算法中定义的响应时间, 与5.2节调度准则中定义的响应时间response time (专门针对第一章提到的交互式系统) 不一样, 各自有其适用范围

Highest Response Rate First (HRRF)

■ HRRF调度算法

- 从就绪队列中选择响应比最高的进程，分配CPU
- 非抢占式

■ 特点

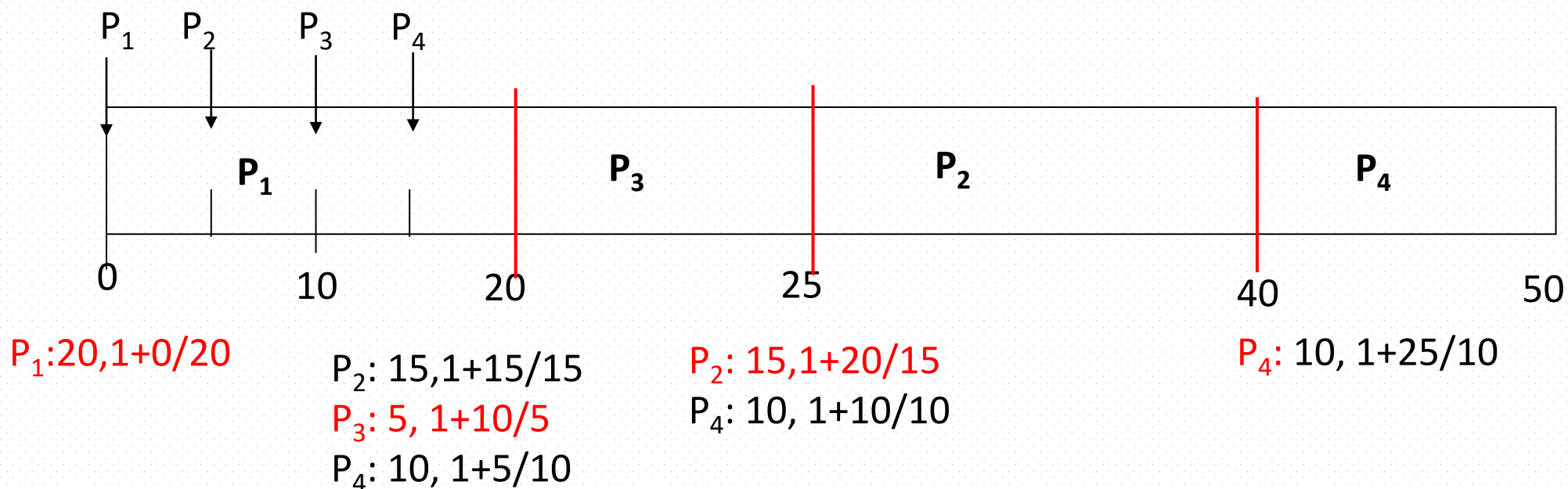
- 短作业的估计计算时间短，响应比较高，因此本算法优待(prefer to)短作业
- 对于长作业，如果在系统中等待时间足够长，将获得很高的响应比，进而被优先调度，不会发生“饿死”现象

- $SJF < HRRF \text{ 的平均周转时间} < FCFS$

Example

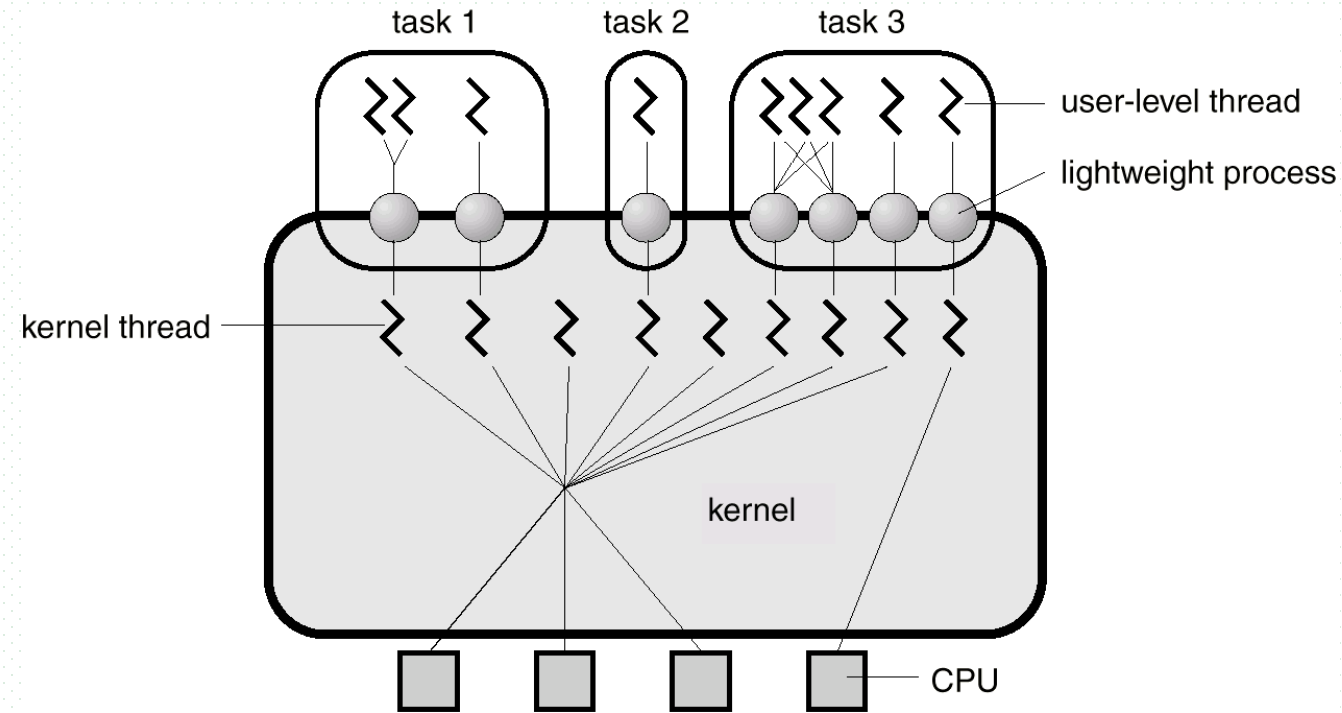
Process	Arrival Time	Burst Time
P ₁	0.0	20
P ₂	5.0	15
P ₃	10.0	5
P ₄	15.0	10

- 开始时, P₁, P₁被选中, 执行时间20
- P₁执行完后, P₂、P₃、P₄的响应比分别为
 $1 + 15/15$ 、 $1 + 10/5$ 、 $1 + 5/10$
 , P₃被选中, 执行时间5
- P₃执行完后, P₂、P₄的响应比分别为
 $1 + 20/15$ 、 $1 + 10/10$
 , P₂被选中, 执行时间15
- P₂执行完后, P₄执行, 执行时间10



5.4 Thread Scheduling

- Kernel-level threads are managed by OS, while user-level threads by thread library
- How to conduct scheduling involving KLT and ULT



- Two-level scheduling for ULT
 - local scheduling
 - how the threads library decides which threads in processes to put onto an available LWP
 - global scheduling
 - how the kernel decides which kernel thread to run next

Thread Scheduling: Contention Scope(竞争范围)

PCS, SCS:
concepts in Pthread

■ Process-contention scope (进程竞争范围, PCS) 【local scheduling】

- for many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
 - since scheduling competition is within the process
/* 决定将应用进程内的哪个线程安排在LWP, 参与后续调度
- the number of LWPs is controlled by thread library
- typically done via priority set by programmer, priority-based preemption is allowed
 - 采用优先级调度, 由编程者设定线程优先级; 允许高优先级线程抢占低优先级线程

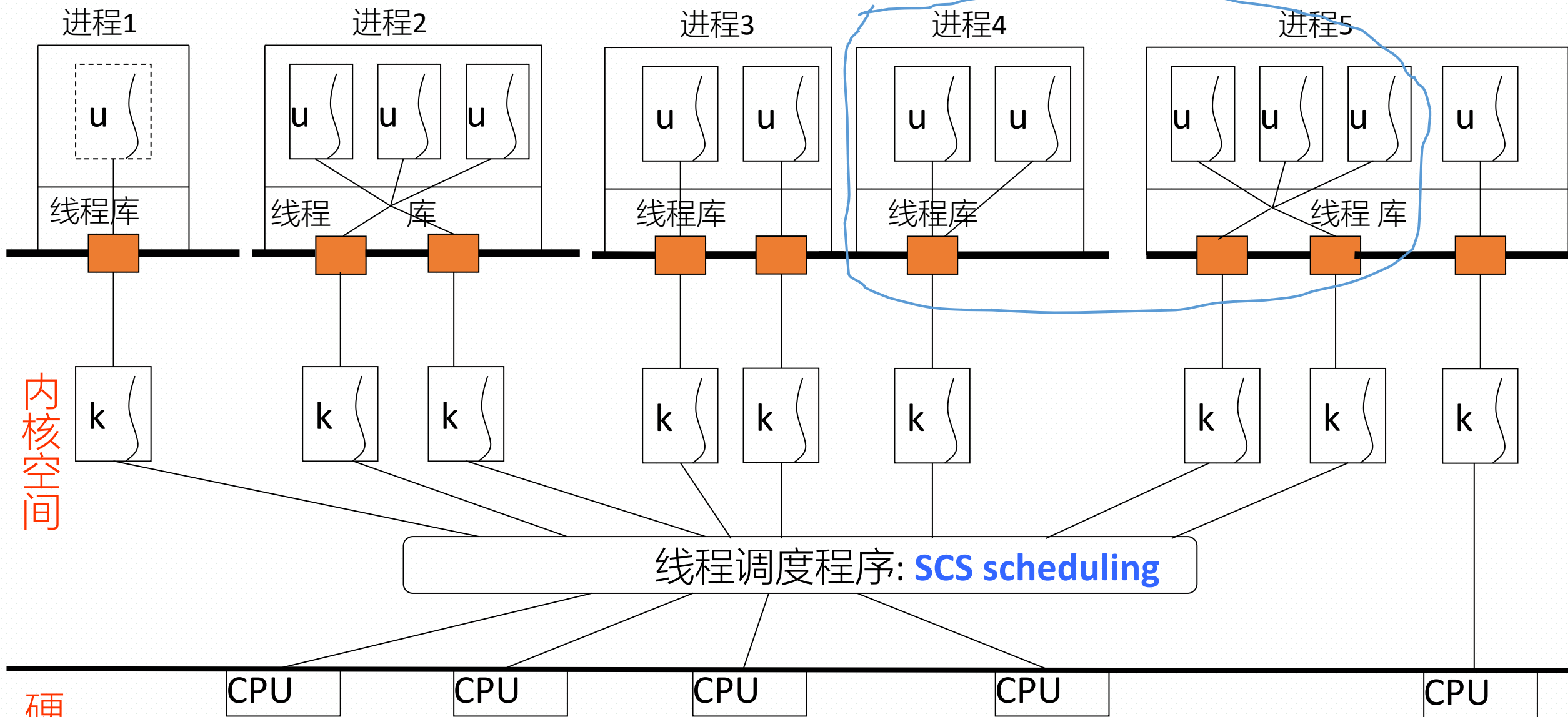
■ System-contention scope (系统竞争范围, SCS) 【global scheduling】

- OS schedules kernel threads onto available CPU
- competition among all kernel threads in system
- for one-to-one mapping in Windows, Linux and Solaris, SCS scheduling is used

用户空间

LWP: 

PCS scheduling



例. Window Server + Java Run-time System

Pthread Scheduling

- Pthread_create() API allows specifying either PCS or SCS during thread creation
 - ▣ PTHREAD_SCOPE_PROCESS: schedules threads using PCS scheduling
 - ▣ PTHREAD_SCOPE_SYSTEM: schedules threads using SCS scheduling
- On system implementing **many-to-many** model, PCS policy schedules user-level threads on to available LWPs, the number of LWPs is maintained by the thread library
 - ▣ PCS policy creates and binds a LWP for each user-level thread, mapping thread using **one-to-one** policy
- Getting and setting contention scope policy in Pthread IPC

```
pthread_attr_setscope(pthread_attr_t *attr, int scope)
pthread_attr_getscope(pthread_attr_t *attr, int scope)
```

Thread的属性集 竞争范围

 - ▣ 执行错误，函数返回非零值

1. 线程创建 `pthread_create()`

调用格式:

```
#include<pthread.h>
```

```
int pthread_create(pthread_t *thread, pthread_attr_t*attr, void*(*start_routine)(void *), void  
*arg)
```

参数:

thread: 指向所创建线程的标识符的指针, 线程创建成功时, 返回被创建线程的 ID

attr: 用于指定被创建线程的属性, NULL 表示使用默认属性

start_routine: 函数指针, 指向线程创建后要调用的函数, 是一个以指向 void 的指针作为参数和返回值的函数指针, 这个被线程调用的函数也被称为线程函数

arg: 指向传递给线程函数的参数

返回值:

创建成功: 0

创建失败: 返回错误码

```

1  /* thread_create.c */
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include<pthread.h>
5
6  /*线程函数1*/
7  void *mythread1(void)
8  {
9      int i;
10     for(i=0;i<5;i++)
11     {
12         printf("I am the 1st pthread,created by mybeilef321\n");
13         sleep(2);
14     }
15 }
16 /*线程函数2*/
17 void *mythread2(void)
18 {
19     int i;
20     for(i=0;i<5;i++)
21     {
22         printf("I am the 2st pthread,created by mybelief321\n");
23         sleep(2);
24     }
25 }
26
27 int main()
28 {
29     pthread_t id1,id2;  /*线程ID*/
30     int res;
31     /*创建一个线程，并使得该线程执行mythread1函数*/
32     res=pthread_create(&id1,NULL,(void *)mythread1,NULL);
33     if(res)
34     {
35         printf("Create pthread error!\n");
36         return 1;
37     }
38     /*创建一个线程，并使得该线程执行mythread2函数*/
39     res=pthread_create(&id2,NULL,(void *)mythread2,NULL);
40     if(res)
41     {
42         printf("Create pthread error!\n");
43         return 1;
44     }
45     /*等待两个线程均推出后，main()函数再退出*/
46     pthread_join(id1,NULL);
47     pthread_join(id2,NULL);
48
49     return 1;
50 }

```

```

song@ubuntu:~/lianxi$ vim thread_create.c
song@ubuntu:~/lianxi$ gcc thread_create.c -o thread_create -lpthread
song@ubuntu:~/lianxi$ ./thread_create
I am the 2st pthread,created by mybelief321
I am the 1st pthread,created by mybeilef321
I am the 2st pthread,created by mybelief321
I am the 1st pthread,created by mybeilef321
I am the 2st pthread,created by mybelief321
I am the 1st pthread,created by mybeilef321
I am the 2st pthread,created by mybelief321
I am the 1st pthread,created by mybeilef321
I am the 2st pthread,created by mybelief321
I am the 1st pthread,created by mybeilef321

```

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;          /*线程属性集合*/
    pthread_attr_init(&attr);    /* get the default attributes */
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0) /*非零值，表示error
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");    /*PCS */
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");     /*SCS */
            else
                fprintf(stderr, "Illegal scope value.\n");
    }
}
```

```
/* set the scheduling algorithm to PCS or SCS, 设置为SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

/* create the threads 创建多个子线程 */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);

/* now join/wait/等待 on each thread, 创建的线程等待 */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param) /*线程执行体 */
{
    /* do some work ... */
    pthread_exit(0);
}
```

5.5 Multiple-Processor Scheduling

- Two-level scheduling in multi-processor system
 - ▣ step1. assign processes to CPUs/processors
 - which processor can be allocated to the selected processes?
 - ▣ step2. schedule processes on a CPU/processor
- Concerns in multiprocessor scheduling
 - ▣ Multiple-processor scheduling approaches
 - homogeneous processors, symmetric/asymmetric multiprocessing
 - ▣ Processor affinity
 - ▣ Load balancing
 - ▣ Multicore processors and Multithreading

Multiple-processor Scheduling Approaches

- CPU scheduling is more complex when multiple CPUs are available
 - ▣ Homogeneous processors (同构处理器) within a multiprocessor system
 - 考虑processor是否适合承担任务
 - e.g. for a gaming task, it should be assigned to a CPU with integrated graphics, such as intel i9xxx processor, not E5 processor without graphics
 - ▣ Asymmetric (非对称) multiprocessing
 - one processor makes scheduling decisions, does I/O operations and accesses the system data structures(管理节点, master), the other processors execute user programs
 - ▣ Symmetric multiprocessing (SMP)
 - each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
 - 每个process的角色、功能一样, 无差别
 - Currently, most common

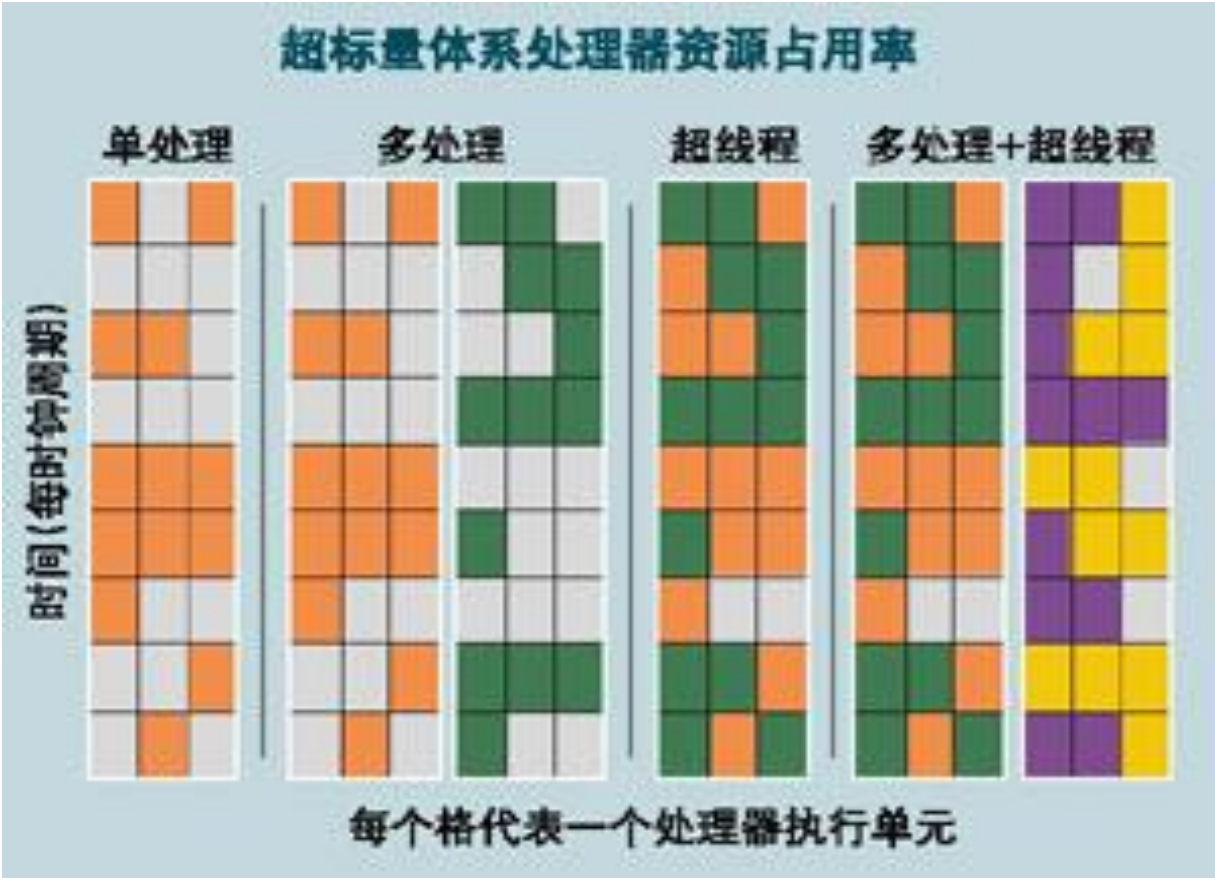
Processor Affinity

- Processor affinity (亲和)
 - because of high scheduling cost, avoid migration of processes from one processor to another, and instead attempt to keep a processor running on the same processor
 - process has affinity for processor on which it is currently running
 - 将进程运行在特定processor上
- Hard affinity
 - alleviate system calls to specify a subset of processors on which the process may run
 - 指定进程运行的处理器
- Soft affinity
 - OS attempt to keep a process on a single processor, but it is possible for the process to migrate between processors
- Linux scheduler implements **soft affinity**, it also provides system call `sched_setaffinity()` to support hard affinity

Load Balancing

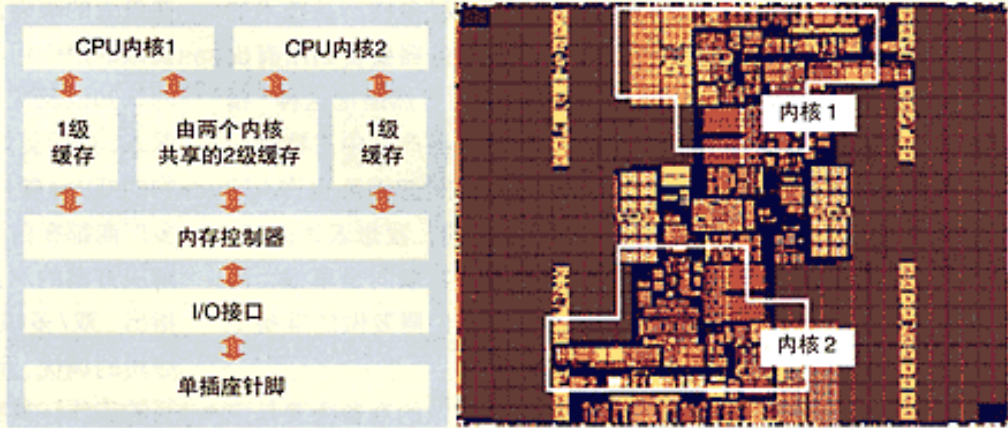
- If SMP, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed
- Push migration
 - ▣ periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- Pull migration
 - ▣ idle processors pulls waiting task from busy processor

Multicore Processors and Multithreading



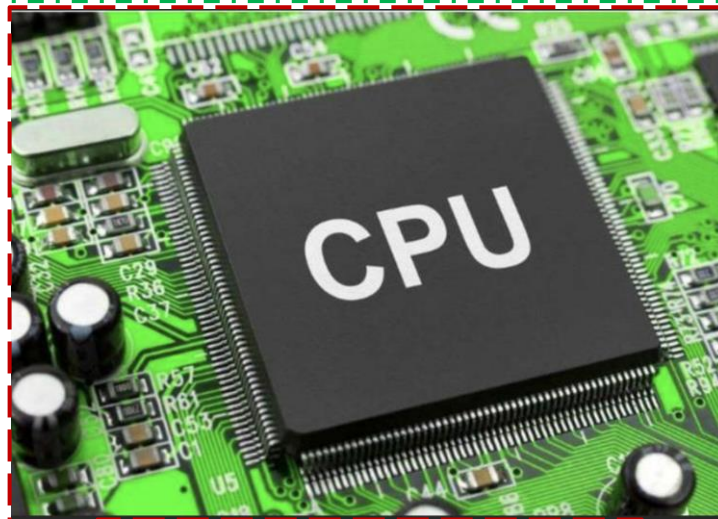
多内核架构

目前的通用多内核处理器设计都是基于对称多处理（SMP）技术的。SMP 是一种并行架构，在这种架构中，所有的处理器运行一个操作系统副本，共享内存和一台计算机的其他资源，并且平等地访问内存、I/O 及外部中断。大多数 SMP 架构是由下图演化而来的：

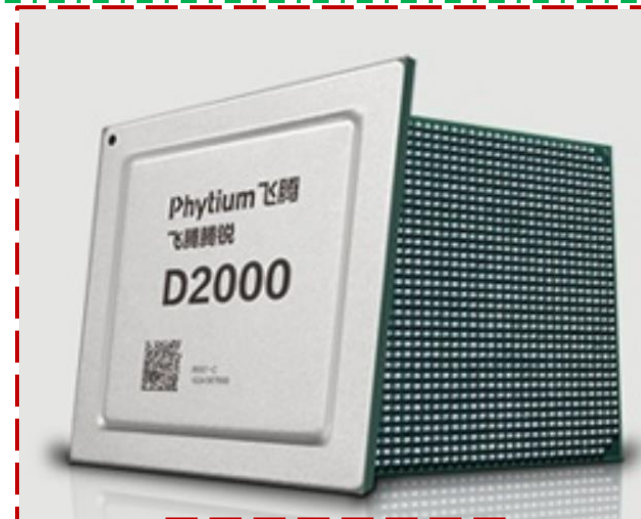




龙芯：
MIPS架构，loongarch指令集



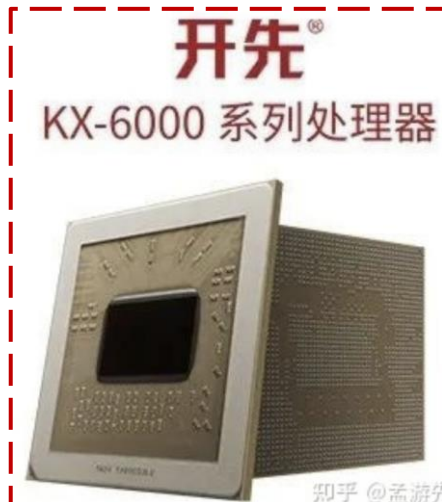
申威：
Alpha架构，sw64指令集



飞腾：
Arm架构/指令集



中科曙光~海光：
x86架构，AMD Zen1微架构



魔都~兆芯：
x86架构/ip内核/指令集



华为~鲲鹏：
ARM架构/指令集

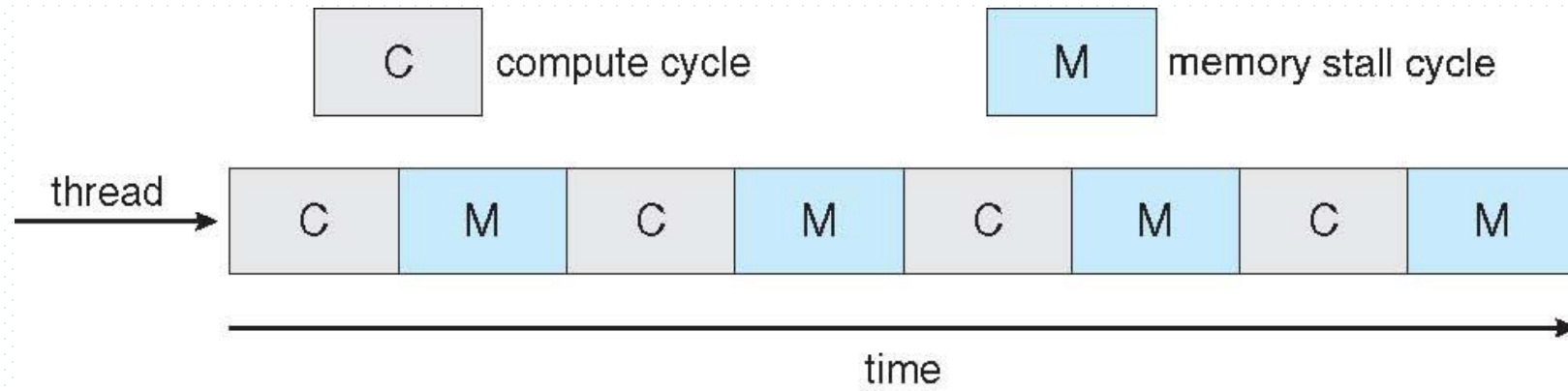


阿里RV~玄铁：
基于开源RISC-V

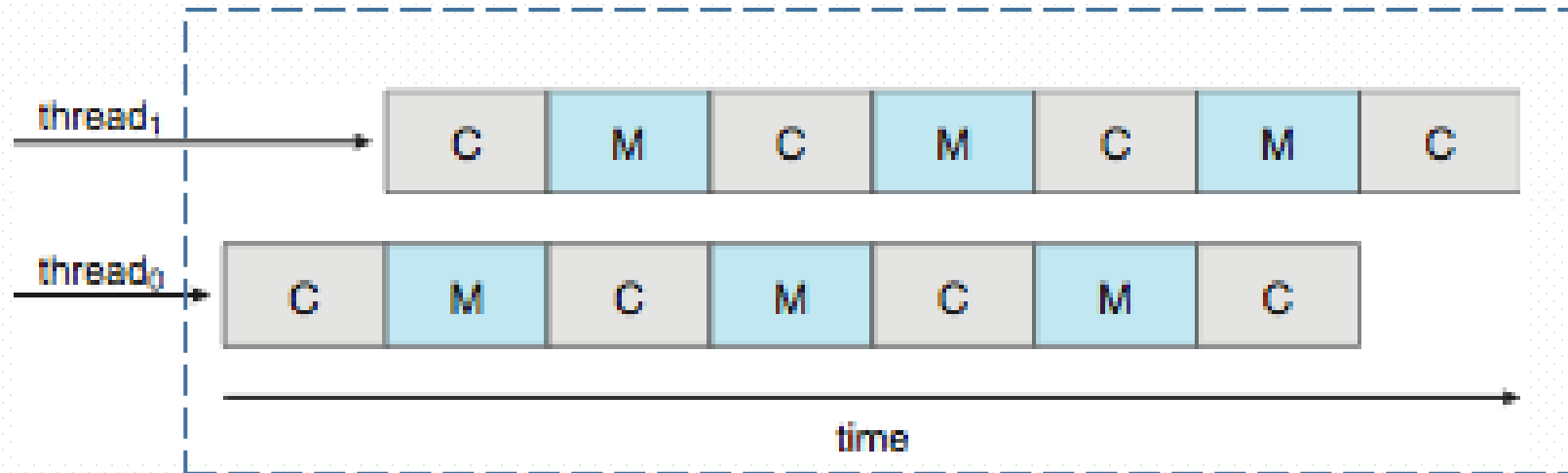
Multithreaded Multicore Processor

- Multiple threads per core—hardware thread/hyperthread（超线程）
 - core with hardware thread, or with hyperthread HT
 - takes advantage of memory stall (内存停顿) to make progress on another thread while memory retrieve happens
 - 一个物理核上运行多个硬件线程（超线程HT?），每个硬件线程类似于一个独立的逻辑处理器
 - 两个硬件线程/逻辑处理器，利用内存停顿，交错执行
 - UltraSPARC T3 CPU
 - 16 cores, 8 hard threads per core, thus 128 logical CPU
 - 从操作系统角度，每个硬件线程/逻辑处理器都是一个可以调度分配的CPU

Multithreaded Multicore Systems



内存停顿周期：
单核运行单线程，访存周期等待内存数据，停顿



一个物理核上的两个硬件线程(or HT)，交错执行



Scheduling in multithreaded multicore

- Two levels of scheduling
 - ▣ OS level
 - OS choose which software thread (应用线程、程序、进程) to run on each hardware thread (logical processor) on CPU
 - ▣ CPU level
 - CPU core decides which hardware thread to run
 - /*哪个逻辑CPU真正占用CPU core的物理资源去执行

Scheduling in multithreaded multicore

- Example1. Scheduling policy at CPU level in UltraSparc T3
 - round- robin algorithm 【时间片轮转】
 - schedule the eight hardware threads to each core
- Example2. Scheduling policy at CPU level in Intel Itanium dual core CPU
 - each hardware thread is assigned a dynamic urgency value ranging from 0 to 7, with 0 representing the lowest urgency and 7 the highest
 - 优先级调度, 优先级=urgency value
 - the Itanium identifies five different events that may trigger a thread switch
 - when one of these events occurs, the thread-switching logic compares the urgency of the two threads and selects the thread with the highest urgency value to execute on the processor core

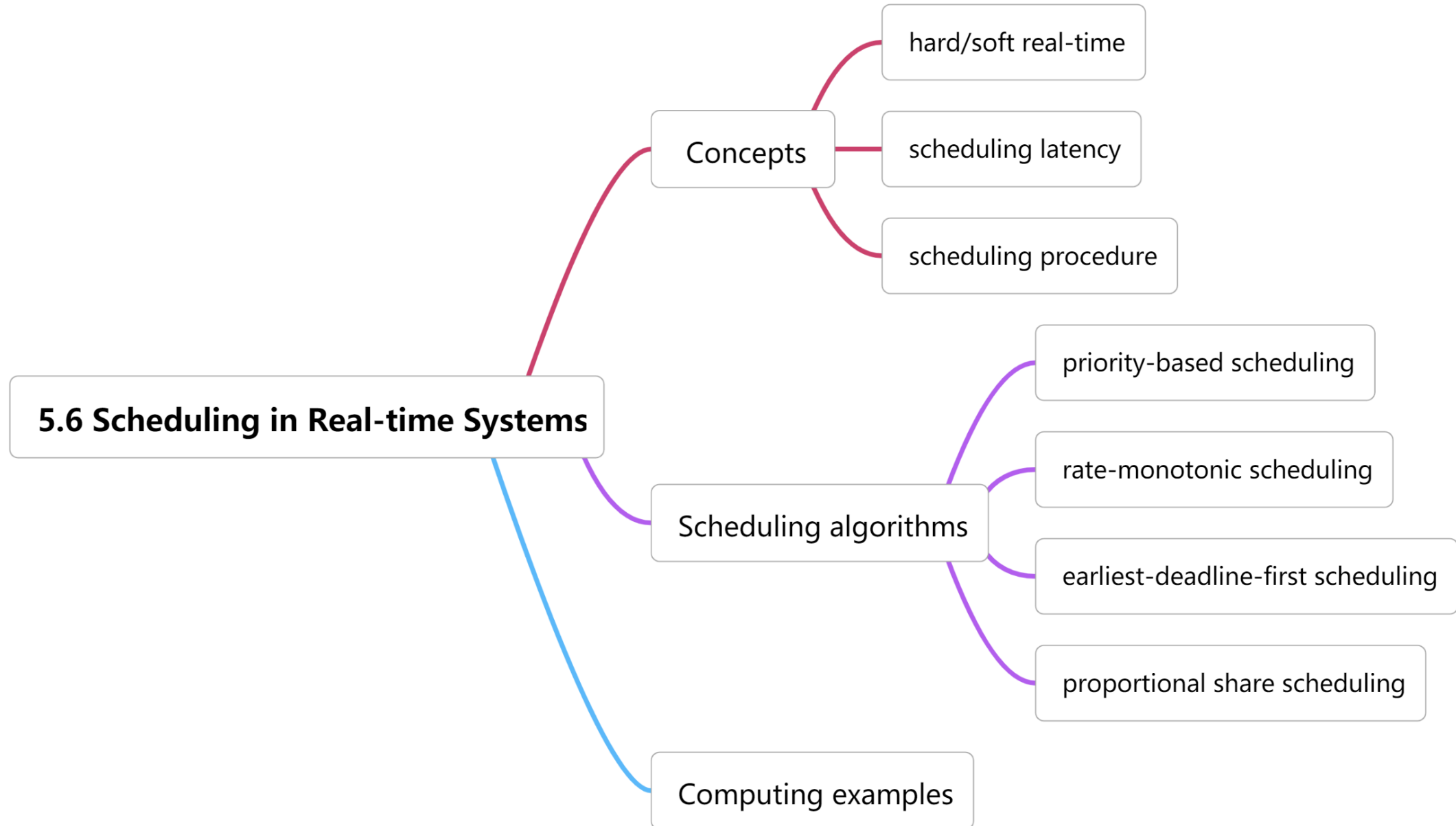
补充资料：Types of Hardware Multithreading in CPU Core

- Objects of multithreading
 - ▣ 同一CPU流水线上执行多个线程，提高（取指、译码、计算、访存等）流水线器件利用率
- 三种实现方式
 - ▣ Interleaved multithreading
 - ▣ Block multithreading
 - ▣ Simultaneous multithreading
- Interleaved multithreading
 - ▣ 细粒度（前述Fine-grained multithreading）
 - ▣ 线程context切换发生在每一个boundary of an instruction cycle，每个周期切换一次，但保留流水线信息

补充资料：Types of Hardware Multithreading in CPU Core

- Block multithreading
 - ▣ 粗粒度（前述coarse-grained multithreading）
 - ▣ 当线程执行发生了latency较高的事件，如memory stall，发起线程切换，并清除流水线中的信息，执行其它线程来隐藏降低latency
- Simultaneous multithreading
 - ▣ Intel CPU中的超线程HyperThreading
 - ▣ 通过增加CPU计算资源，如取指单元、译码单元、Issue队列、CDB、BTB、分支预测器，以及寄存器等部件数量，提高运算单元利用率

5.6 Scheduling in Real-time Systems



Scheduling in Real-time Systems

■ Real-time system

- ❑ a computer and/or software system that reacts to **external events** in limited time intervals (**deadline**) before the events become obsolete (失效)
- ❑ specific-purpose systems, such as industrial control systems
- ❑ each external event corresponds to critical task or process in the system, that is, for each external event, there is a critical task or process to react to or handle it

■ Soft real-time systems

- ❑ no guarantee as to when critical real-time process will be scheduled

■ Hard real-time systems

- ❑ task must be serviced by its deadline

Example

- The task or process **deadline** in real time systems
 - ▣ the time limit before which the task/process should be selected by scheduler, run on CPU and complete processing, in order to react to and handle the external event in time

<u>Events</u>	<u>Critical process</u>	<u>Arrival Time</u>	<u>burst time</u>	<u>deadline</u>	<u>priority number</u>
1	P ₁	0.0	4.0	7.00	3
2	P ₂	3.0	2.0	5.50	1
3	P ₃	4.0	2.0	12.01	4
4	P ₄	6.0	4.0	11.00	2

Note: a small priority number means higher priority

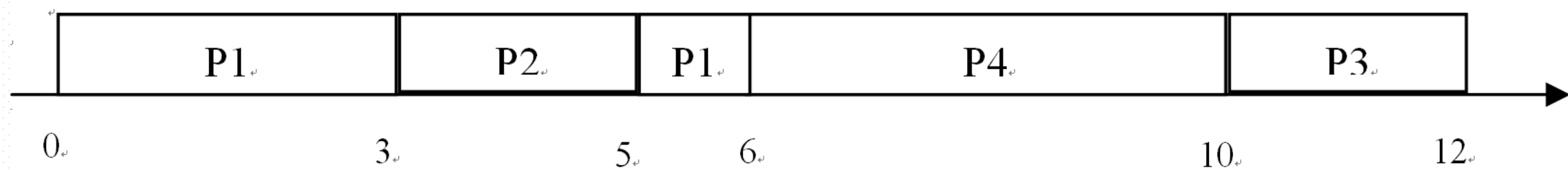


Example

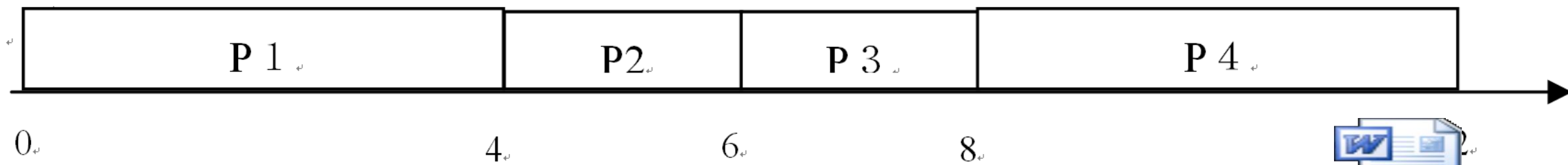
- Suppose that FCFS scheduling is employed,
 - give the Gantt charts illustrating the execution of these processes
 - give the turnaround time and waiting time of each process
 - give the average turnaround time and the waiting time
 - which event will be treated with in time, and which not ?
 - be treated with in time:
the event will not finish after its deadline
- Suppose that priority-based preemptive scheduling scheduling is employed,
 - give the Gantt charts illustrating the execution of these processes
 - give the turnaround time and waiting time of each process
 - give the average turnaround time and the waiting time
 - which event will be treated with in time, and which not ?

Example

- Priority-based preemptive
 - ▣ all processes are treated with in time



- FCFS
 - ▣ e1, e3 are treated with in time



deadline到达时，如果进程没有结束，应继续执行，不会被强行终止



!!
章典型例题-第5章-

Minimizing Latency

- When an event occurs, the system must respond to and service it as quickly as possible.
- Real-time system is **event-driven** and **time-critical** system
 - external events → interrupts → scheduling among critical processes → the critical process responsible for reacting to and processing these events in limited time (before the deadline)
- Event latency
 - the amount of time that elapses from when an event occurs to when it is serviced
 - similar to **response time**

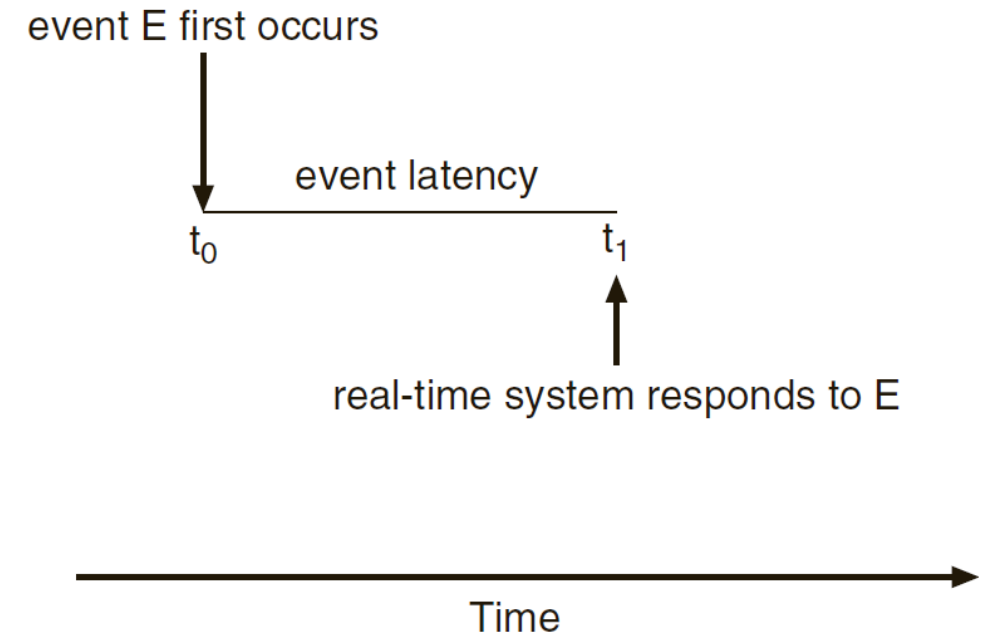


Figure 5.12 Event latency.

Minimizing Latency

- Two types of latencies affect performance
 - ▣ Interrupt latency
 - time from arrival of interrupt to start of routine that services interrupt
 - ▣ Dispatch latency
 - time for schedule to take current process off CPU and switch to another

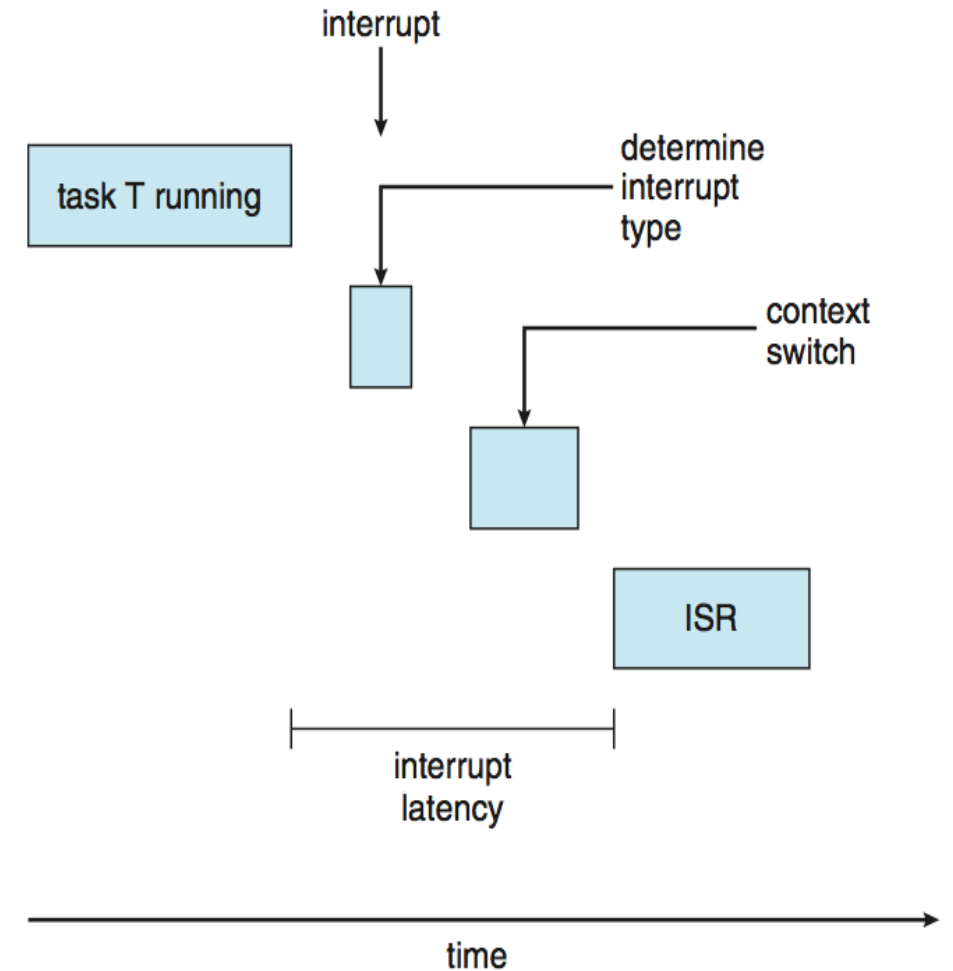


Figure 5.13 Interrupt latency

Minimizing Latency

■ Dispatch latency consists of two phases

▣ conflict phase

- 1. preemption of any process running in kernel mode
- 2. releasing the resources occupied by low-priority processes(e.g. P1) and needed by the high-priority process (e.g. P2) in the above-mentioned example

▣ dispatch phase

- allocate resources for the selected high-priority process (e.g. P2) to react to the external event and start the process' running

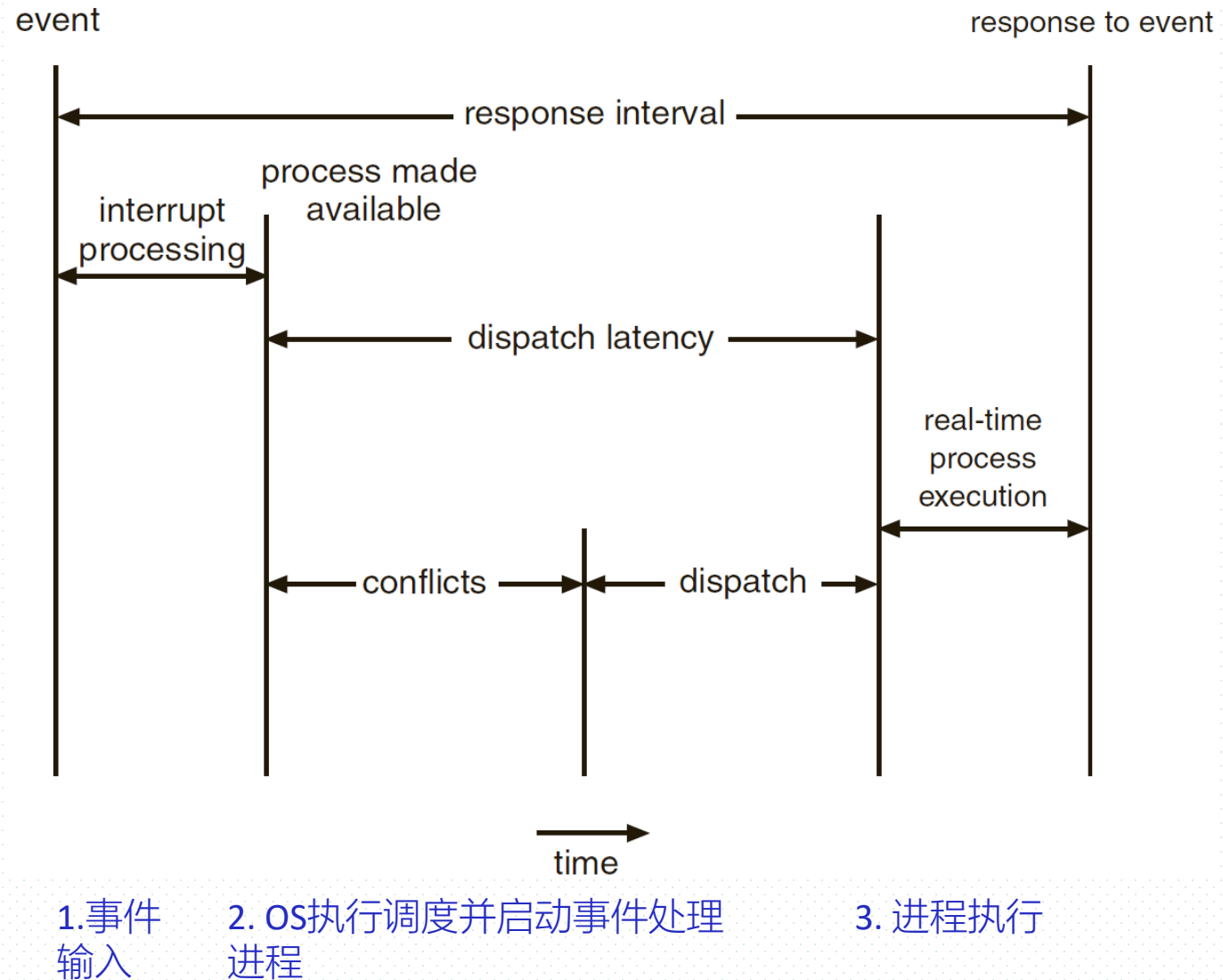


Figure 5.14 Dispatch latency

Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
 - but only guarantees soft real-time
- Scheduling in soft real-time system
 - all external requests are admitted to enter the system
 - the real-time processes have highest priorities, and do not degrade over time
 - the **dispatch latency** must be smaller, in order for a real-time process to start executing as faster as possible, once it is runnable. For this purpose, system calls should be allowed to be preemptive

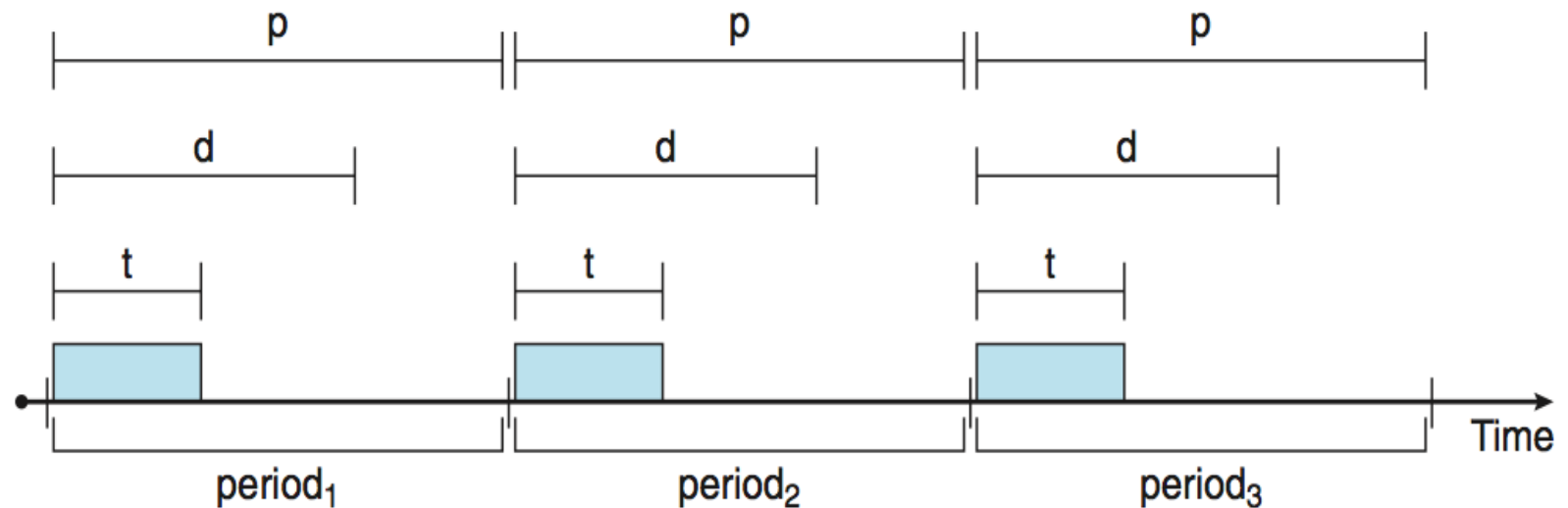


Priority-based Scheduling

- For hard real-time, scheduler must also provide ability to meet deadlines
- Scheduling in **hard** real-time system
 - **priority-based and preemptive scheduling**
 - the scheduler should know exactly about the critical tasks, such as arrival time, burst time, deadline and priority
 - on the basis of schedulability analysis (可调度性分析) , the enough resources are configured in systems, making that when the scheduler admits the critical processes to response to external events, these processes are guaranteed to complete before their deadlines
 - this is known as **resource reservation**
- Another Example: **确定性网络**
 - 带宽、延迟必须保证
 - 用于工业互联网等场景
 - 方法：在网络协议层2提供保障机制

Priority-based Scheduling

- Processes have new characteristics
 - **periodic** ones require CPU at constant intervals
 - * e.g. 工业控制系统中的周期性温度巡检、压力巡检
 - Has processing time t , deadline d , period p
 - $0 \leq t \leq d \leq p$,
 - e.g. $p=60s$, $t=100ms$
 $d=150ms$
 - **Rate** of periodic task is $1/p$



Scheduling policies for tasks in real-time system

- Rate Monotonic Scheduling
 - ▣ for periodic tasks
- Earliest Deadline First Scheduling (EDF)
- Proportional Share Scheduling

Rate Monotonic (单调) Scheduling

- A priority is assigned to a process based on the **inverse** of its period
 - shorter periods = higher priority
 - longer periods = lower priority
 - 周期 p 大, 发生频率小, 任务不紧迫, 调度的优先级低
- Example: P_1 is assigned a higher priority than P_2 ,
 - P_1 's **period** $p_1=50\text{ms}$, P_1 's processing time/CPU burst $t_1=20\text{ms}$
 - P_2 's **period** $p_2=100\text{ms}$, P_2 's processing time/CPU burst $t_2=35\text{ms}$
 - deadline for P_1 and P_2 : complete its CPU burst by the start of its next period

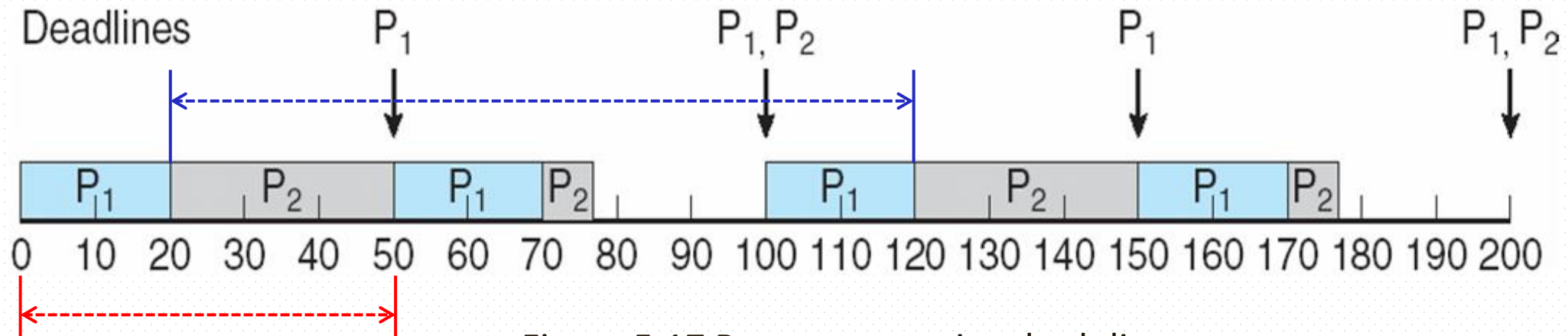


Figure 5.17 Rate-monotonic scheduling

Rate Monotonic Scheduling

- Despite being optimal, then, rate-monotonic scheduling has a limitation
 - CPU utilization is bounded, and it is not always possible fully to maximize CPU resources
 - cannot guarantee that processes can be scheduled so that they meet their deadlines
- The **worst-case** (最小, 下界, Ω) CPU utilization for scheduling N processes is

$$N(2^{1/N} - 1).$$

- $N=1$, utilization=100%
- $N=2$, bounded about 83%
- $N \rightarrow \infty$, utilization $\rightarrow 69\%$

Rate Monotonic Scheduling

- Rate monotonic scheduling cannot guarantee that processes can be scheduled so that they meet their deadlines

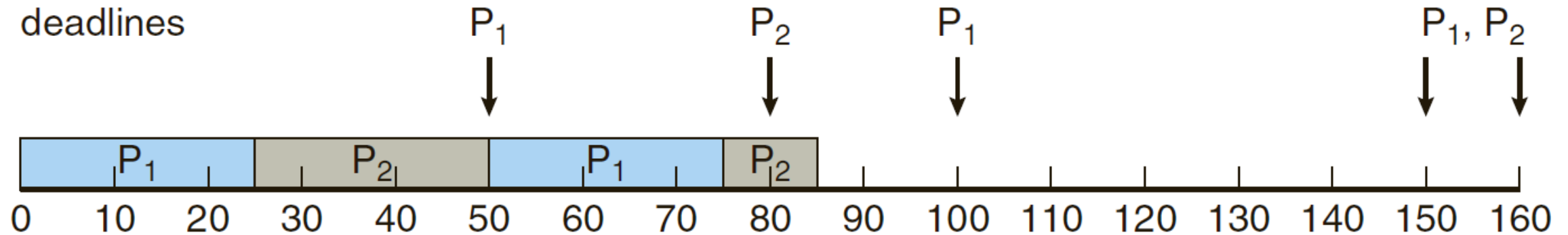


Figure 5.18 Missing deadlines with rate-monotonic scheduling

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned to processes according to their deadlines
 - the earlier the deadline, the higher the priority
 - the later the deadline, the lower the priority /*截止期ddl越小, 任务越紧迫, 优先调度
- Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process
 - dynamic priority
- Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst
 - can be applied to non-period tasks
- The only requirement
 - a process announce its deadline to the scheduler when it becomes runnable

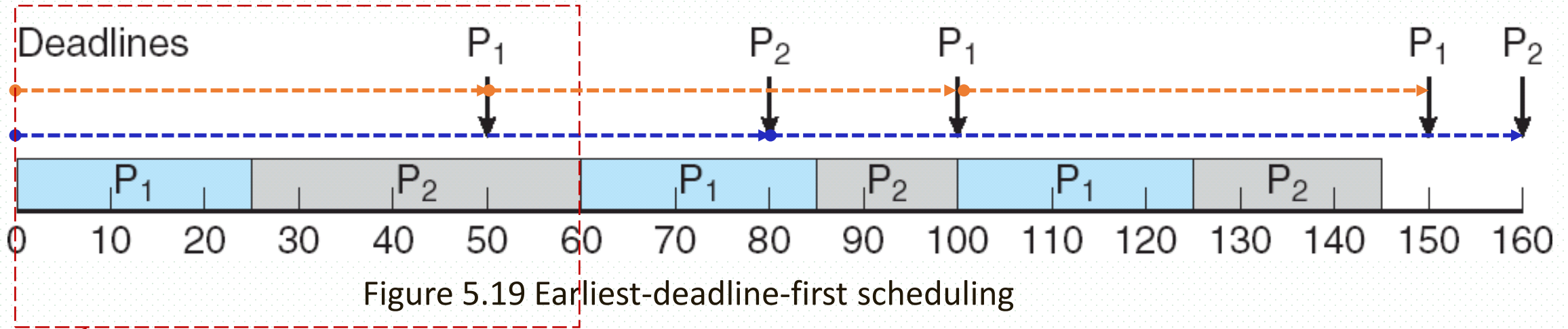
Earliest Deadline First Scheduling (EDF)

- EDF scheduling is **theoretically optimal**
 - ▣ theoretically, it can schedule processes so that
 - each process can meet its deadline requirements
 - CPU utilization will be 100 percent

Earliest Deadline First Scheduling (EDF)

■ Example

- P_1 's **period** $p_1=50\text{ms}$, P_1 's processing time/CPU burst $t_1=25\text{ms}$
- P_2 's **period** $p_2=80\text{ms}$, P_2 's processing time/CPU burst $t_2=35\text{ms}$
- deadline for P_1 and P_2 : complete its CPU burst by the start of its next period



- Step 1: Initially, P_1 is assigned an **initial higher priority** than P_2 , due to its shorter period/deadline 50ms, and is selected to run; and P_2 begins running at time 25, i.e. at the end of the CPU burst for P_1 , and then ends its first burst at time 60

Earliest Deadline First Scheduling (EDF)

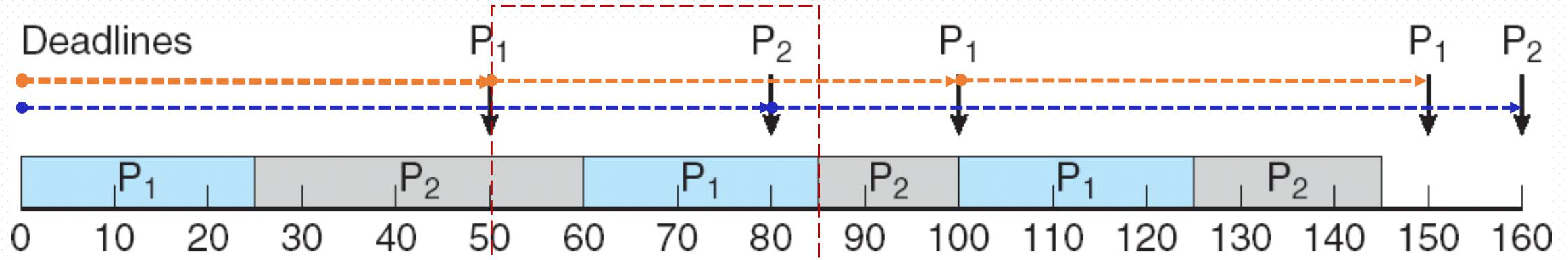
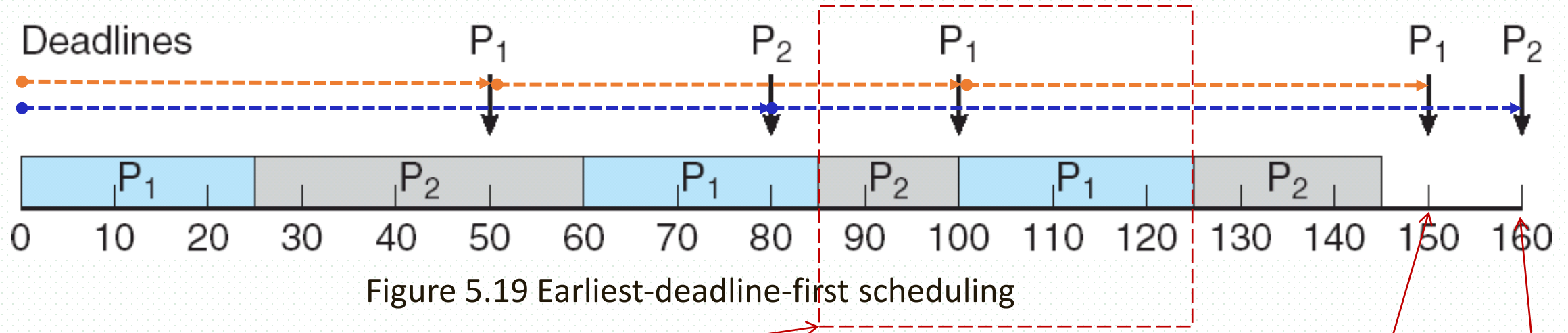


Figure 5.19 Earliest-deadline-first scheduling

- Step 2: At time 50, the P_1 's second burst comes, but EDF scheduling allows P_2 to continue running
 - P_2 now has a higher priority than P_1 , because its next deadline (at time 80, P_2 's second burst comes) is earlier than that of P_1 (at time 100, P_1 's third burst comes)
 - thus, both P_1 and P_2 meet their first deadlines
- Process P_1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100

Earliest Deadline First Scheduling (EDF)



- Step 3: At time 85, P_2 's second burst, which comes at time 80, begins running, only to be preempted by the third burst of P_1 at the start of its next period at time 100
 - ▣ at time 100, P_2 is preempted, because P_1 has an earlier deadline (time 150) than P_2 (time 160)

Proportional Share Scheduling (比例分享)

- CPU time is divided into T shares and are allocated among all processes in the system
- An application receives N shares where $N < T$, controlled by an admission controller (AC, 接纳控制器)
 - AC can reject processes that cannot guarantee resources, deadline processes
 - e.g. phone call not through
- This ensures each application will receive N / T of the total processor time
- E.g. CPU time is divided into $T=100$ shares
 - process A, B, C and D sequentially need 50, 15, 20, and 30 shares, respectively
 - AC allocates 50, 15, and 20 shares of CPU time to process A, B and C, and rejects the request of 30 shares from process D
 - for A, B, C, $50+15+20=85 < T=100$, their requests are granted
 - for D, its request $30 > 100 - (50+15+20)$, is rejected

Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
 - ▣ Not knowing it does not own the CPUs
 - ▣ Can result in poor response time
 - ▣ Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests

POSIX Real-Time Scheduling

- The POSIX.1b standard provides extensions for real-time computing
 - ▣ its API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads
 - ▣ SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
 - ▣ SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
`pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
`pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

5.7 Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling

Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order $O(1)$ scheduling time
 - Preemptive, priority based
 - Two priority ranges: time-sharing and real-time
 - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
 - Map into global priority with numerically lower values indicating higher priority
 - Higher priority gets larger q
 - Task run-able as long as time left in time slice (**active**)
 - If no time left (**expired**), not run-able until all other tasks use their slices
 - All run-able tasks tracked in per-CPU **runqueue** data structure
 - Two priority arrays (active, expired)
 - Tasks indexed by priority
 - When no more active, arrays are exchanged
 - Worked well, but poor response times for interactive processes

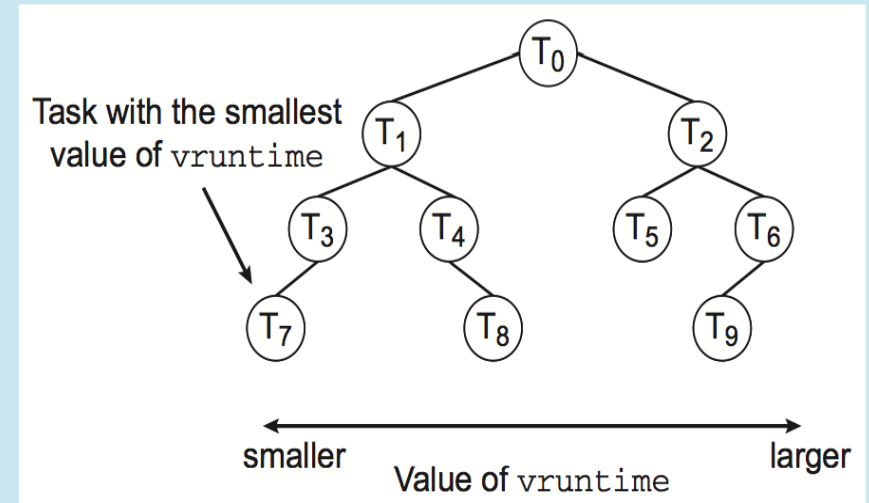
Linux Scheduling in Version 2.6.23 +

- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
 - ❑ Each has specific priority
 - ❑ Scheduler picks highest priority task in highest scheduling class
 - ❑ Rather than quantum based on fixed time allotments, based on proportion of CPU time
 - ❑ 2 scheduling classes included, others can be added
 - Default
 - real-time
- Quantum calculated based on **nice value** from -20 to +19
 - ❑ Lower value is higher priority
 - ❑ Calculates **target latency**
 - interval of time during which task should run at least once
 - ❑ Target latency can increase if say number of active tasks increases

Linux Scheduling in Version 2.6.23 +

- CFS scheduler maintains per task **virtual run time** (虚拟运行时间) in variable **vruntime**
 - ▣ Associated with decay factor based on priority of task
 - lower priority is higher decay rate
 - ▣ Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

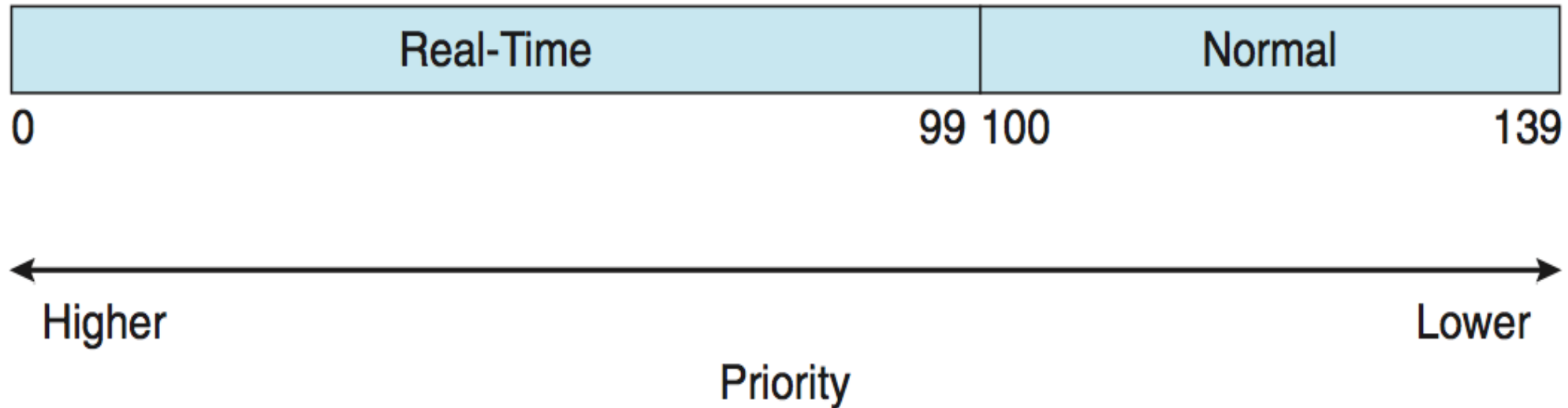


When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

CFS Performance

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
 - ▣ Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
 - ▣ REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
 - ▣ All are variable except REALTIME
- A thread within a given priority class has a relative priority
 - ▣ TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling (UMS)**
 - ▣ Applications create and manage threads independent of kernel
 - ▣ For large number of threads, much more efficient
 - ▣ UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

Windows Priorities

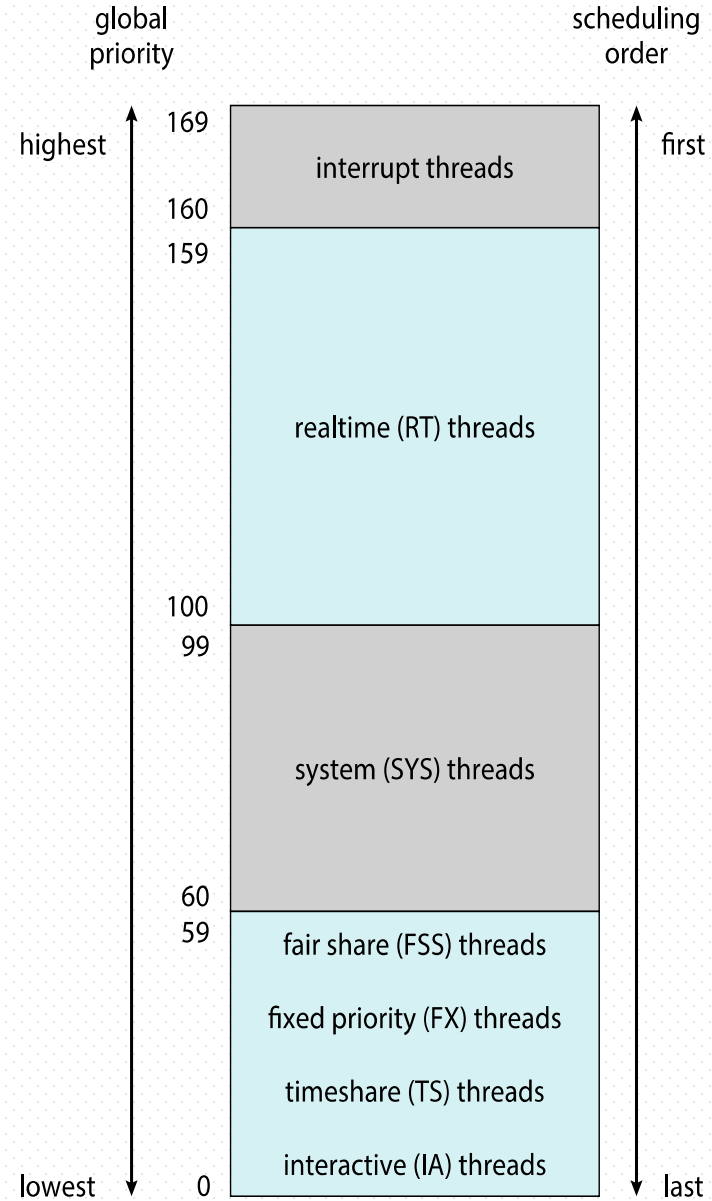
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

- Priority-based scheduling
- Six classes available
 - ▣ Time sharing (default) (TS)
 - ▣ Interactive (IA)
 - ▣ Real time (RT)
 - ▣ System (SYS)
 - ▣ Fair Share (FSS)
 - ▣ Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
 - ▣ Loadable table configurable by sysadmin

Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris Scheduling



Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
 - ▣ thread with highest priority is selected to runs next
 - ▣ thread runs until
 - (1) be blocked,
 - (2) uses time slice, and the slice expires
 - (3) preempted by higher-priority thread
 - ▣ multiple threads at same priority are selected via RR

5.8 Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- Four strategies for evaluating scheduling algorithms
 - deterministic modeling and analytical evaluation
 - taking a particular predetermined **workload** and defines the performance of each algorithm for that workload
 - queuing models
 - Implementing algorithms and applying them to practical systems, observe and evaluate their performance
 - evaluation of CPU Schedulers by simulation

Appendix 5-1 基于指数平均的CPU burst预测

Exponential Averaging prediction of next CPU bursts

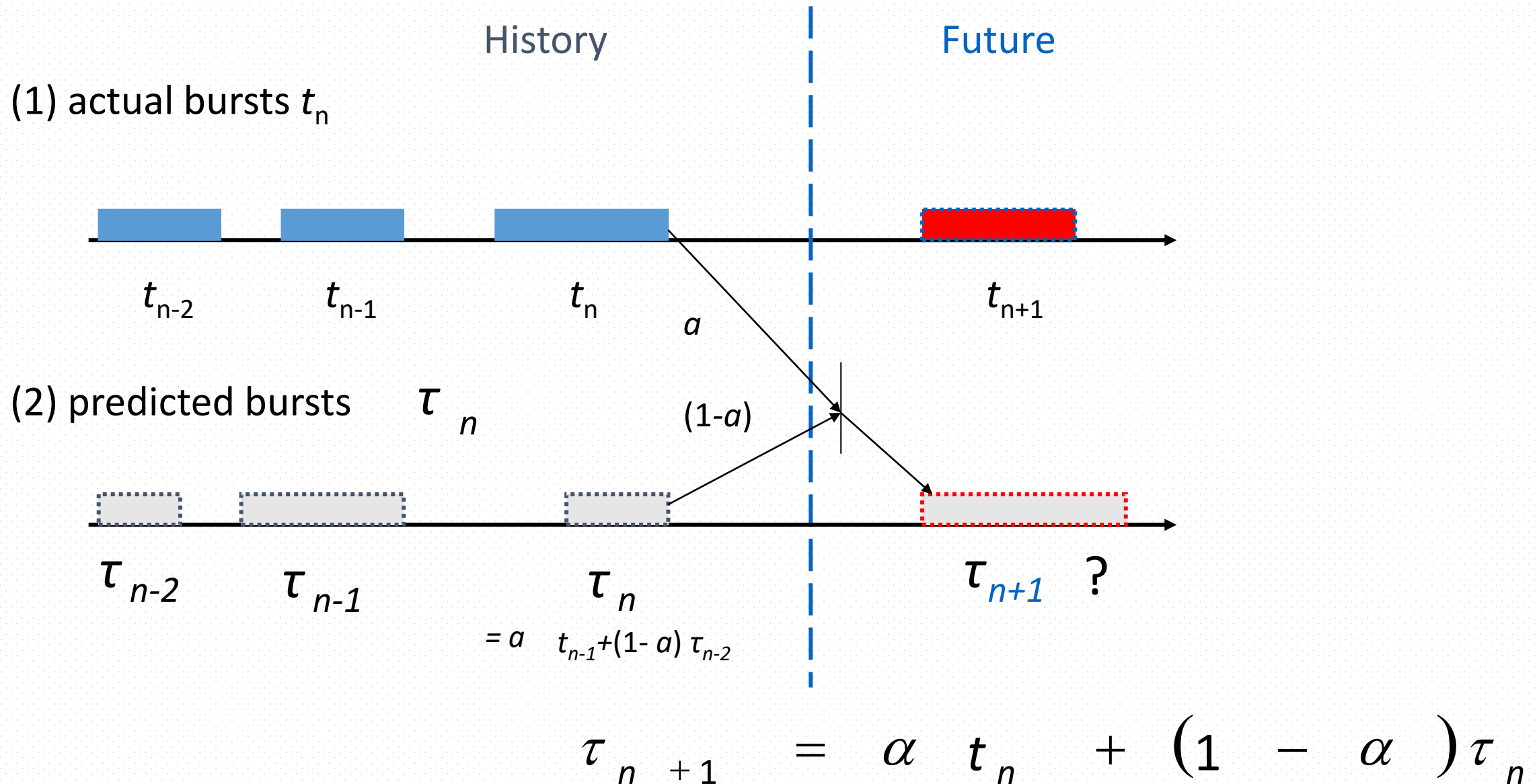
- Define

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$

- Then

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

Exponential Averaging prediction of next CPU bursts



Exponential Averaging prediction of next CPU bursts

- Examples of Exponential Averaging

- $\alpha = 0$

- $\tau_{n+1} = \tau_n$, recent history does not count

- $\alpha = 1$

- $\tau_{n+1} = t_n$, only the actual last CPU burst counts

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

- If we expand the formula, we get

- $\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots$

- $+ (1 - \alpha)^j \alpha t_{n-j} + \dots$

- $+ (1 - \alpha)^{n+1} \tau_0$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Appendix 5-2 Priority-based scheduling in Unix

- 基于优先级/权/数的调度
 - 广泛采用的一种进程调度算法
 - 由系统（或用户、或系统与用户结合）赋予每个进程一个优先数，在处理机空闲或调度程序启动时，进程调度程序从就绪进程中选择一个优先数最大（或者最小）的进程占用CPU（该进程就从就绪状态转换成运行状态）
- 关键
 - 如何确定进程的优先数
 - 进程的优先数确定之后是固定的，还是随着该进程运行的情况的变化而变化

Priority-based scheduling in Unix

■ 两种方法确定优先级

▣ 1. 静态优先级

- 进程的优先数在进程创建时确定后就不再变化
- 进程优先数确定

(1) 系统确定：运行时间、使用资源，进程的类型

(2) 用户确定：紧迫程度，计费与进程优先数有关

(3) 系统与用户结合：

用户可以为本用户进程设置优先数，但不作为调度的唯一依据，系统将用户设置的进程优先数作为确定进程优先级的一个参数

▣ 2. 动态优先级

- 系统在运行的过程中，根据系统的设计目标，不断地调整进程的优先数
- 优点：客观地反映进程的实际情况，保证达到系统设计目标

Priority-based scheduling in Unix

■ UNIX系统的进程调度

- ❑ UNIX系统采用优先数调度算法，每个进程有一个进程优先数`p_pri`，`p_pri`是`proc`结构中的一个变量，其取值范围是 $-127 \sim 127$ ，其值越小，进程优先级越高
- ❑ 调度程序从就绪状态的进程中，选择一个优先数最小、优先级最高的进程占用CPU

■ 进程优先数计算公式

$$p_pri = \min \{127, (p_cpu/16 + p_nice + PUSER) \}$$

- ❑ `p_cpu` 进程占用CPU的程度，到目前为止已经在CPU上运行的时间
- ❑ `p_nice` 用户通过系统调用`nice (priority)` 设置的进程优先数
- ❑ `PUSER` 常数，其值为100

Priority-based scheduling in Unix

- 进程进入高优先级睡眠的原因
 - ▣ 0 # 进程 (- 100优先数)
 - ▣ 因资源请求得不到满足的进程, 磁盘 (- 80) , 打印机 (- 20) , ...;
 - ▣ 等待块设备I/O完成的进程, (- 50)
- 进程进入低优先级睡眠的原因
 - ▣ 因等待字符设备I/O完成的进程, (0 ~ 20的优先数)
- 处于用户态运行进程, 其优先数一般情况下大于100



Thanks for your
attention



北京邮电大学