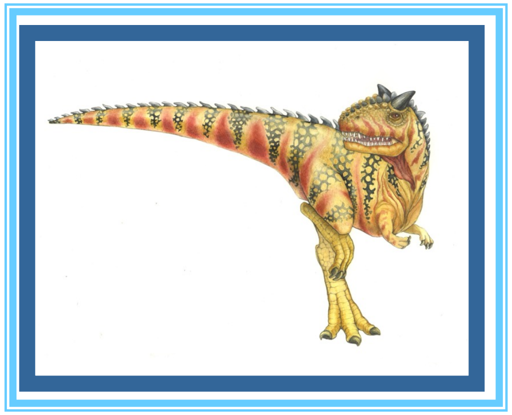
Chapter 5: CPU Scheduling



Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation

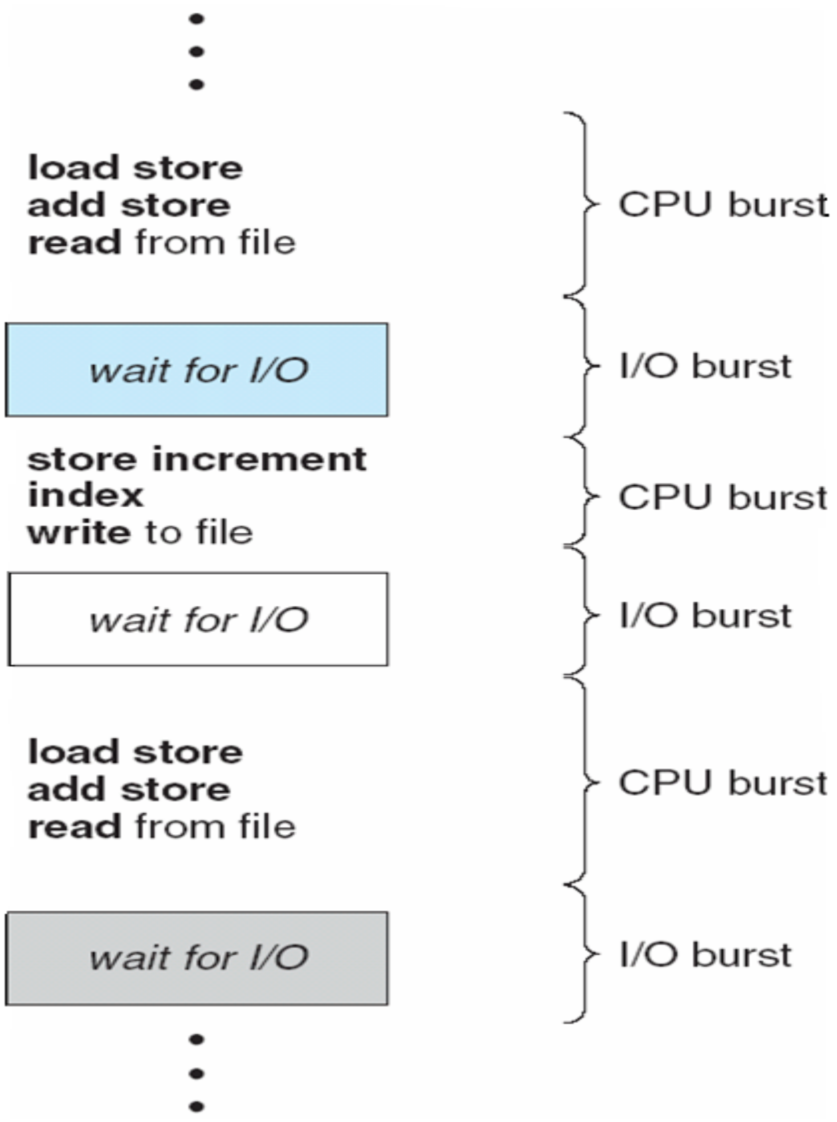
Objectives

- 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;

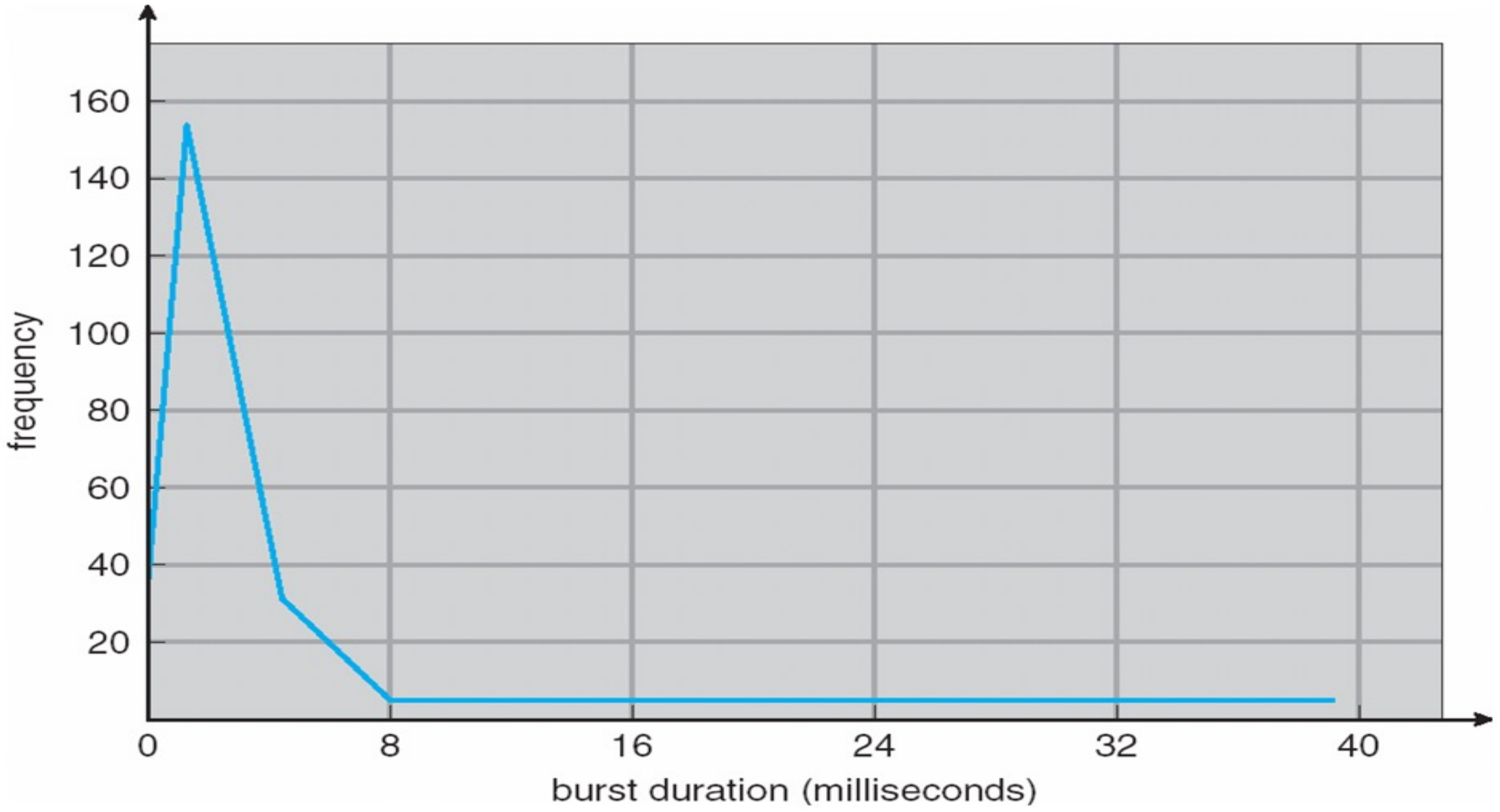
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming ;
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait ;
- **CPU burst** distribution ;

Alternating Sequence of CPU And I/O Bursts



Histogram of CPU-burst Times



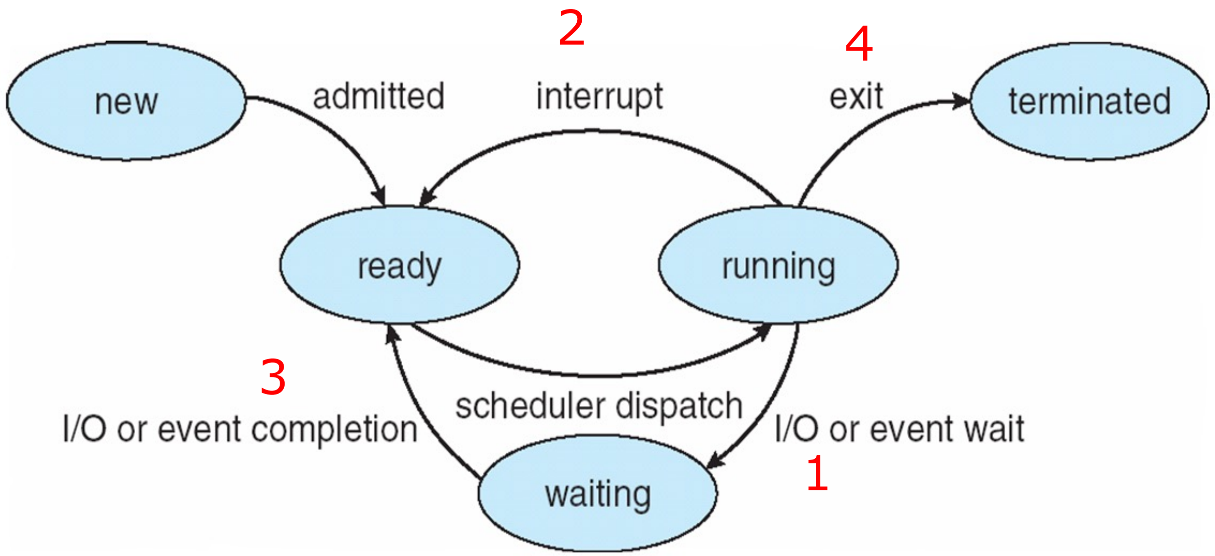
- An I/O-bound program typically has many short CPU bursts
- A CPU-bound program might have a few long CPU bursts.

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them;
- Many processes in “ready” state;
- Which ready process to pick to run on the CPU?

CPU Scheduler

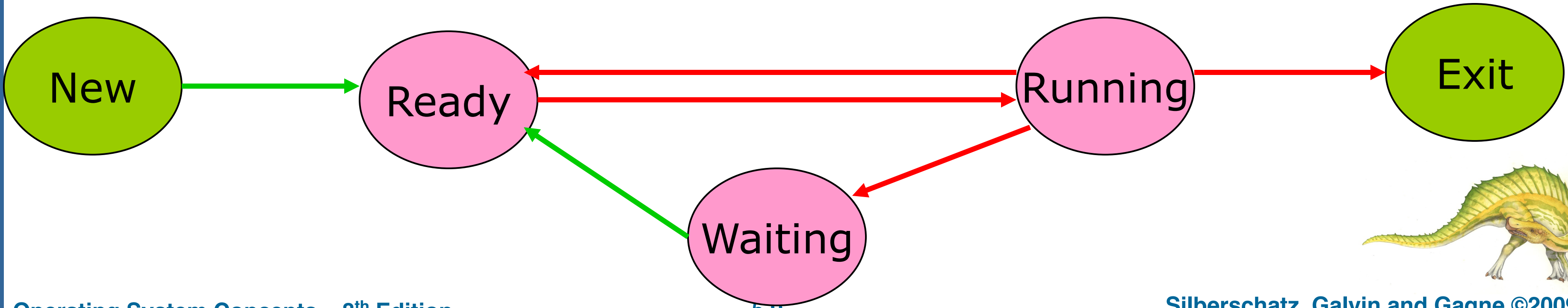
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates



- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

When does scheduler run?

- Non-preemptive minimum
  - Process runs until voluntarily relinquish CPU
    - process blocks on an event (e.g., I/O or synchronization)
    - process terminates
    - process yields
- Preemptive minimum
  - All of the above, plus:
    - Event completes: process moves from blocked to ready
    - Timer interrupts
    - Implementation: process can be interrupted in favor of another



Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput(吞吐量)** – # of processes that complete their execution per time unit
- **Turnaround time(周转时间)** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
- Fairness: everyone makes progress, no one starves

"The perfect CPU scheduler"

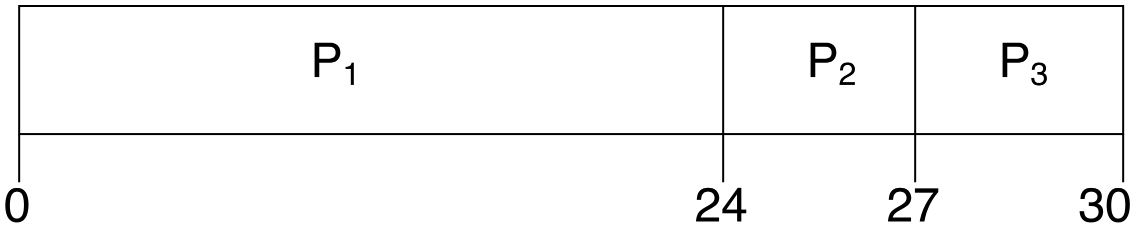
Problem Cases

- Blindness about job types
  - I/O goes idle
- Optimization involves favoring jobs of type “A” over “B”.
  - Lots of A’s? B’s starve
- Interactive process trapped behind others.
  - Response time sucks for no reason
- Priority Inversion: A depends on B. A’s priority > B’s.
  - B never runs

First-Come, First-Served (FCFS) Scheduling

Process	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:



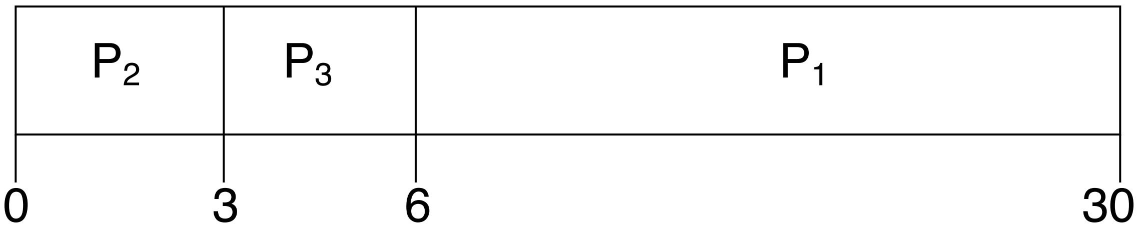
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont)

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
- *Convoy effect*: short process behind long process

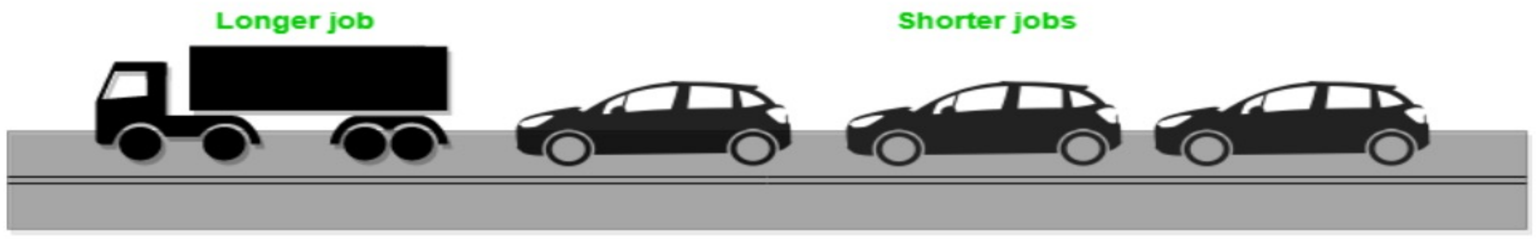


Figure - The Convey Effect, Visualized

Convoy Effect

- A CPU bound job will hold CPU until done,
  - or it causes an I/O burst
    - rare occurrence, since the thread is CPU-bound

  ⇒ long periods where no I/O requests issued, and CPU held

  - Result: poor I/O device utilization
- Example: one CPU bound job, many I/O bound(希望I/O型多运行)
    - CPU bound runs (I/O devices idle)
    - CPU bound blocks
    - I/O bound job(s) run, quickly block on I/O
    - CPU bound runs again
    - I/O completes
    - CPU bound still runs while I/O devices idle (continues…)

Scheduling Algorithms LIFO

- **Last-In First-out (LIFO)**
  - Newly arrived jobs are placed at head of ready queue
  - Improves response time for newly created threads
- **Problem:**
  - May lead to starvation – early processes may never get CPU

Problem

- You work as a short-order cook
  - Customers come in and specify which dish they want
  - Each dish takes a different amount of time to prepare
- Your goal:
  - minimize average time the customers wait for their food
- What strategy would you use ?
  - Note: most restaurants use FCFS.

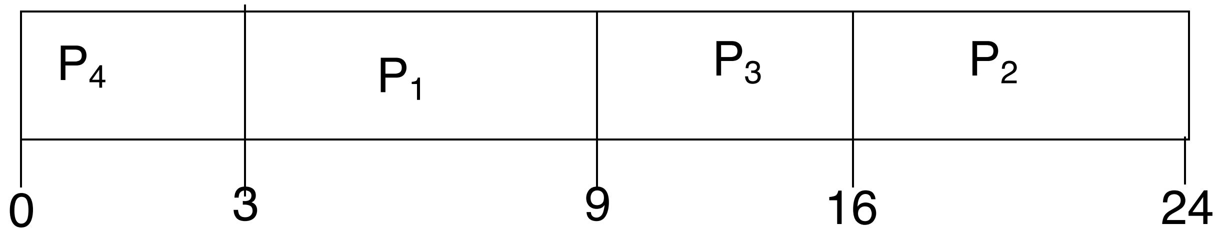
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request

Example of SJF

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

➢ **Problem:**
Impossible to know the length of the next CPU burst

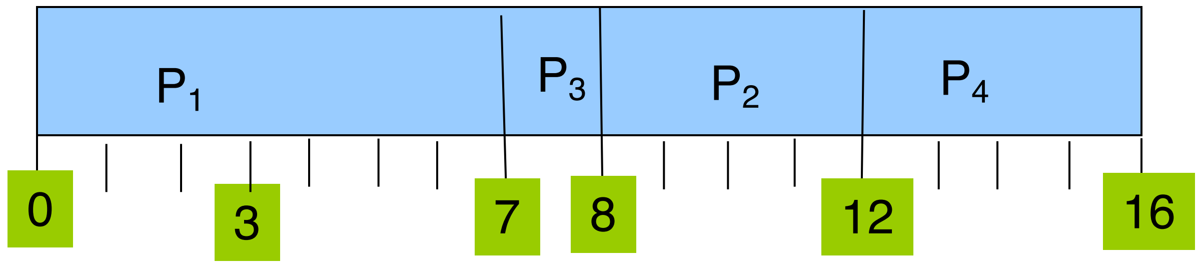
SJF Scheduling(con.)

- SJF can be either preemptive or non-preemptive
  - New, short job arrives; current process has long time to execute

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



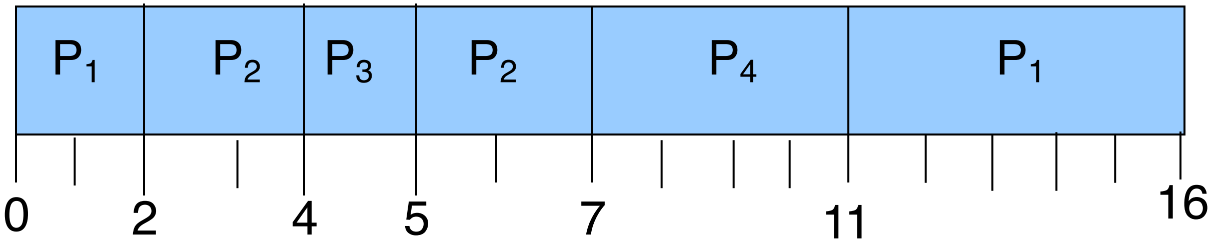
- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Preemptive SJF is called ***shortest remaining time first(SRTF)***

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



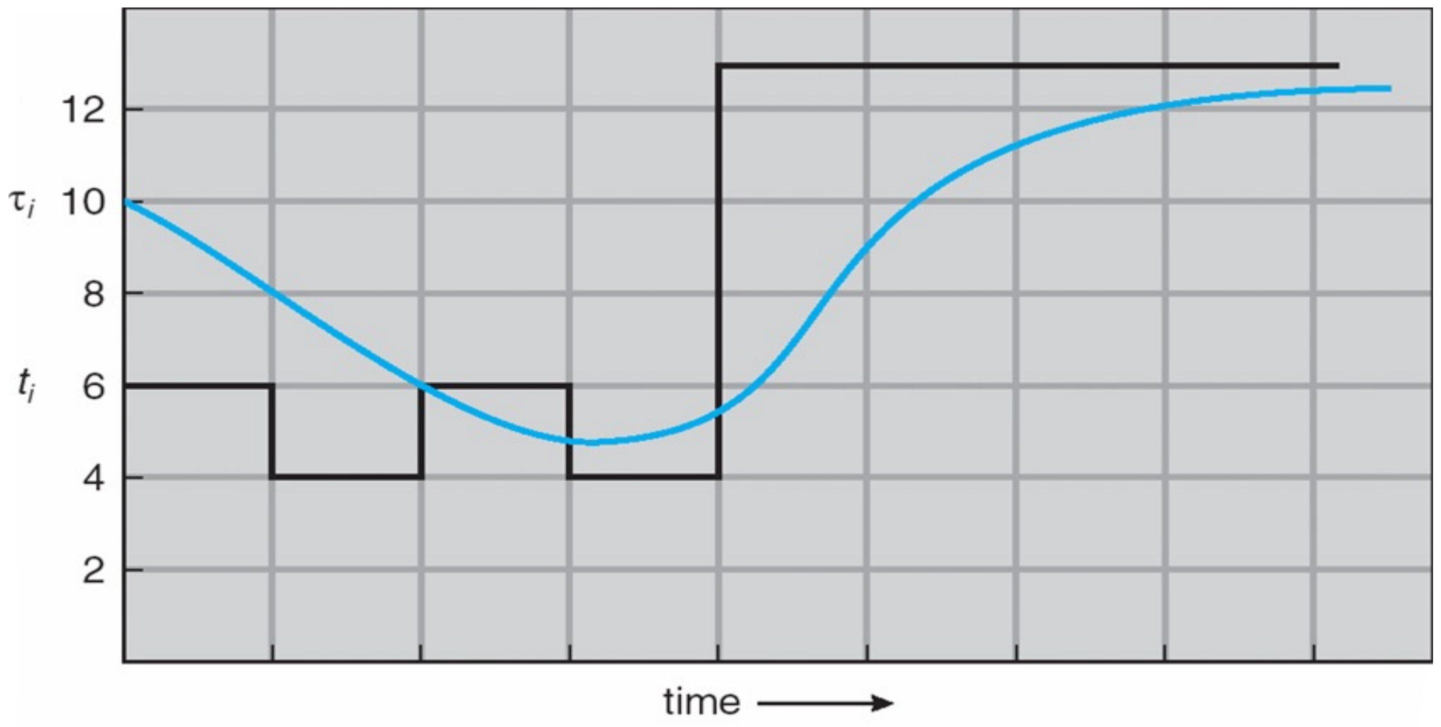
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging（指数平均法）

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count（不考虑实际值）
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\ t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} &= \alpha\ t_n + (1 - \alpha)\alpha\ t_n - 1 + \ldots \\ &+ (1 - \alpha)^j \alpha\ t_{n-j} + \ldots \\ &+ (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

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

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Priority Scheduling

- OS Solutions to process starvation
  - Decreasing priority & aging: the Unix approach
    - Decrease priority of CPU-intense process
    - Exponential averaging of CPU usage to slowly increase priority of blocked process
    - p_pri=min{127, (p_cpu/16+PUSER+p_nice)},
  - Priority Elevation: the Windows approach
    - Increase priority of a thread on I/O completion
    - System gives starved threads an extra burst

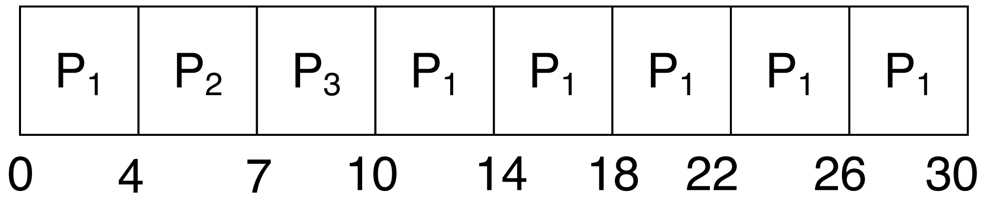
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*). After this time has elapsed, the process is preempted and added to the end of the ready queue.
- There are n processes in the ready queue and the time quantum is q. No process waits more than $(n-1)q$ time units.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

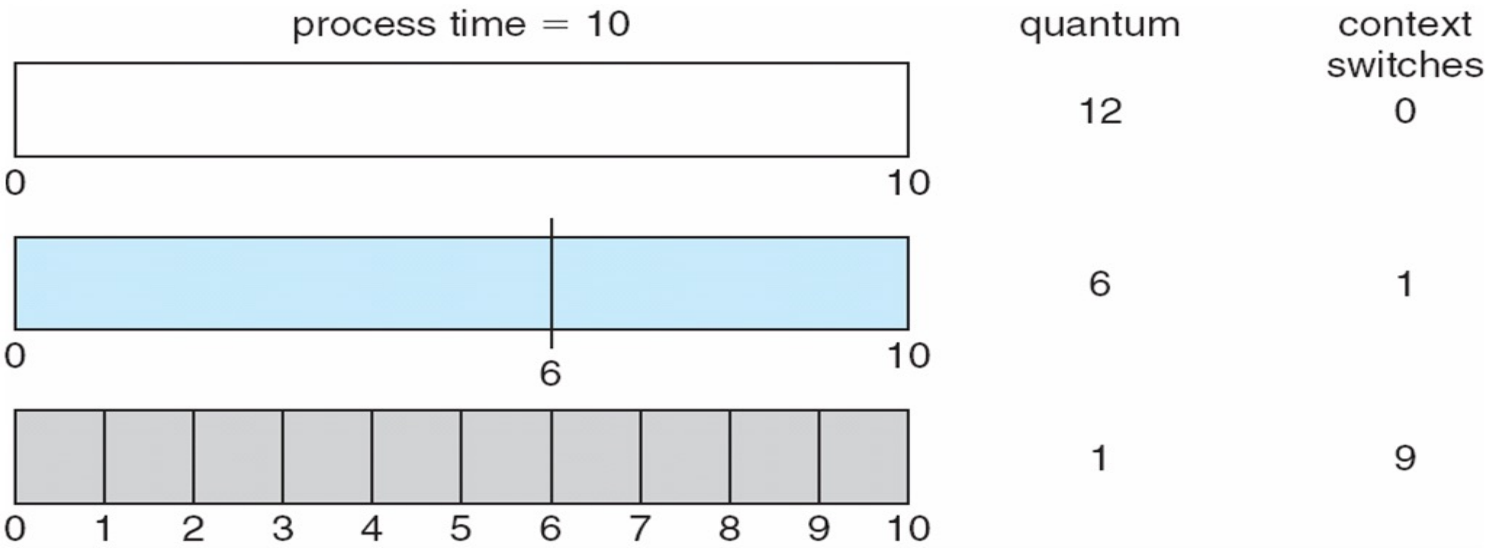
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



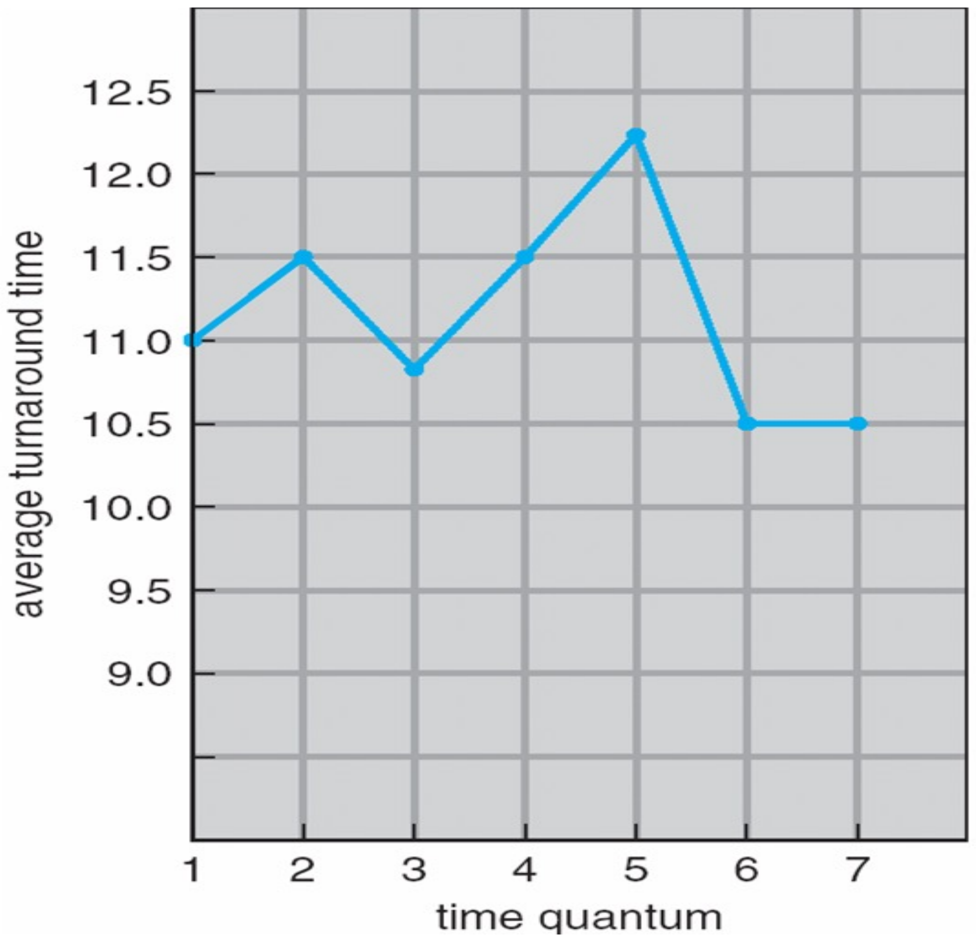
- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



- Performance depends on length of the timeslice
  - Context switching isn't a free operation（ overhead ）.
  - If timeslice time is set too high
    - attempting to amortize context switch cost, you get FCFS.
  - If it's set too low
    - you're spending all of your time context switching between threads.
  - Timeslice frequently set to 10~100 milliseconds
  - Context switches typically cost < 1 millisecond

Turnaround Time Varies With The Time Quantum



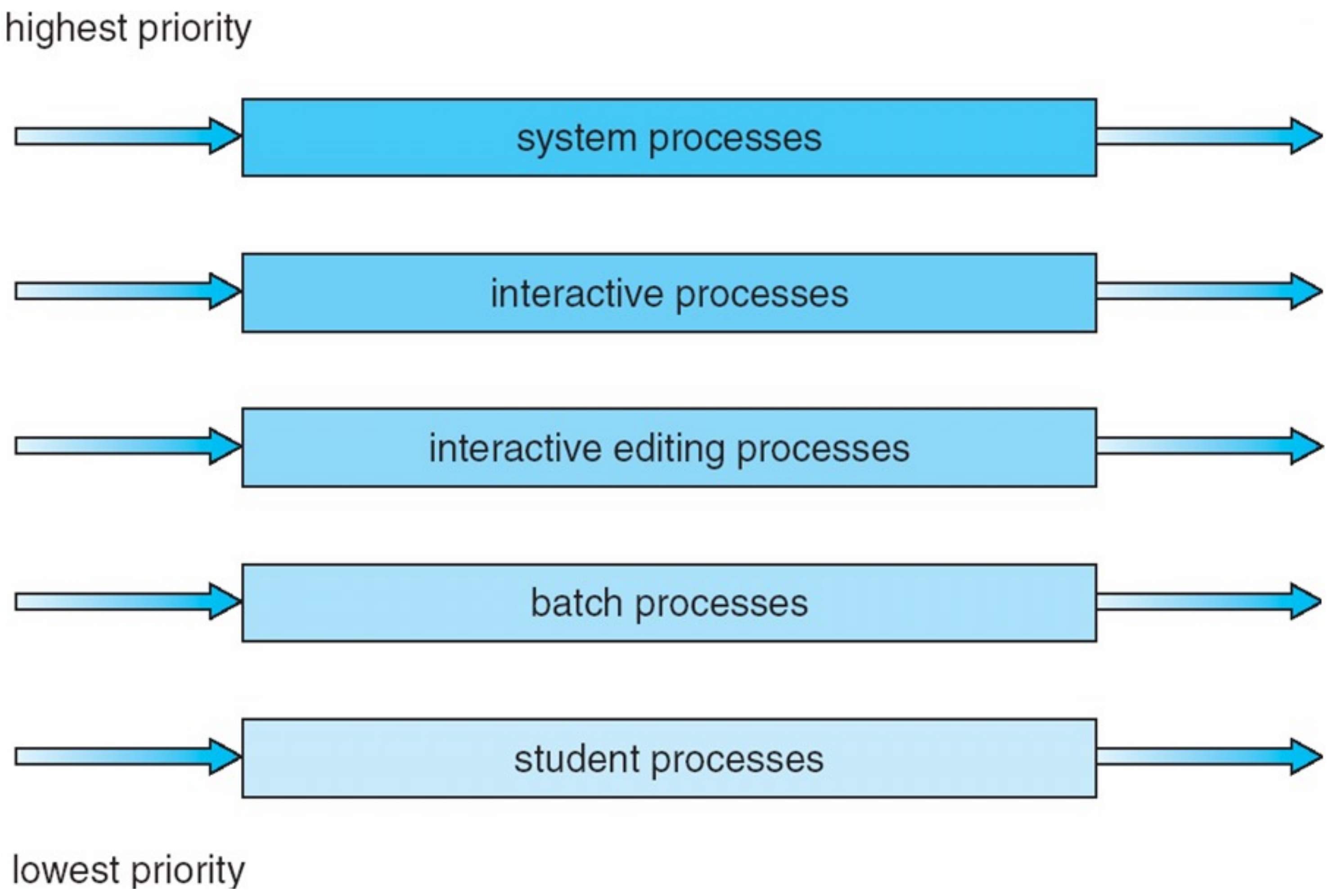
process	time
P_1	6
P_2	3
P_3	1
P_4	7

周转时间随时间片变化而变化

Multilevel Queue

- Ready queue is partitioned into separate queues:
  1.foreground (interactive)
  2.background (batch)
- Each queue has its own scheduling algorithm
  - ✓ foreground – RR
  - ✓ background – FCFS
- Scheduling must be done between the queues
  - ✓ Fixed priority scheduling; (i.e., 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;
  - ✓ i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling

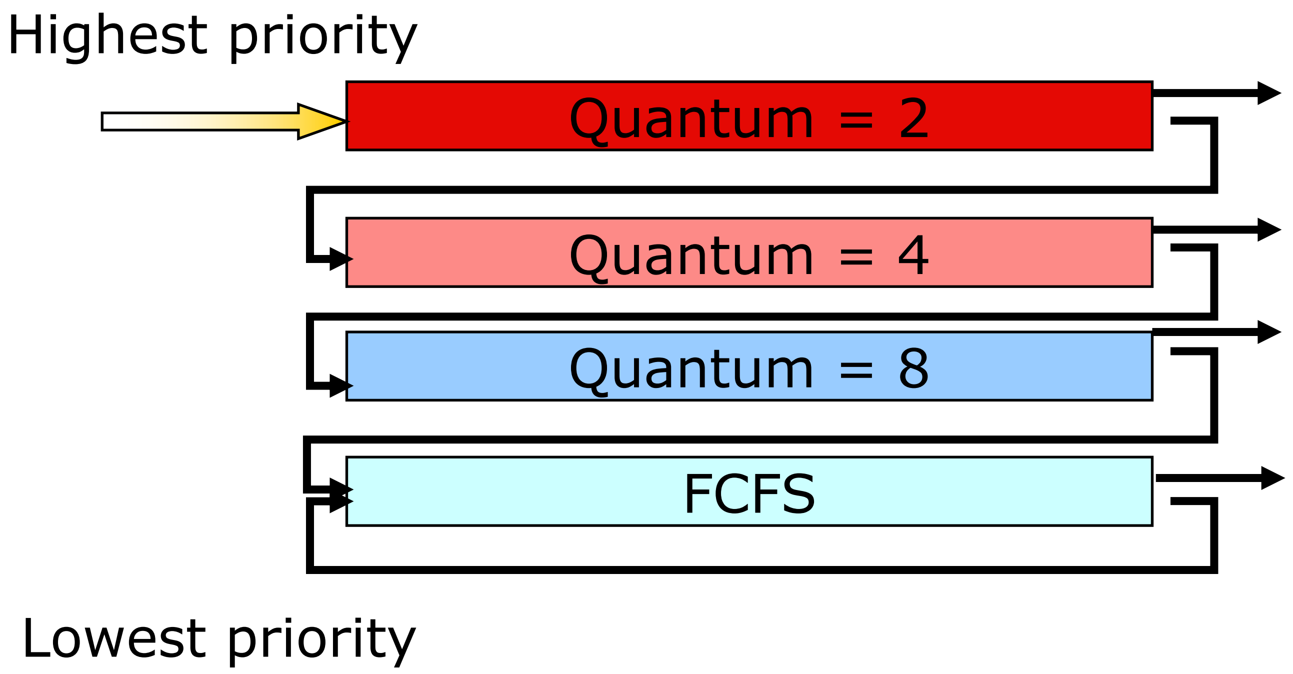


Multilevel Feedback Queue

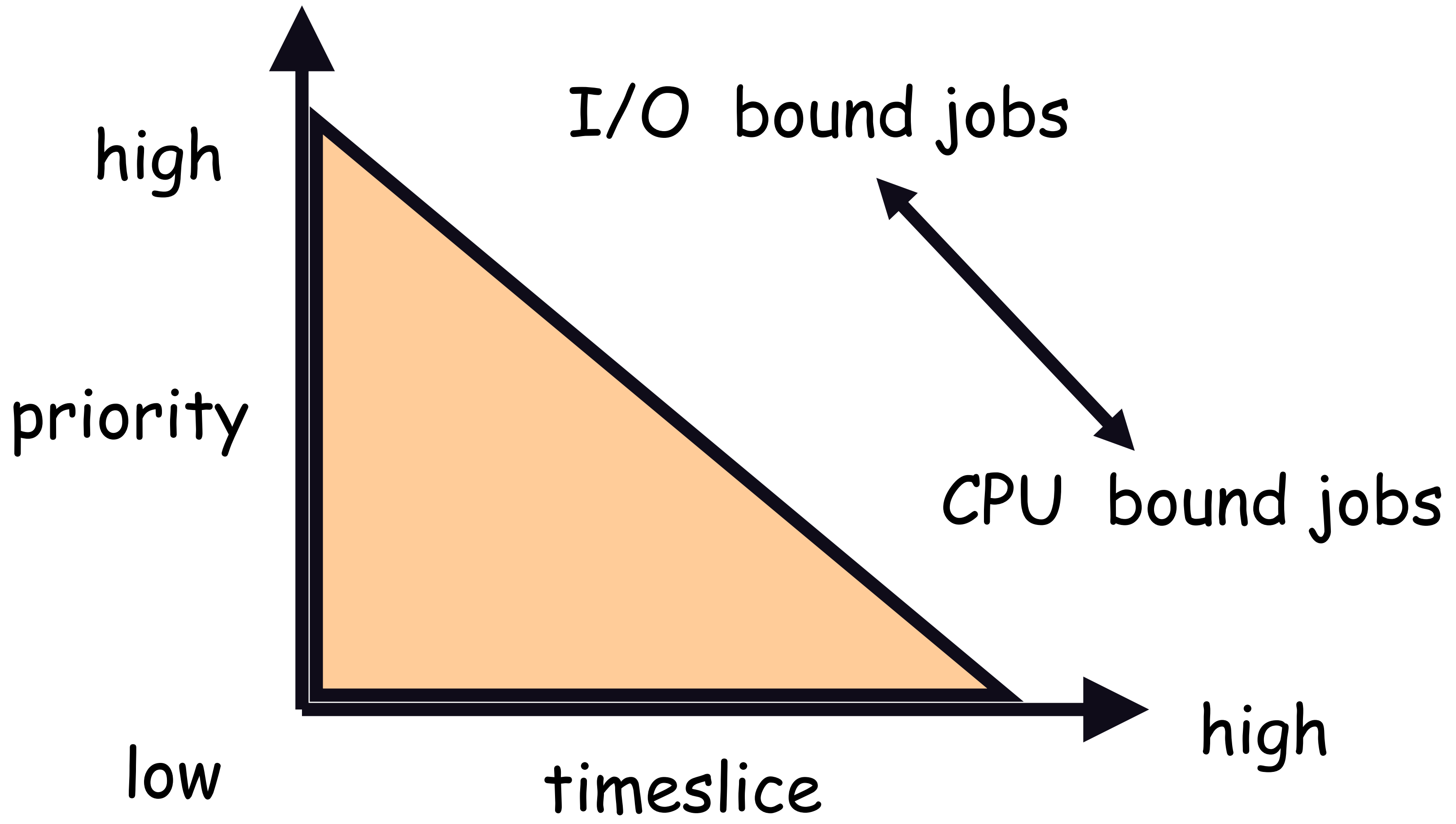
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service（服务哪个队列）

Example of Multilevel Feedback Queue

- Four queues:
  - Q_0 – RR with time quantum 2 milliseconds
  - Q_1 – RR time quantum 4 milliseconds
  - Q_2 – RR time quantum 8 milliseconds
  - Q_3 – FCFS



A Multi-level System



Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process **(**进程内竞争**)**
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system **(**系统内竞争**)**

Pthread Scheduling

- 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.
- POSIX:
  - PTHREAD SCOPE PROCESS/PTHREAD SCOPE SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{    int i;
   pthread t tid[NUM THREADS];
   pthread attr t attr;        /* get the default attributes */
   pthread attr init(&attr);/* set the scheduling algorithm to PROCESS or SYSTEM
   */
   pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);
                                             /* set the scheduling policy - FIFO,
   RT, or OTHER */
   pthread attr setschedpolicy(&attr, SCHED OTHER);/* create the threads */
   for (i = 0; i < NUM THREADS; i++)   pthread
   create(&tid[i],&attr,runner,NULL);/* now join on each thread */
   for (i = 0; i < NUM THREADS; i++)   pthread join(tid[i], NULL);
} /* Each thread will begin control in this function */
void *runner(void *param)
{
   printf("I am a thread\n");
   pthread exit(0);
```

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity(关联) for processor on which it is currently running
  - **soft affinity(**试图使进程在同一个CPU上运行, 但不保证)；
  - **hard affinity(**不允许进程在不同CPU间迁移)；

Processor affinity例子

1.通过命令行设置进程在哪个处理器执行(Linux)

例如，在处理器0或4上执行进程9030，采用命令：　$ taskset -cp 0,4 9030

2.库函数 **sched_setaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask)**

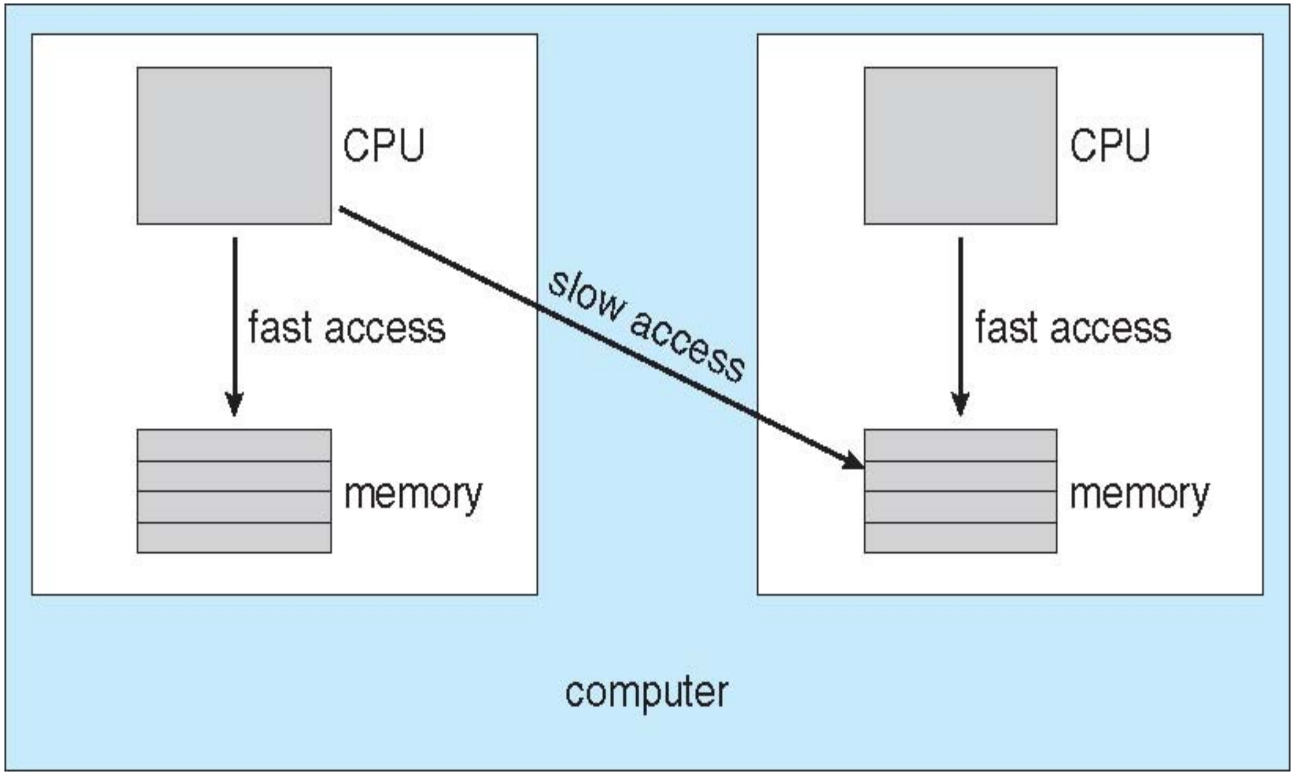
pid为进程（线程）号；cpusetsize为一个结构体大小; mask为掩码，标识在哪个CPU执行进程

```
#include <stdio.h> //我们课题组OSV虚拟机监控器子域的启动代码，让进程0在处理器0上执行;
#include <sched.h>
#include <unistd.h>
int main(int argc, char **argv)
{   int  opt, cpuid = 0, domainid = 1;
    if(argc <= 4)
       printf("args incomplete, use cpuid=0 domainid = 1 for default\n\n");
    ... ...
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(cpuid, &mask);
    if(sched_setaffinity(0, sizeof(cpu_set_t), &mask) == 0)
        printf("affinity set succeed.\n");   ... ...
}
```

Multiple-Processor

- NUMA(Non-Uniform Memory Access)和SMP(Symmetric Multi-Processor)是两种不同的CPU硬件体系架构；
- SMP的主要特征是共享，所有的CPU共享使用全部资源：内存、总线和I/O等，多个CPU对称工作，没有主次之分，平等地访问共享的资源，会引起资源竞争问题，引起扩展能力的问题；
- NUMA技术将CPU划分成不同的组（Node)，每个Node由多个CPU组成，有独立的本地内存、I/O等资源。Node之间通过互联模块连接，每个CPU可访问本地内存和远端Node的内存，访问远端内存效率会差一些，用Node之间的距离（Distance）来定义各个Node之间互访资源的开销。

NUMA and CPU Scheduling



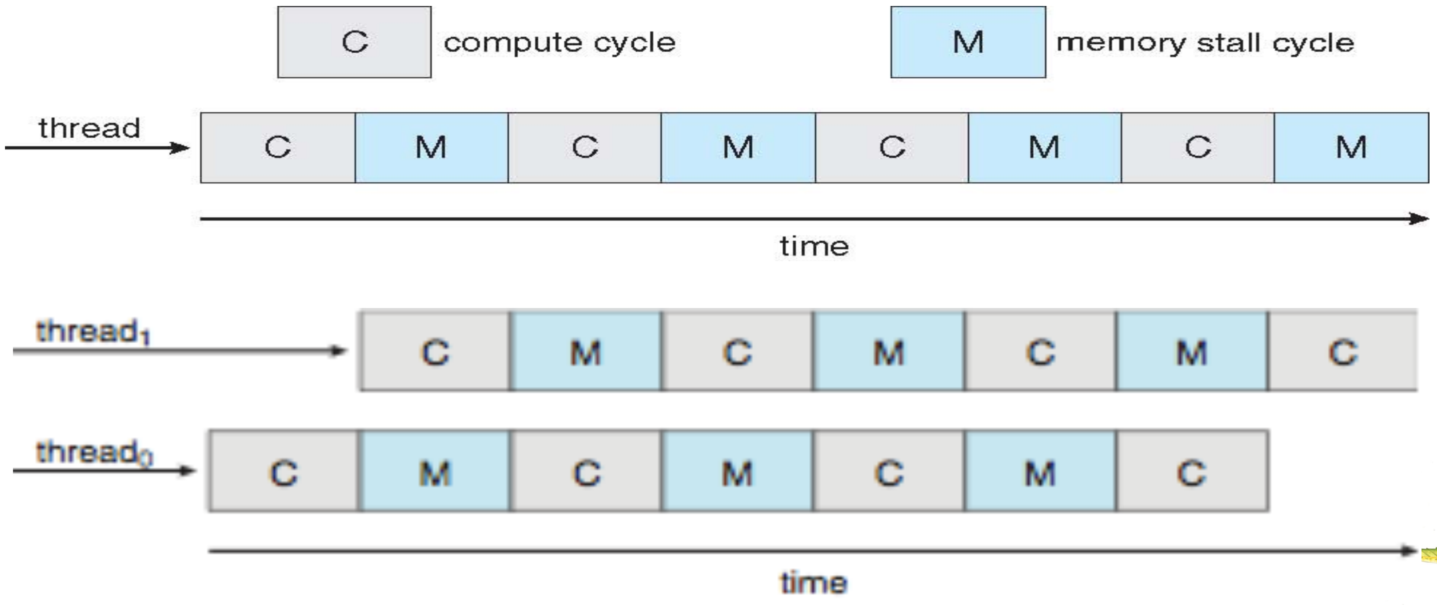
NUMA（Non Uniform Memory Access Achitecture）

NUMA架构在逻辑上遵循对称多处理(SMP)架构。

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip,Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens(利用存储器的延迟，当发生内存检索时另一个线程运行)

Multithreaded Multicore System



Thread Scheduling

Since all threads share code & data segments

- Option 1: Ignore this fact
- Option 2: Gang(组) scheduling
  - run all threads belonging to a process together (multiprocessor only)
  - if a thread needs to synchronize with another thread
    - the other one is available and active
- Option 3: Two-level scheduling:
  - Medium level scheduler
  - schedule processes, and within each process, schedule threads
  - reduce context switching overhead and improve cache hit ratio
- Option 4: Space-based affinity（数据关联的一组线程）：
  - assign threads to processors (multiprocessor only)
  - improve cache hit ratio, but can bite under low-load condition

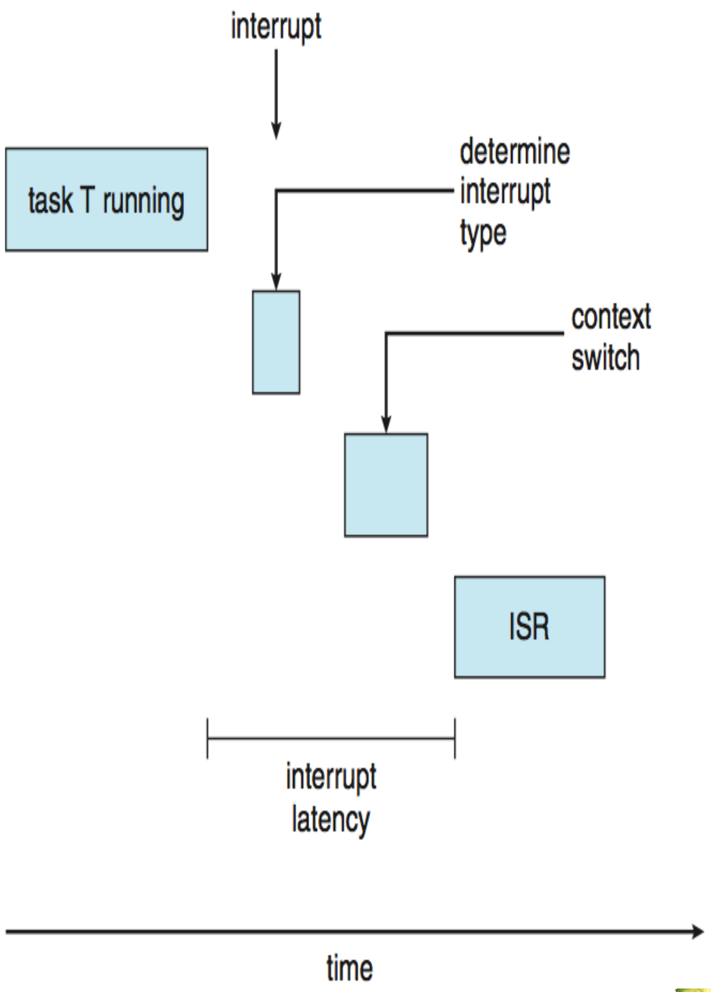
What are Real-time Systems ?

- **Real-time systems**

  Those systems in which the **correctness** of the system depends **not only** on the the **logical result** of computation, **but also** on the **time** at which the results are produced.

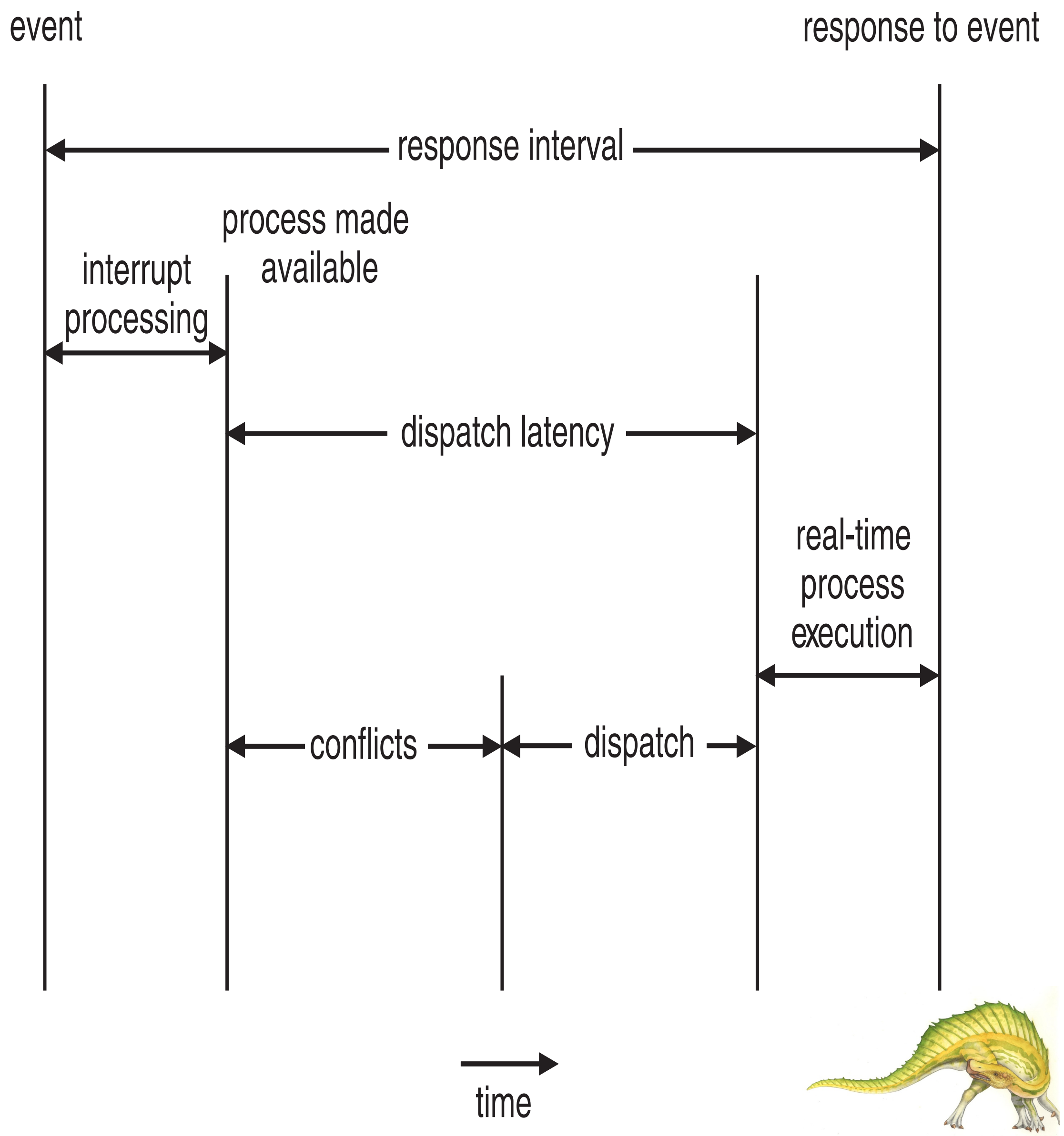
Real-Time CPU Scheduling

- Can present obvious challenges
- **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
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes



Dispatch Latency

- *To keep dispatch latency low ,we need to allow system calls to be preemptible.*为降低分派延迟，允许系统调用被抢占
  - One is to insert preemption points in long-duration system calls.preemption points can be placed at only safe locations in the kernel.一种方法是在长系统调用中插入抢占点(抢占点只能被插在内核中安全的位置)
  - Another method for dealing with preemption is to make the entire kernel preemptible. All kernel data structures must be protected through the use of various synchronization mechanisms.另一种方法是使得整个内核可被抢占,但所有内核数据结构必须通过各种同步机制加以保护。

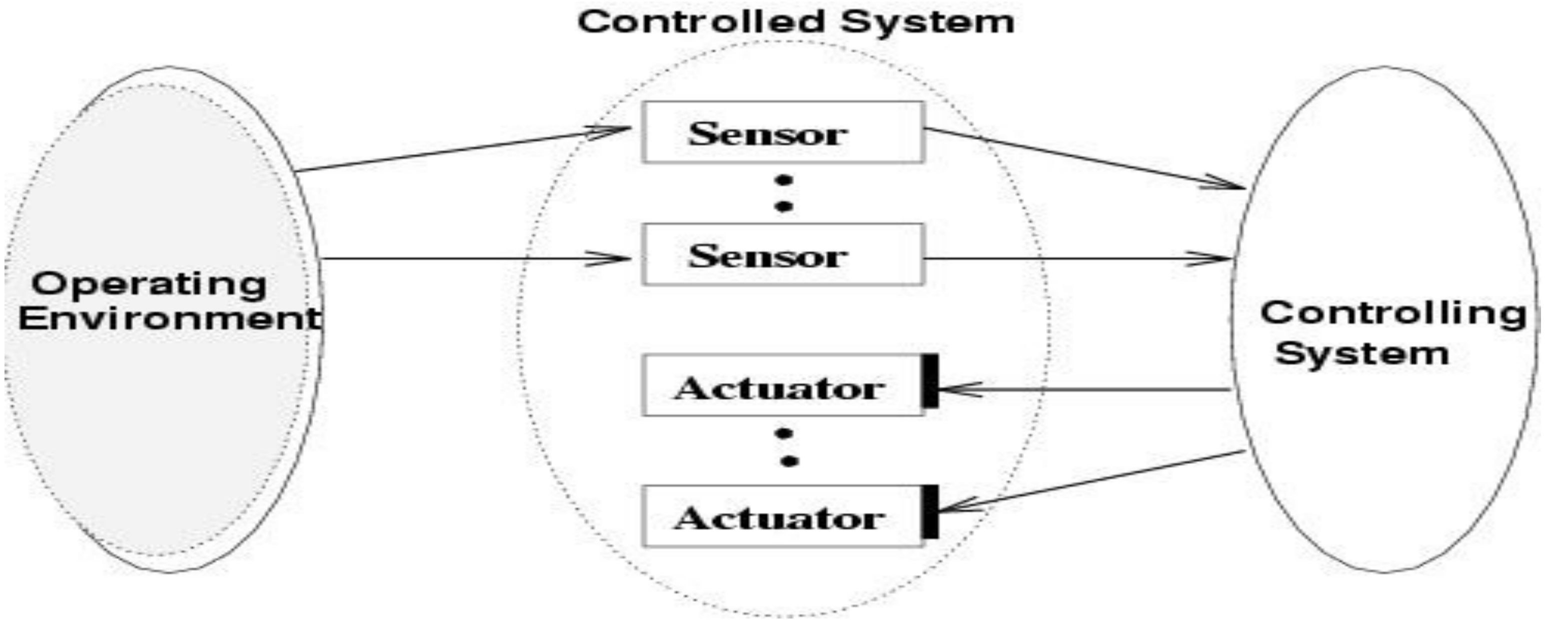
Dispatch Latency

- 优先级倒挂：一个低优先级进程持有一个被高优先级进程认为所需的一个共享资源，比如内核数据，高优先级进程需要等待低优先级进程的完成。
- 该现象可能引起严重后果．例如，美国的火星探测器Mars Pathfinder就是因为优先级倒挂而出现故障．
- 解决方案:Priority-inheritance protocol优先级继承:

  (正在访问高优先级进程所需资源的)低优先级进程继承高优先级，直到其释放共享资源，它的优先级再返回原来的值．
- 其它：系统可重入问题等。

Real-Time Characteristics

- Real-time systems often are comprised of a *controlling* system, *controlled* system and *environment*.
  - *Controlling* system: acquires information about environment using *sensors* and controls the environment with *actuators*.



- *Timing constraints* derived from *physical* impact of controlling systems activities. Hard and soft constraints.
  - Periodic Tasks: Time-driven recuring at regular intervals.
  - Aperiodic: event-driven.

A Sample Real Time System (1)

- ***Mission:*** Reaching the destination safely.
- **Controlled System:** Car.
- **Operating environment:** Road conditions.
- **Controlling System**

  - *Human driver:* Sensors - Eyes and Ears of the driver.

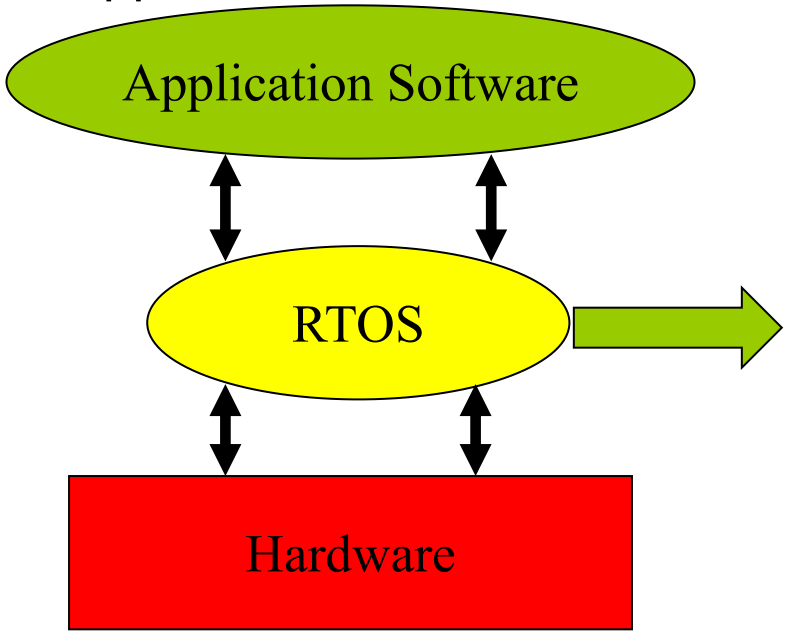
  - *Computer:* Sensors - Cameras, Infrared receiver(红外探测), and Laser telemeter(测量垂直距和倾斜角).

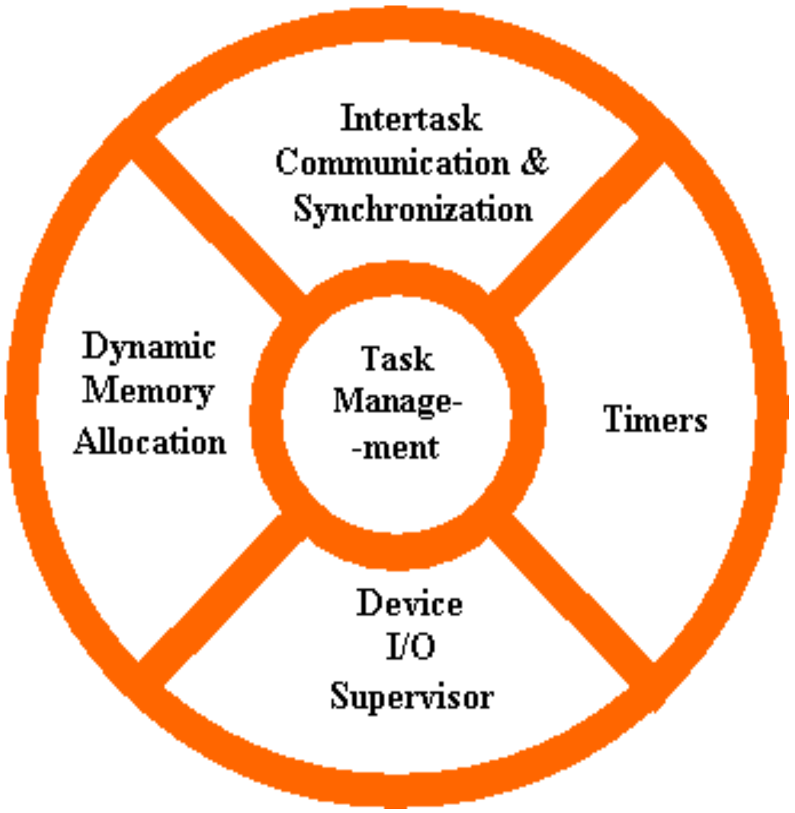
A Sample Real Time System (2)

- **Controls:** Accelerator, Steering wheel, Break-pedal.
- **Actuators:** Wheels, Engines, and Brakes.
- **Critical tasks:** Steering and breaking （Hard constraints）
- **Non-critical tasks:** Turning on radio
  - ✓ **Performance** is not an absolute one.
  - ✓ **Cost of fulfilling the mission** → Efficient solution.
  - ✓ **Reliability of the driver** → Fault-tolerance is a must.

RTOS Kernel

- RTOS Kernel provides an Abstraction layer that hides from application software the hardware details of the processor / set of processors upon which the application software shall run.




Intertask Communication & Synchronization
Dynamic Memory Allocation
Task Manage- -ment
Timers
Device I/O Supervisor

- Main service offered is Task Scheduling
  - controls the execution of application software tasks.
  - can make them run in a very timely and responsive fashion.

Task Scheduling

- Non Real -time systems usually use Non-preemptive Scheduling
  - Once a task starts executing, it completes its full execution
- Most RTOS perform priority-based preemptive task scheduling.
  - The Highest Priority Task that is Ready to Run, will be the Task that Must be Running.

Task Switch

- Each time the priority-based preemptive scheduler is alerted by an External world trigger / Software trigger, it shall go through the following steps that constitute a **Task Switch**:

  ① Determine whether the currently running task should continue to run. determine which task should run next.

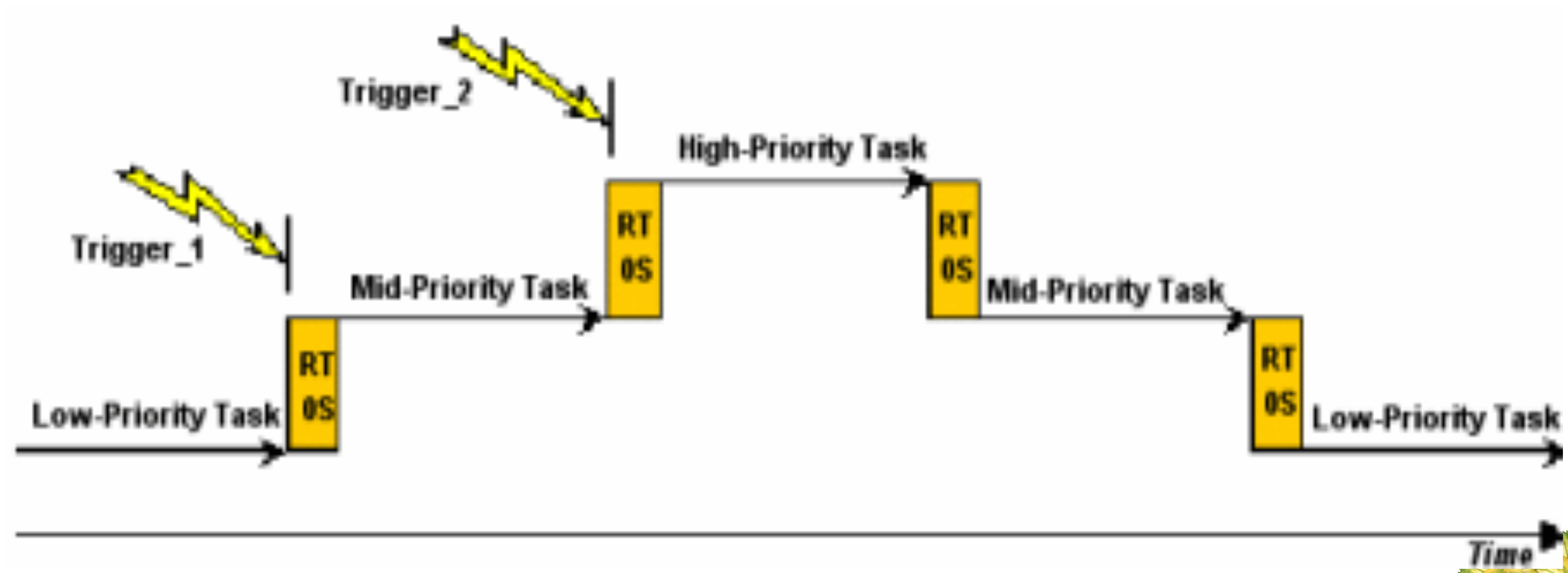
  ② Save the environment of the task that was stopped (so it can continue later).

  ③ Set up the running environment of the task that will run next.

  ④ Allow the selected task to run.

Priority-based Scheduling

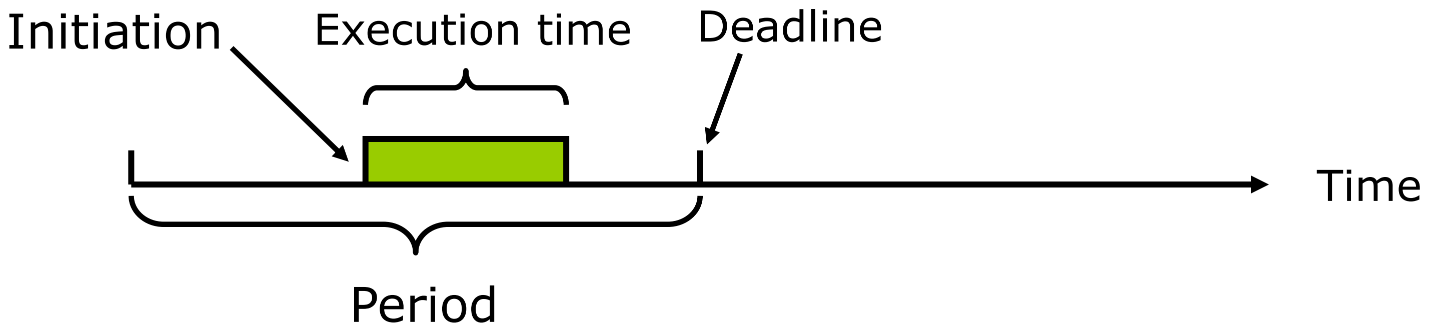
- Typical RTOS based on fixed-priority preemptive scheduler， assign each process a priority.
- At any time, scheduler runs highest priority process ready to run, Higher Priority = Higher Need for Quick Response.
- Process runs to completion unless preempted.
- Follows nested preemption.



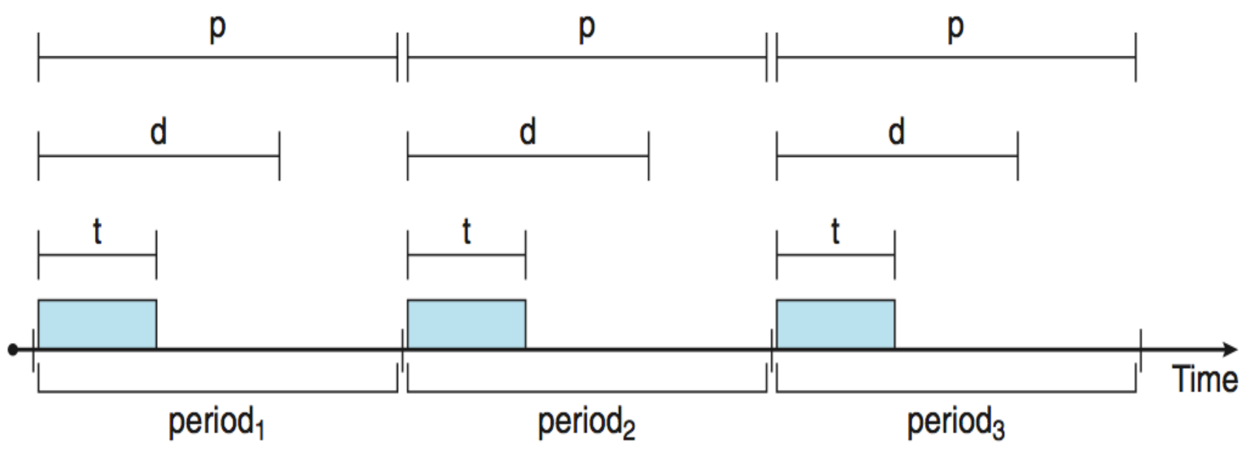
Timeline for Priority-based Preemptive Scheduling

Typical RTOS Task Model

- Each task a triplet: (execution time, period, deadline)
- Usually, deadline = period
- Can be initiated any time during the period

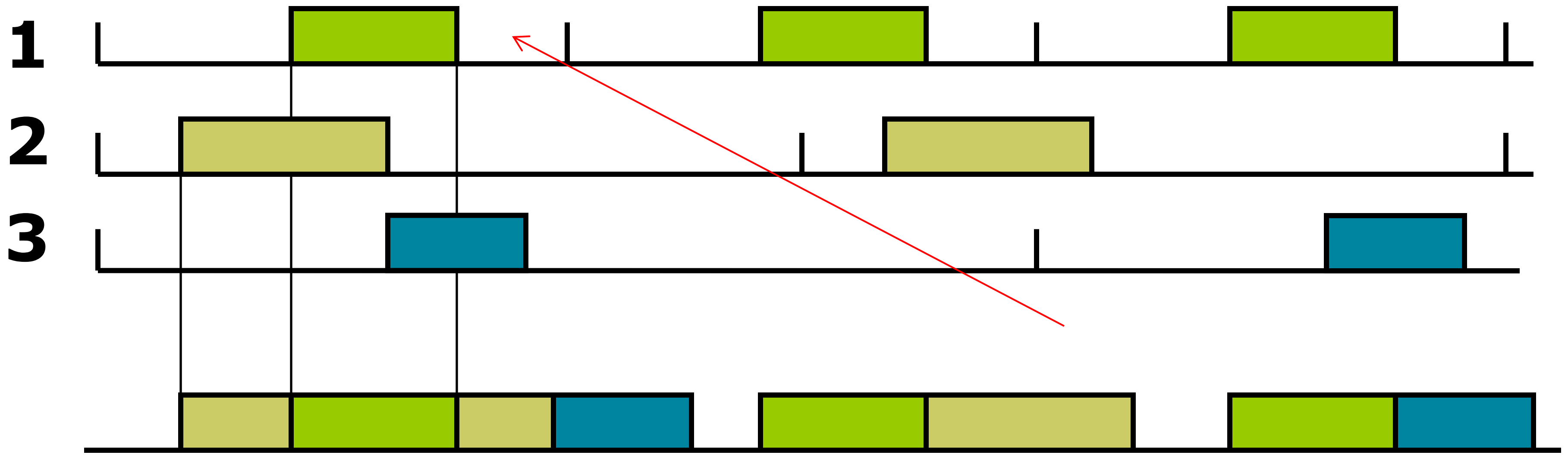


- Has processing time t, deadline d, period p;
- $0 \leq t \leq d \leq p$;
- **Rate** of periodic task is $1/p$;



Priority-based Preemptive Scheduling

- Always run the highest-priority runnable process



Priority-Based Preempting Scheduling

- Multiple processes at the same priority level?
- A few solutions
  ① Simply prohibit: Each process has unique priority
  ② Time-slice processes（处理）at the same priority
    - Extra context-switch overhead
    - No starvation dangers at that level
  ③ Processes at the same priority never preempt the other
    - More efficient
    - Still meets deadlines if possible

Real-time Scheduling

- Real-time processes have timing constraints
  - Expressed as deadlines or rate requirements
- Common RT scheduling policies
  - Rate monotonic
    - Just one scalar priority related to the periodicity of the job
    - Priority = 1/rate（短周期优先）
    - Static
  - Earliest deadline first (EDF)
    - Dynamic but more complex
    - Priority = deadline
- Both require admission control to provide guarantees

Rate-Monotonic Scheduling（ RMS ）

- Common way to assign priorities
- Result from Liu & Layland, 1973 (JACM)
- Simple to understand and implement:

Processes with shorter period given higher priority

- E.g.,

Period	Priority	
10	1	(highest)
12	2	
15	3	
20	4	(lowest)

Key RMS Result

- Rate-monotonic scheduling is optimal:

If there is fixed-priority schedule that meets all deadlines, then RMS will produce a feasible schedule

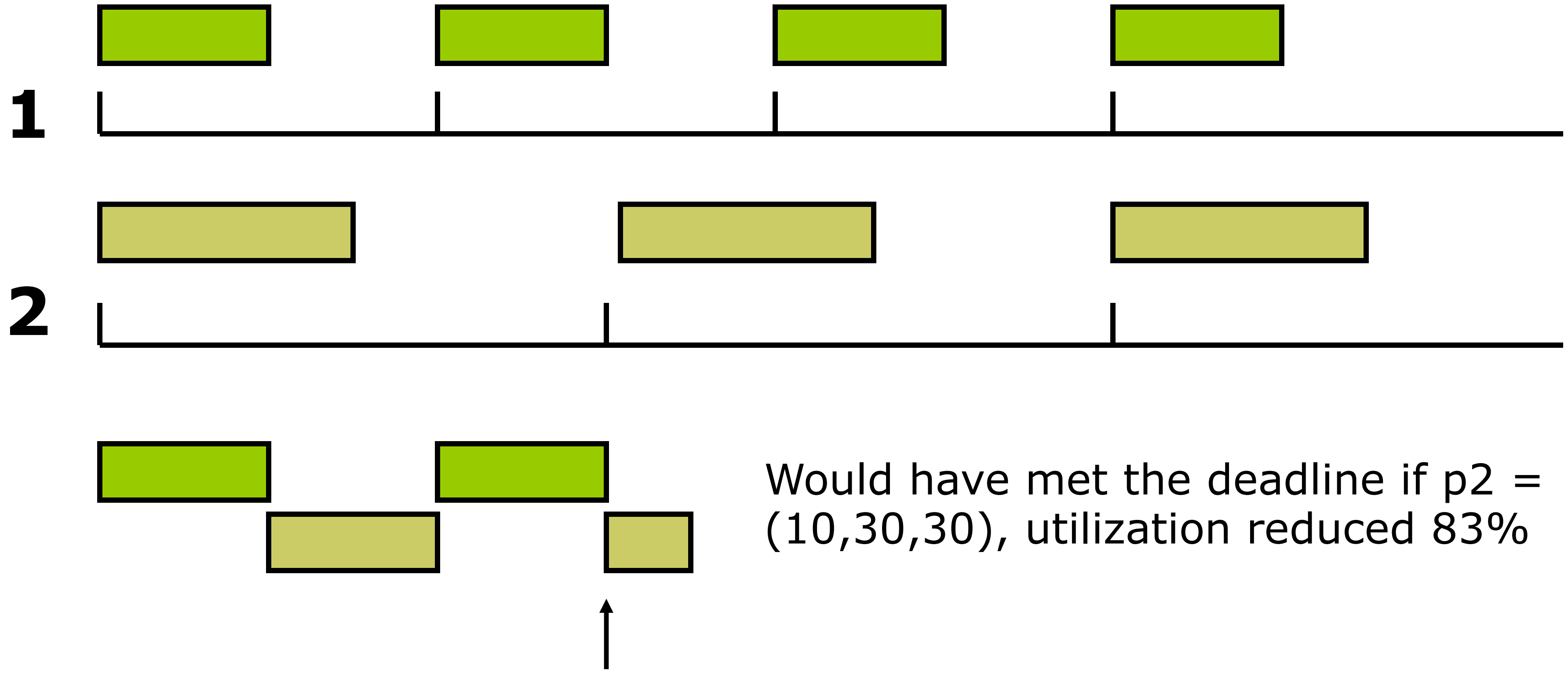
- Task sets do not always have a schedule
- Simple example: P1 = (10, 20, 20), P2 = (5, 9, 9)
  - Requires more than 100% processor utilization

$(10/20 + 5/9 > 1)$

triplet: (execution time, period, deadline)

RMS Missing a Deadline

- p1 = (10,20,20), p2 = (15,30,30) utilization is 100%



Would have met the deadline if p2 = (10,30,30), utilization reduced 83%

P2 misses first deadline

When Is There an RMS Schedule?

- Key metric is processor utilization: sum of compute time divided by period for each process:

$$U = \Sigma\, c_i \,/\, p_i$$

- No schedule can possibly exist if $U > 1$
  - No processor can be running 110% of the time
- Fundamental result:
  - RMS schedule always exists if $U < n\,(2^{1/n} - 1)$

When Is There an RMS Schedule?

- Also compute worst-case CPU utilization for scheduling N processes

$$CPU_{worst\text{-}case} = 2(2^{1/n} - 1)$$

- Only if $CPU_{actual} < CPU_{worst\text{-}case}$ is Rate-Monotonic Scheduling guaranteed to schedule the processes so that they meet their deadlines

(Proof in Liu and Layland,1973)

First Example

- Two processes

  P1, P2

- Scheduling parameters

  P1: c=20; p=50; d= 50

  P2: c=35; p=100; d=100

- Confirm that the tasks can be scheduled.

$$CPU_{actual} = \Sigma\, c_i \,/\, p_i = 20/50 + 35/100 = .75$$

$$CPU_{worst\text{-}case} = 2(2^{1/n} - 1) = 2\,(2^{1/2} - 1) = .828$$

- Rate-Monotonic Scheduling guaranteed to have them meet their deadlines

68

Other Example

- Three processes

  P1, P2, P3

- Scheduling parameters

  P1: c=20; p=50; d= 50

  P2: c=35; p=100; d=100

  P3: c=10; p=75; d=75

- Can the tasks be scheduled?

EDF Scheduling

- RMS assumes fixed priorities
- Can you do better with dynamically-chosen priorities?
- Earliest deadline first:

Processes with soonest deadline given highest priority

Earliest-Deadline-First (EDF) Scheduling

- Preemptive & dynamically assigns priorities according to deadline
  - Earlier deadline, higher priority
  - Later deadline, lower priority
- Highest priority process available gets CPU
- Example

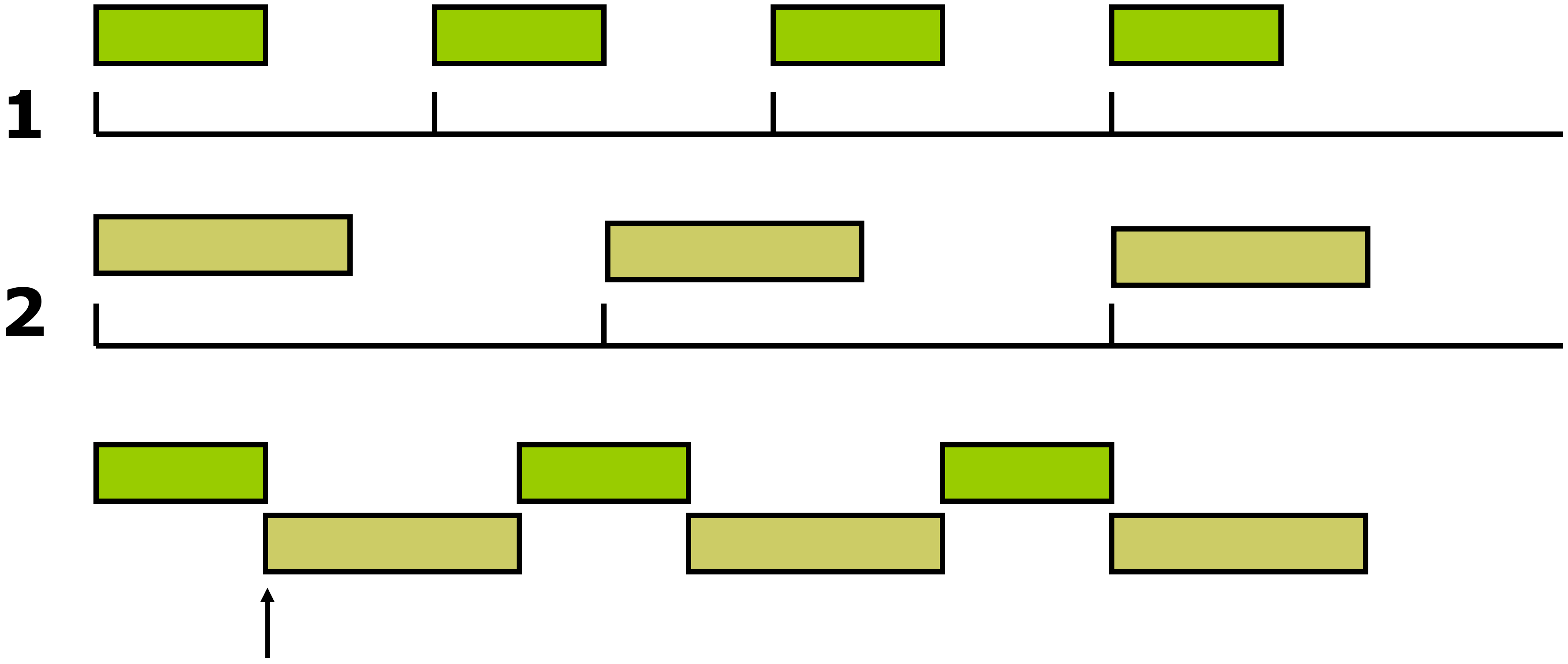
  P1: c=25; p=50; d=50

  P2: c=35; p=80; d=80

  Show a Gantt chart for first 145 time units

EDF Meeting a Deadline

- p1 = (10,20,20) p2 = (15,30,30) utilization is 100%



P2 takes priority because its
deadline is sooner

Key EDF Result

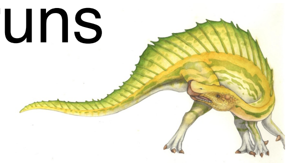
- Earliest deadline first scheduling is optimal:

  If a dynamic priority schedule exists, EDF will produce a feasible schedule

- Earliest deadline first scheduling is efficient:

  A dynamic priority schedule exists if and only if utilization is no greater than 100%

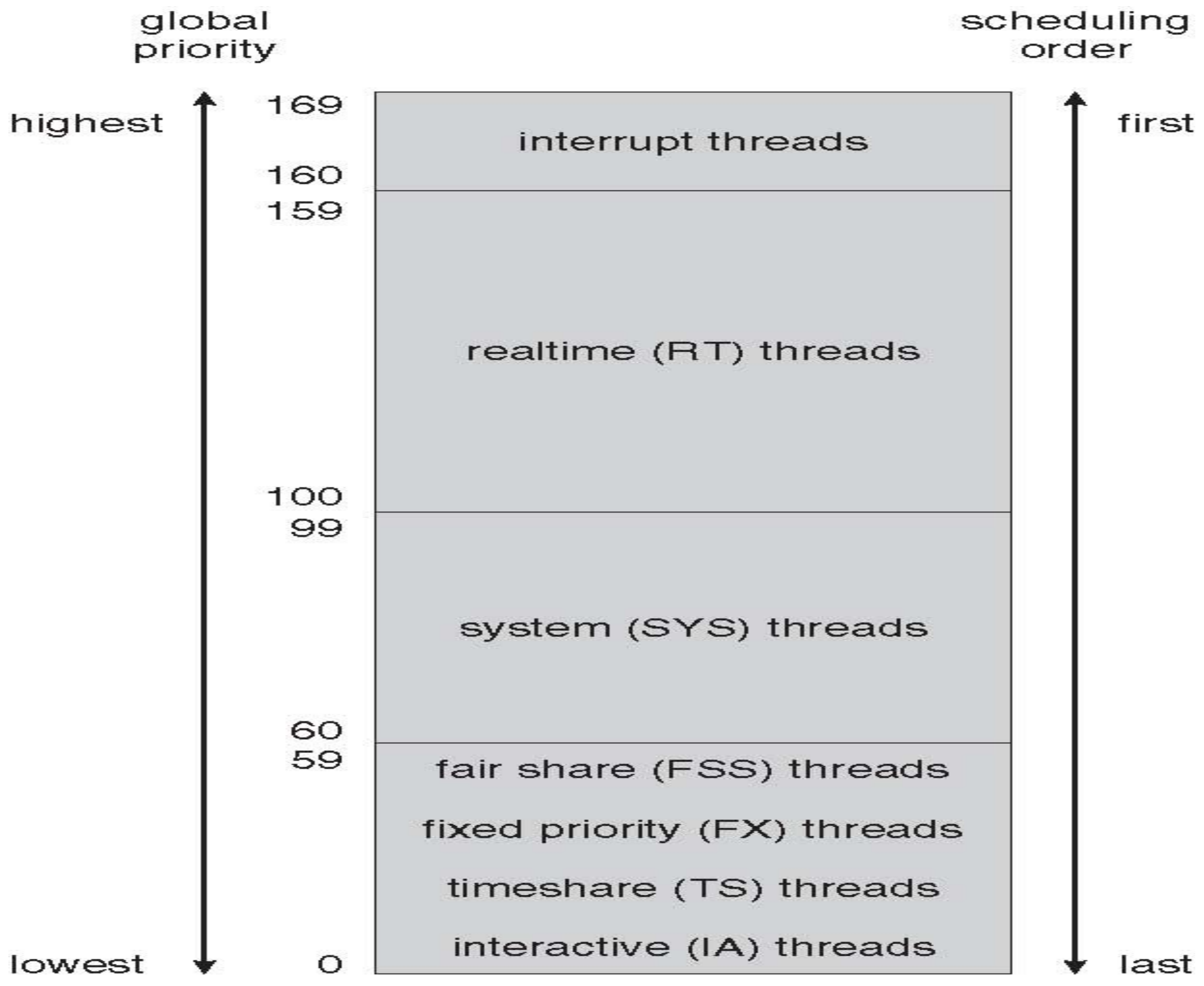
- EDF guarantees it at 100%
- EDF is complicated enough to have unacceptable overhead
- More complicated than RMA: harder to analyze
- Less predictable: can't guarantee which process runs when



Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris Scheduling



Solaris Dispatch Table for

New priority of a thread that has used its entire time quantum

Priority of a thread returning from sleeping

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

策略:因时间片用完的进程可以慢些;因睡眠而继续的可以加快.

Windows XP scheduling

- Priority-driven, preemptive scheduling system
- Thread runs for time amount of *quantum*
- Dispatcher routines triggered by the following events(调度时机):
  - Thread becomes ready for execution
  - Thread leaves running state (quantum expires, wait state)
  - Thread's priority changes

Windows XP scheduling

- Multiple threads may be ready to run, “Who gets to use the CPU?”
- From Windows API point of view:
  - Processes are given a priority class upon creation
    - Idle, Below normal, Normal, Above normal, High, Realtime
  - Threads have a relative priority within the class
    - Idle, Lowest, Below_Normal, Normal, Above_Normal, Highest, and Time_Critical

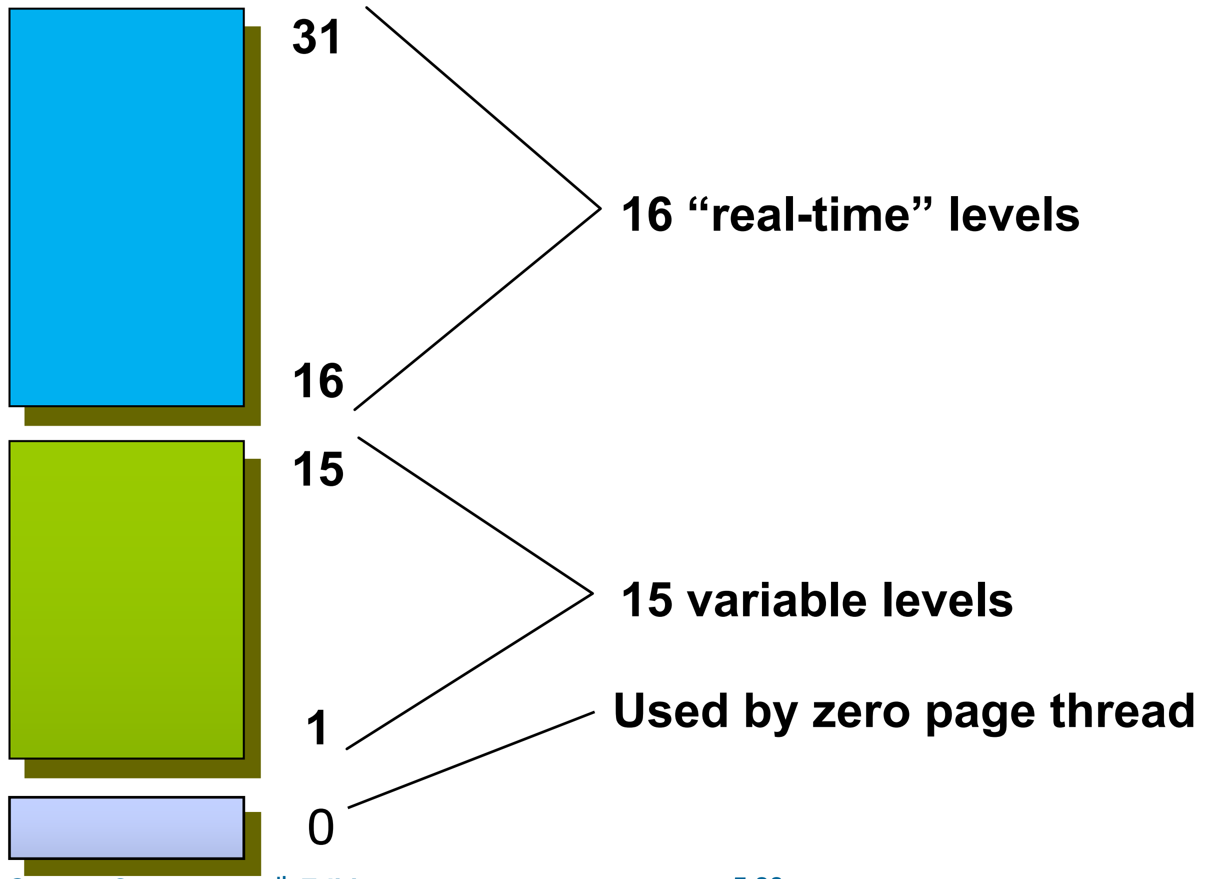
Windows XP Priorities

Processes Threads	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

- From the kernel's view:
  - Threads have priorities 0 through 31
  - Threads are scheduled, not processes
  - Process priority class is not used to make scheduling decisions

Kernel: Thread Priority Levels

- Threads within same priority are scheduled following the Round-Robin policy
- Non-Realtime Priorities are adjusted dynamically
- Real-time priorities (i.e.; > 15) are assigned statically to threads



Priority Adjustments for Windows XP Threads

- Five types:
  - I/O completion
  - Wait completion on events or semaphores
  - When threads in the foreground process complete a wait
  - When GUI threads wake up for windows input
  - For CPU starvation avoidance
- No automatic adjustments in “real-time” class (16 or above)
  - “Real time” here really means “system won’t change the relative priorities of your real-time threads”
  - Hence, scheduling is predictable with respect to other “real-time” threads (but not for absolute latency)

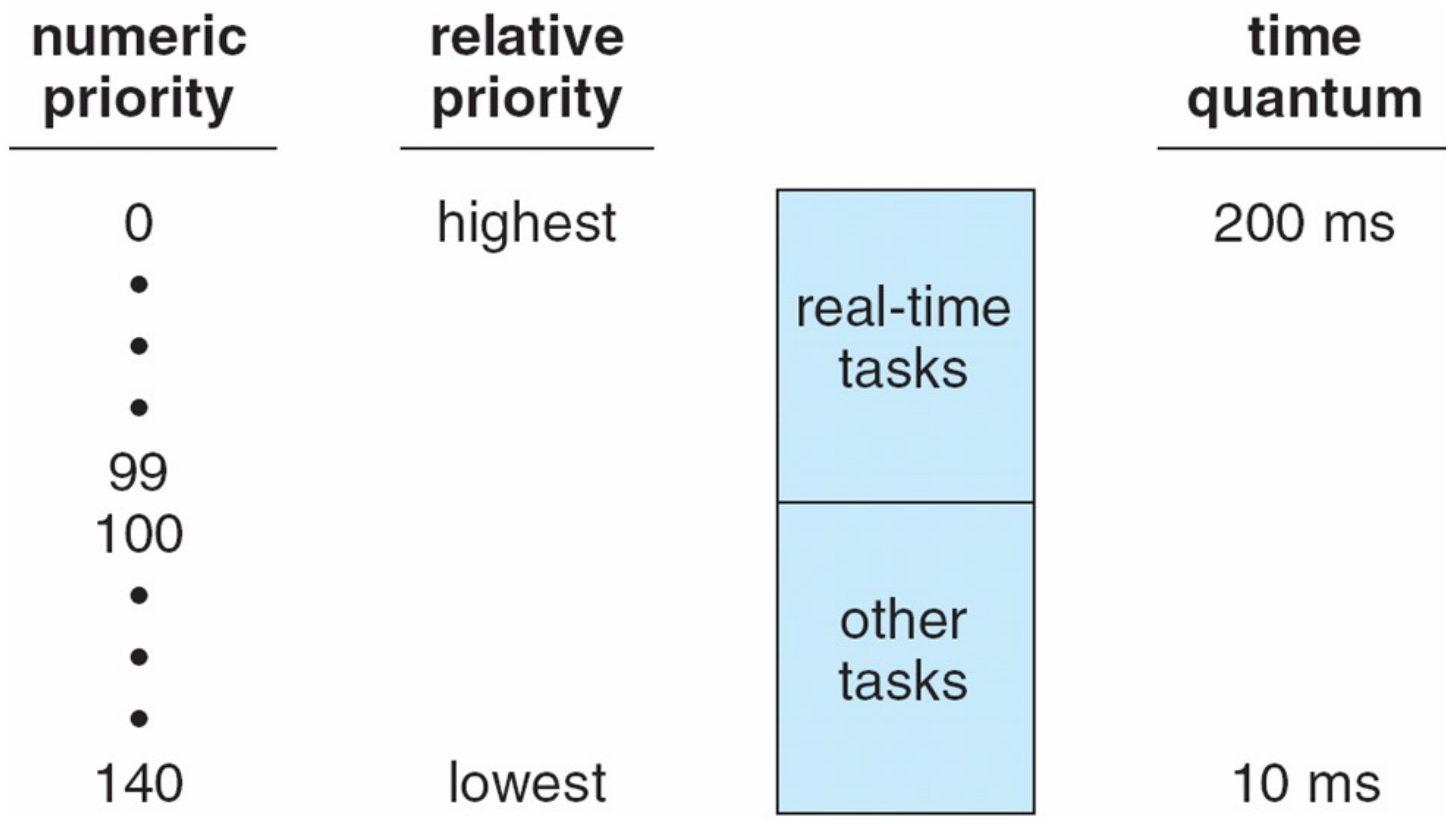
Linux Scheduling

- The Linux scheduler is a preemptive, priority-based algorithm with two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99
  - **nice** value from 100 to 140

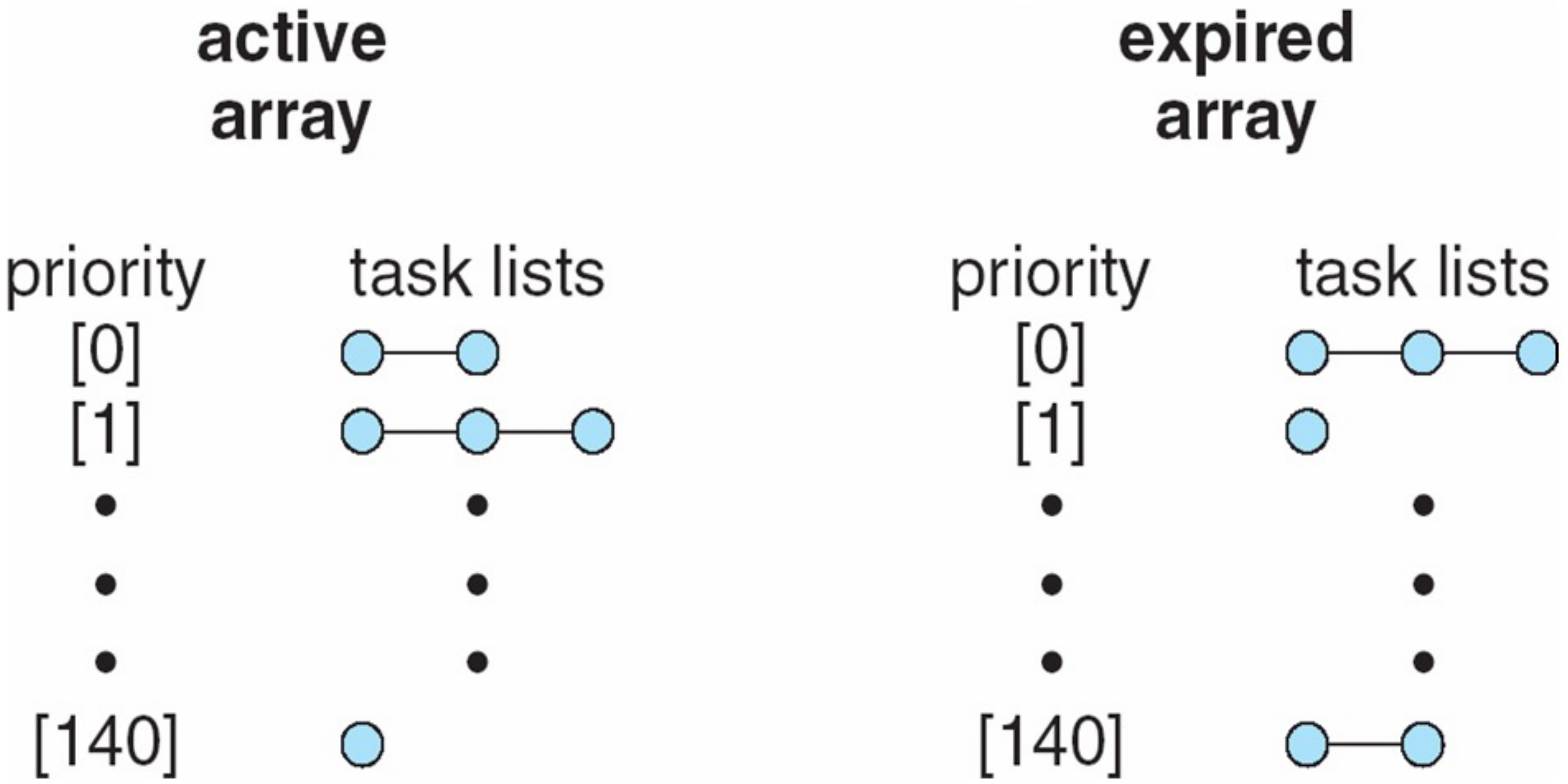
  p_pri=min{127，(p_cpu/16+PUSER+p_nice)}
- These two ranges map into a global priority scheme whereby numerically lower values indicate higher priority

（数值低表示优先级高）

Priorities and Time-slice length



List of Tasks Indexed According to Priorities



Linux Scheduling

- Linux supports SMP, each processor maintains its own runqueue and schedules itself independently.
- Each runqueue contains two priority arrays :
  - Active: contains all tasks with time remaining in their time slices
  - Expired: contains all expired tasks
  - Each of these arrays contains a list of tasks indexed according to priority
- The scheduler chooses the task with the highest priority from the active array for execution on the CPU.
- When all tasks have exhausted their time slices, the priority arrays are exchanged.

Why two priority arrays?

Algorithm Evaluation

- How do we select a CPU scheduling algorithm for a particular system?
- The first problem is defining the criteria to be used in selecting an algorithm. According to system goal :
  - CPU utilization
  - Throughput
  - Average turnaround time
  - Responstime
- Then select an algorithm
- And then evaluate the algorithm selected

Algorithm Evaluation

- Deterministic modeling
  - takes a particular predetermined workload and defines the performance of each algorithm for that workload(比如Benchmark)
- Queueing models
- Simulations
- Implementation

Algorithm Evaluation

- Queueing models
  - The computer system is described as a network of servers. Each server has a queue of waiting processes.
  - The CPU is a server with its ready queue, and I/O system with device queues.
  - Knowing arriving rates and service rates, we can compute utilization, average queue length, average wait time, and so on.
  - E.g. let *w* be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue, let *n* be the average queue length, then

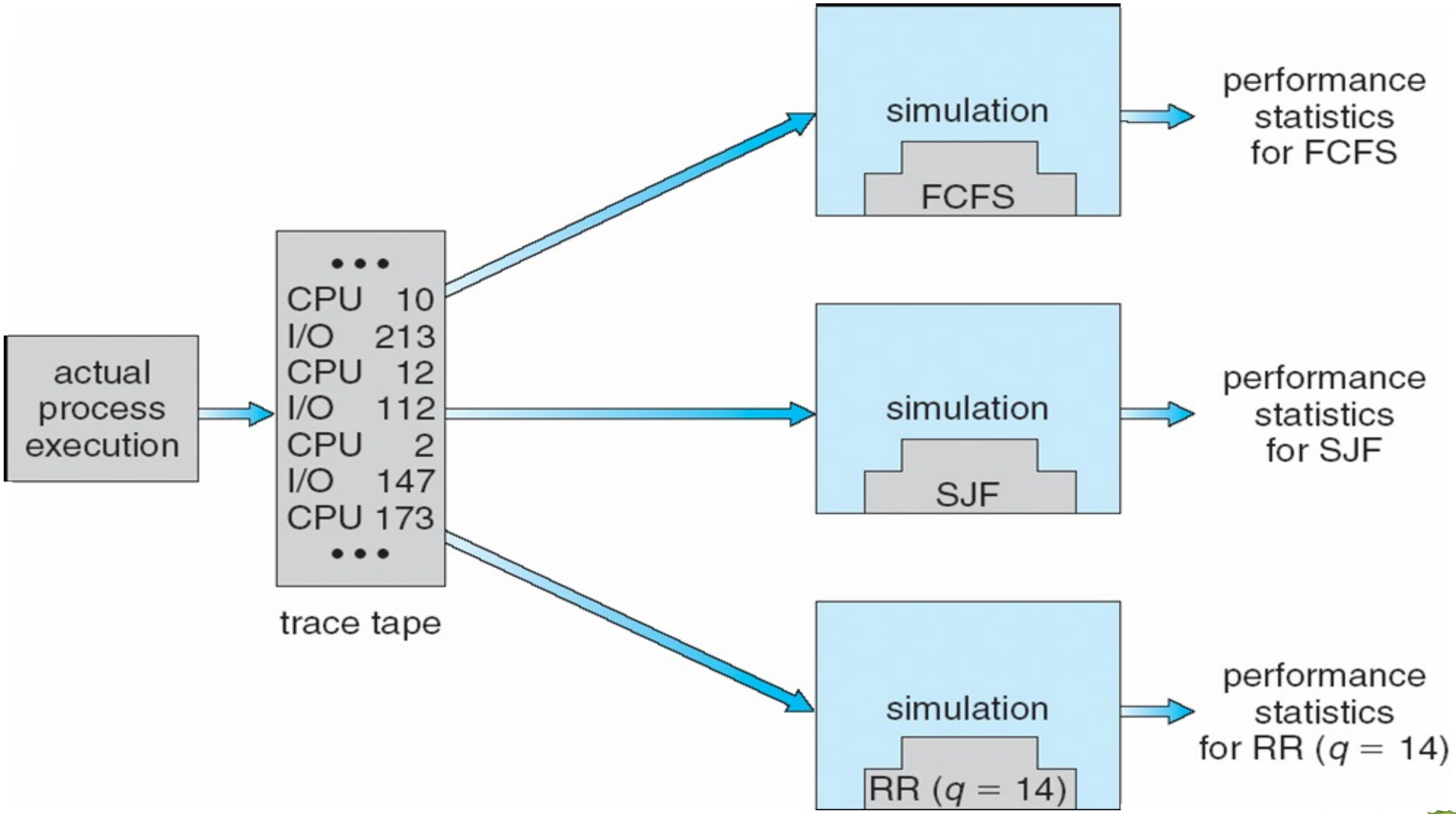
$$n \;\square\; \lambda \times W$$

Algorithm Evaluation

Simulations

- Programming a model of the computer system
    - Software data structures represent the major components of the system.
    - The simulator has a variable representing a clock
    - As the variable's value is increased, the simulator modifies the system state to reflect the activity of the devices, the processes , and the scheduler.
  - As the simulator executes, statistics that indicate algorithm performance are gathered and printed.

Evaluation of CPU schedulers by Simulation



Algorithm Evaluation

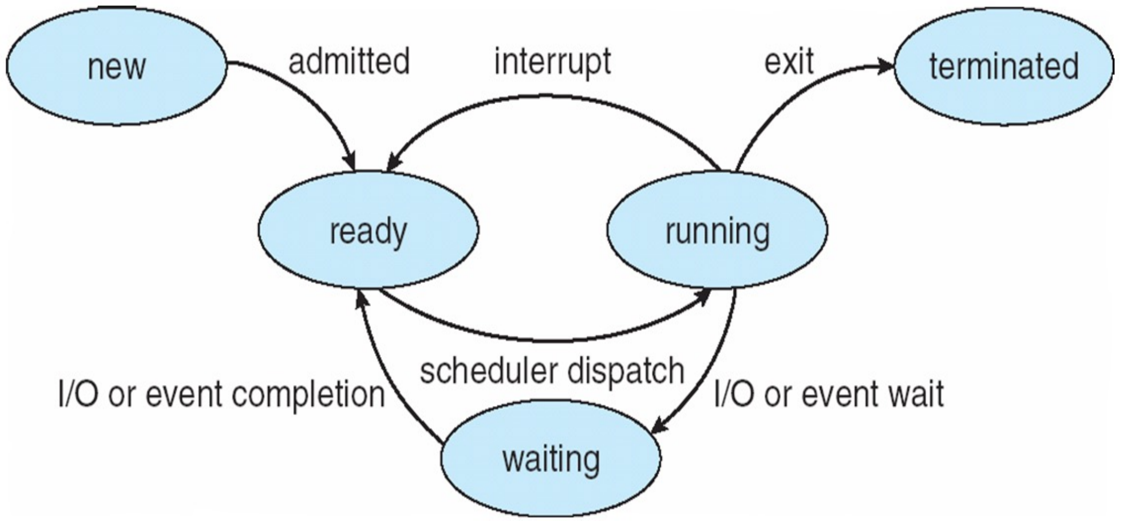
- Implementation
  - The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the OS, and see how it works.

Exercise

- 5.1
- 5.4
- 5.5

课堂小测验：

根据进程状态图分析并回答分时系统与实时系统实现上的区别：



End of Chapter 5