操作系统
Operating Systems

刘川意 教授

哈尔滨工业大学（深圳）

2025年9月

Module 3：调度

1. Mechanism: Limited Direct Execution

1.1 Direct Execution的局限性 – 基础

- 问题1：Restricted Operation – 基础
- 问题2：Switching Between Process – 基础

1.2 Limited Direct Execution – 基础

2. Policy: Scheduling Algorithms

2.1 Introduction – 基础

2.2 The Multi-Level Feedback Queue – 基础

2.3 Proportional Share – 进阶

OS如何可控的实现CPU虚拟化

- OS内核实现物理CPU在进程间共享的基本思路：time sharing.
- 要解决2个核心问题
  - **Performance**: How can we implement virtualization without adding excessive overhead to the system? 即引入的额外性能尽可能小
  - **Control**: How can we run processes efficiently while retaining control over the CPU? 即不要“跑飞”了，不要被“劫持”了

用户program直接运行在物理CPU上

- Just run the program directly on the CPU.

OS	Program
1. Create entry for process list 2. Allocate memory for program 3. Load program into memory 4. Set up stack with `argc` / `argv` 5. Clear registers 6. Execute call `main()`	
	7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

Without *limits* on running programs,
the OS wouldn't be in control of anything and
thus would be "just a library"
让program无限制的运行在物理CPU上，则OS沦为library

Problem 1: Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as …如果一个进程希望执行某种受限操作，例如
  - Issuing an I/O request to a disk 向磁盘发出 I/O 请求
  - Gaining access to more system resources such as CPU or memory 获得对更多系统资源（例如 CPU 或内存）的访问权限

- **Solution**: Using 保护模式（protected control transfer）
  - User mode: Applications do not have full access to hardware resources. 应用程序没有对硬件资源的完全访问权限
  - Kernel mode: The OS has access to the full resources of the machine 操作系统可以访问机器的全部资源

System Call

- Allow the kernel to carefully expose certain key pieces of functionality to user program, such as …
  允许内核谨慎地向用户程序公开某些关键功能，例如
  - Accessing the file system 访问文件系统
  - Creating and destroying processes 创建和销毁进程
  - Communicating with other processes 与其他进程通信
  - Allocating more memory 分配更多内存

- **Trap** instruction
  - Jump into the kernel 进入内核态
  - Raise the privilege level to kernel mode 将权限级别提升到内核模式

- **Return-from-trap** instruction
  - Return into the calling user program 返回到调用用户程序
  - Reduce the privilege level back to user mode
    将权限级别降低回用户模式

Limited Direction Execution Protocol

OS @ boot (kernel mode)	Hardware	
initialize trap table 初始化trap表		
	remember address of syscall handler 记住系统调用处理程序的地址	
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**

- Create entry for process list 为进程列表创建条目
- Allocate memory for program 为程序分配内存
- Load program into memory 将程序加载到内存中
- Setup user stack with argv 使用 argv 设置用户堆栈
- Fill kernel stack with reg/PC 使用 reg/PC 填充内核堆栈
- **return-from-trap**
  - restore regs from kernel stack 从内核堆栈恢复寄存器
  - move to user mode 移动到用户模式
  - jump to main 跳转到主程序
    - Run main()
    - ...
    - Call system
    - **trap** into OS

Limited Direction Execution Protocol (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	➢ save regs to kernel stack 将寄存器保存到内核堆栈 ➢ move to kernel mode 移动到内核态 ➢ jump to trap handler 跳转到陷阱处理程序	
➢ Handle trap ➢ Do work of syscall ➢ **return-from-trap**		
	➢ restore regs from kernel stack 从内核堆栈恢复寄存器 ➢ move to user mode 移动到用户态 ➢ jump to PC after trap 陷阱后跳转到PC	
		➢ … ➢ return from main ➢ trap (via `exit()`)
➢ Free memory of process ➢ Remove from process list		

Problem 2: Switching Between Processes

- How can the OS regain control of the CPU so that it can switch between *processes*? 操作系统如何重新获得对 CPU 的控制权，以便在进程之间切换
  - A cooperative Approach: **Wait for system calls**

    一种合作的方法：等待系统调用
  - A Non-Cooperative Approach: **The OS takes control**

    一种非合作方法：操作系统控制

A cooperative Approach: Wait for system calls

- Processes periodically give up the CPU by making **system calls** such as `yield`. 进程周期性地通过系统调用（例如yield）放弃CPU
  - The OS decides to run some other task. 操作系统决定运行一些其他任务。
  - Application also transfer control to the OS when they do something illegal. 当应用程序做一些非法的事情时，它们也会将控制权转移给操作系统。
    - Divide by zero 除0
    - Try to access memory that it shouldn't be able to access 非法访问
  - Example: Early versions of the Macintosh OS, The old Xerox Alto system
    示例：Macintosh OS 的早期版本，旧的 Xerox Alto 系统

A process gets stuck in an infinite loop.
→ Reboot the machine

A Non-Cooperative Approach: OS Takes Control

- **A timer interrupt**
  - During the boot sequence, the OS start the timer.
    在启动序列中，操作系统启动计时器
  - The timer raise an interrupt every so many milliseconds.
    计时器每隔这么多毫秒引发一次中断
  - When the interrupt is raised :
    - The currently running process is halted. 当前运行的进程被暂停
    - Save enough of the state of the program. 保存足够的程序状态
    - A pre-configured interrupt handler in the OS runs.
      操作系统中预配置的中断处理程序运行

A timer interrupt gives OS the ability to run again on a CPU.

Saving and Restoring Context

- Scheduler makes a decision: 调度器做出决定
  - Whether to continue running the **current process**, or switch to a **different one**. 是继续运行当前进程，还是切换到其他进程
  - If the decision is made to switch, the OS executes context switch. 如果决定切换，则操作系统执行上下文切换。

Context Switch

- A low-level piece of assembly code 一段低级的汇编代码
  - **Save a few register values** for the current process onto its kernel stack 将当前进程的一些寄存器值保存到其内核堆栈中
    - General purpose registers 通用寄存器
    - PC
    - kernel stack pointer 内核堆栈指针
  - **Restore a few** for the soon-to-be-executing process from its kernel stack 从内核堆栈中为即将执行的进程恢复一些
  - **Switch to the kernel stack** for the soon-to-be-executing process 切换到即将执行的进程的内核堆栈

Module 3：调度

1. Mechanism: Limited Direct Execution

1.1 Direct Execution的局限性 – 基础

- 问题1：Restricted Operation – 基础
- 问题2：Switching Between Process – 基础

1.2 Limited Direct Execution – 基础

2. Policy: Scheduling Algorithms

2.1 Introduction – 基础

2.2 The Multi-Level Feedback Queue – 基础

2.3 Proportional Share – 进阶

Limited Direction Execution Protocol (Timer interrupt)

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of ...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	

Limited Direction Execution Protocol (Timer interrupt) (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
Handle the trap		
Call switch() routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) from k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		…

The xv6 Context Switch Code


```
1  # void swtch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl swtch
6  swtch:
7          # Save old registers
8          movl 4(%esp), %eax          # put old ptr into eax
9          popl 0(%eax)                # save the old IP
10         movl %esp, 4(%eax)          # and stack
11         movl %ebx, 8(%eax)          # and other registers
12         movl %ecx, 12(%eax)
13         movl %edx, 16(%eax)
14         movl %esi, 20(%eax)
15         movl %edi, 24(%eax)
16         movl %ebp, 28(%eax)
17
18         # Load new registers
19         movl 4(%esp), %eax          # put new ptr into eax
20         movl 28(%eax), %ebp         # restore other registers
21         movl 24(%eax), %edi
22         movl 20(%eax), %esi
23         movl 16(%eax), %edx
24         movl 12(%eax), %ecx
25         movl 8(%eax), %ebx
26         movl 4(%eax), %esp          # stack is switched here
27         pushl 0(%eax)               # return addr put in place
28         ret                         # finally return into new ctxt
```

Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
  如果在中断或陷阱处理期间发生另一个中断，会发生什么
- OS handles these situations: 操作系统处理这些情况
  - **Disable interrupts** during interrupt processing 在中断处理期间禁用中断
  - Use a number of sophisticate **locking** schemes to protect concurrent access to internal data structures.
    使用许多复杂的锁定方案来保护对内部数据结构的并发访问

Module 3：调度

1. Mechanism: Limited Direct Execution

2. Policy: Scheduling Algorithms

2.1 Introduction – 基础

- 概述 – 基础
- 经典调度算法 – 基础
  - 先进先出 FIFO – 基础
  - 最短任务优先 SJF – 基础
  - 轮转调度 RR – 基础

2.2 The Multi-Level Feedback Queue – 基础

2.3 Proportional Share – 进阶

Scheduling: Introduction

- Workload assumptions(工作负载初始化假设，后续算法逐步放宽这些假设):
  1. Each job runs for the **same amount of time.** 每个作业运行相同的时间
  2. All jobs **arrive** at the same time. 所有工作同时到达
  3. All jobs only use the **CPU** (i.e., they perform no I/O). 所有作业仅使用CPU（例如，它们不执行 I/O）
  4. The **run-time** of each job is known. 每个作业的运行时间是已知的

Scheduling Metrics 调度指标

- Performance metric: Turnaround time（周转时间）
  - The time at which **the job completes** minus the time at which **the job arrived** in the system. （任务完成时间减去任务到达系统的时间）

$$T_{turnaround} = T_{completion} - T_{arrival}$$

- Another metric is fairness（公平）.
  - Performance and fairness are often at odds in scheduling.
    性能和公平性在调度中经常不一致
- The time from **when the job arrives** to the **first time it is scheduled**.

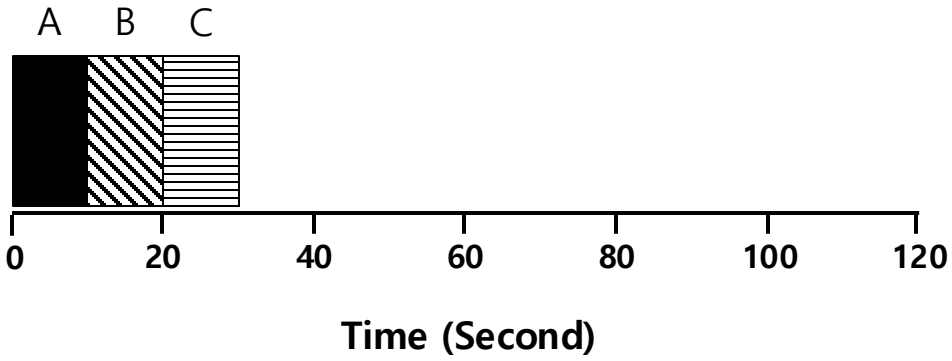
  响应时间是指：从任务到达系统到首次运行（首次被调度）的时间

$$T_{response} = T_{firstrun} - T_{arrival}$$

  - STCF and related disciplines are not particularly good for response time. STCF最短完成时间优先策略，用响应时间来评估不是好方法

First In, First Out 先进先出 (FIFO)

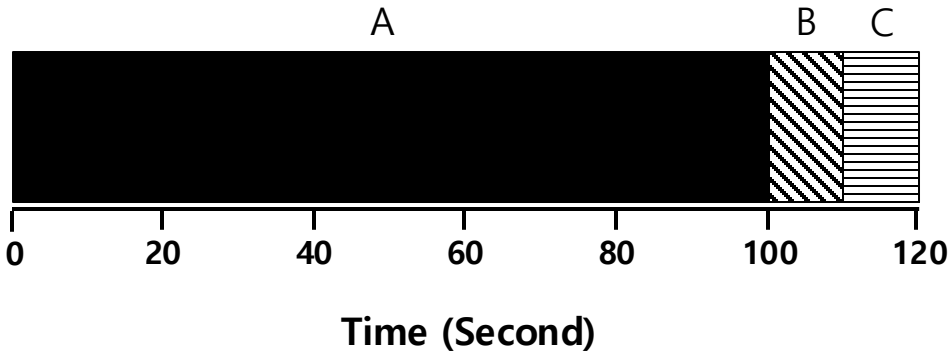
- First Come, First Served (FCFS)
  - Very simple and easy to implement
- Example:
  - A arrived just before B which arrived just before C.
  - Each job runs for 10 seconds.



$$Average\ turnaround\ time = \frac{10 + 20 + 30}{3} = 20\ sec$$

Why FIFO is not that great? – Convoy effect（护航效应）

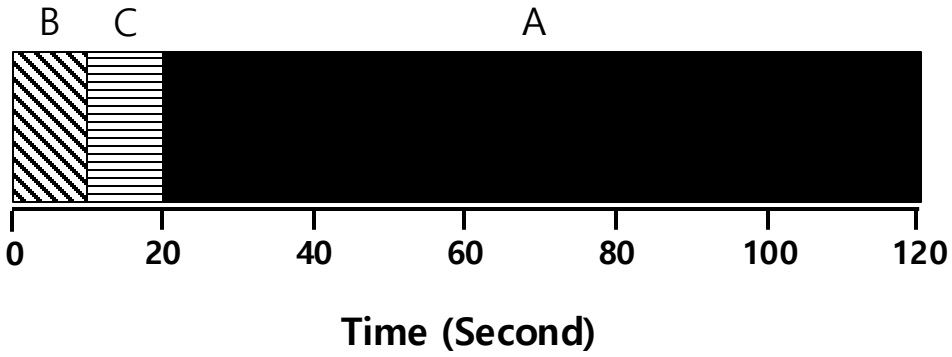
- Let's relax assumption 1: Each job **no longer** runs for the same amount of time. 让我们放宽假设 1：每个作业不再运行相同的时间
- Example:
  - A arrived just before B which arrived just before C.
  - A runs for 100 seconds, B and C run for 10 each.



$$Average\ turnaround\ time = \frac{100 + 110 + 120}{3} = 110\ sec$$

Shortest Job First (最短任务优先 SJF)

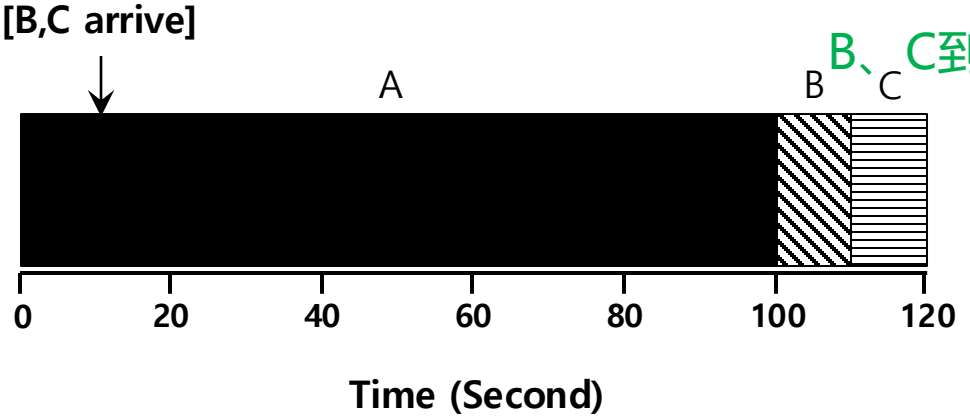
- Run the shortest job first, then the next shortest, and so on
  首先运行最短的作业，然后运行下一个最短的作业，依此类推
  - Non-preemptive（非抢占式）scheduler
- Example:
  - A arrived just before B which arrived just before C.
  - A runs for 100 seconds, B and C run for 10 each.



$$Average\ turnaround\ time = \frac{10 + 20 + 120}{3} = 50\ sec$$

SJF with Late Arrivals from B and C

- Let's relax assumption 2: Jobs can arrive at any time. 让我们放宽假设2：工作可以随时到达
- Example:
  - A arrives at t=0 and needs to run for 100 seconds. A到达t=0，需要运行100秒
  - B and C arrive at t=10 and each need to run for 10 seconds



B、C到达t=10，各跑10秒

$$Average\ turnaround\ time = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33\ sec$$

Shortest Time-to-Completion First

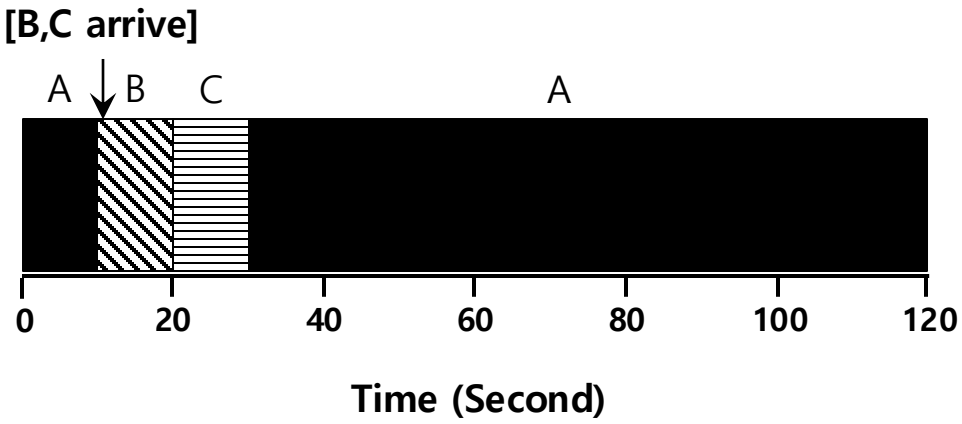
—— 最短完成时间优先 STCF

- Add preemption（抢占）to SJF
  - Also knows as Preemptive Shortest Job First (PSJF)
    也称为抢占式最短任务优先
- A new job enters the system:
  - Determine of the remaining jobs and new job 确定剩余工作和新工作
  - Schedule the job which has the lest time left 安排剩余时间最少的作业

Shortest Time-to-Completion First

—— 最短完成时间优先 STCF

- Example:
  - A arrives at t=0 and needs to run for 100 seconds.
  - B and C arrive at t=10 and each need to run for 10 seconds



$$Average\ turnaround\ time = \frac{(120-0)+(20-10)+(30-10)}{3} = 50\ sec$$

Shortest Job First (最短任务优先 SJF)

- 【例题】现在有三个同时到达的作业J1、J2和J3，它们的执行时间分别是T1、T2和T3，而且T1<T2<T3。系统按单道方式运行且采用短作业优先调度算法，则平均周转时间是（ ）。

  A. $T_1+T_2+T_3$

  B. $(3T_1+2T_2+T_3)/3$

  C. $(T_1+T_2+T_3)/3$

  D. $(T_1+2T_2+3T_3)/3$

答案：B

系统采用短作业优先调度算法，作业的执行顺序为J1、J2和J3 ，它们的周转时间分别为T_1，T_1+T_2和$T_1+T_2+T_3$。

所以平均周转时间为$(3T_1+2T_2+T_3)/3$。

Shortest Job First (最短任务优先 SJF)

- 【例题】假设4个任务到达系统的时刻和运行时间见下表系统在t=2时开始作业调度。若分别采用先来先服务和最短任务优先调度算法，则选中的任务分别是( )

A、J2、J3

B、J1、J4

C、J2、J4

D、J1、J3

作业	到达时间t	运行时间
J_1	0	3
J_2	1	3
J_3	1	2
J_4	3	1

答案： D

解析：先来先服务调度算法时作业来得越早，优先级越高，因此会选择J1。短作业优先调度算法是作业运行时间越短，优先级越高，因此会选择J3。

New scheduling metric: Response time

——新的调度指标：响应时间

- The time from **when the job arrives** to the **first time it is scheduled**.

  响应时间是指：从任务到达系统到首次运行（首次被调度）的时间

$$T_{response} = T_{firstrun} - T_{arrival}$$

- STCF and related disciplines are not particularly good for response time.

  STCF 和相关策略对响应时间不是特别好

How can we build a scheduler that is sensitive to response time?

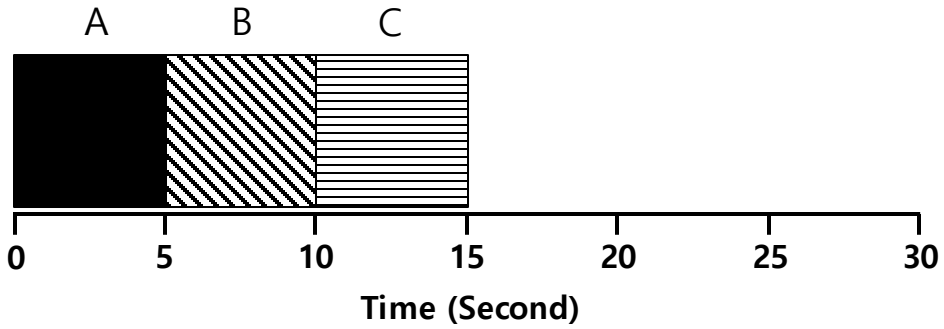
Round Robin (RR) Scheduling

——轮转调度

- Time slicing Scheduling
  - Run a job for a time slice(时间片) and then switch to the next job in the **run queue** until the jobs are finished.
    - Time slice is sometimes called a scheduling quantum(调度量子).
  - It repeatedly does so until the jobs are finished.
  - The length of a time slice must be *a multiple of* the timer-interrupt period (时钟**中断周期**). 时间片的长度必须是定时器中断周期的倍数

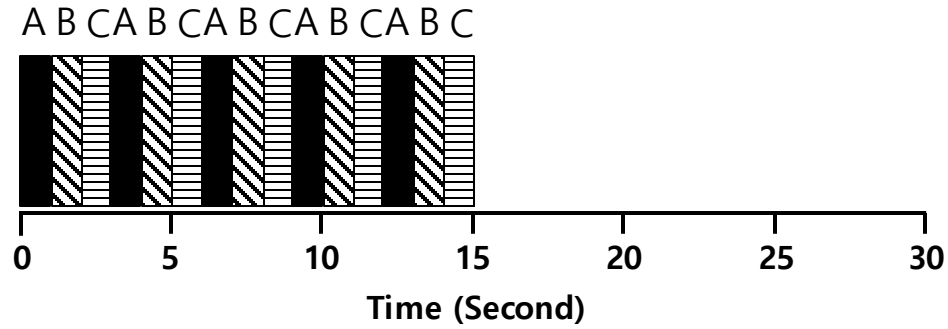
RR Scheduling Example

- A, B and C arrive at the same time.
- They each wish to run for 5 seconds.



SJF (Bad for Response Time)

$$T_{average\ response} = \frac{0 + 5 + 10}{3} = 5sec$$



RR with a time-slice of 1sec (Good for Response Time)

$$T_{average\ response} = \frac{0 + 1 + 2}{3} = 1sec$$

Round Robin (RR) Scheduling

——轮转调度

- Time slicing Scheduling
  - Run a job for a time slice(时间片) and then switch to the next job in the **run queue** until the jobs are finished.
    将一个作业运行一个时间片（时间片），然后切换到运行队列中的下一个作业，直到作业完成
    - Time slice is sometimes called a scheduling quantum(调度量子).
  - It repeatedly does so until the jobs are finished.
    它反复这样做，直到作业完成
  - The length of a time slice must be *a multiple of* the timer-interrupt period (时钟中断周期). 时间片的长度必须是定时器中断周期的倍数

RR is fair, but performs poorly on metrics such as turnaround time
RR 是公平的，但在指标上表现不佳
比如周转时间

Round Robin (RR) Scheduling

——轮转调度

【例题】下列有关时间片的进程调度的描述中，错误的是（ ）

A. 时间片越短，进程切换的次数越多，系统开销也越大

B. 当前进程的时间片用完后，该进程状态由执行态变为阻塞态

C. 时钟中断发生后，系统会修改当前的进程在时间片内的剩余时间

D. 影响时间片大小的主要因素包括响应时间、系统开销和进程数量等

答案：B

解析：进程切换带来系统开销，切换次数越多，系统开销越大，选项A正确。当前进程的时间片用完后，其状态由执行态变为就绪态，选项B错误。时钟中断是系统中特定的周期性时钟节拍，操作系统通过它来确定时间间隔，实现时间的延时跟任务的超时，选项C正确。现代操作系统为了保证性能最优，通常根据响应时间、系统开销、进程数目、进程运行时间、进程切换开销等因素确定时间片大小，选项D正确。

Round Robin (RR) Scheduling

——轮转调度

【例题】假设某系统使用时间片轮转调度算法进行CPU调度，时间片大小为5ms，系统共有10个进程，初始时均处于就绪队列，执行结束前仅处于执行态或就绪态。若队尾的进程P所需CPU时间最短，时间为25ms，在不考虑系统开销的情况下，则进程P的周转时间为

A. 200ms　　B. 205ms

C. 250ms　　D. 295ms

答案：C

解析：进程P所需CPU时间为25ms，因为时间片为5ms，进程P需要5轮时间片才能完成执行。

每轮中，10个进程均需执行一个时间片，总时间为：

10×5ms=50ms

5轮共250ms

The length of the time slice is critical.

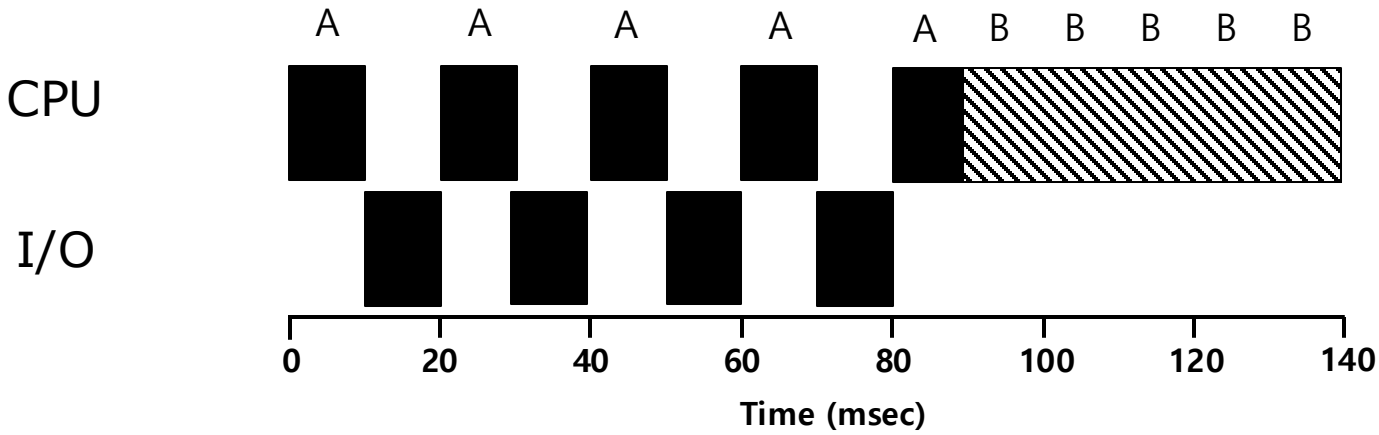
- The shorter time slice
  - Better response time
  - The cost of context switching will dominate overall performance.
    上下文切换的成本将主导整体性能
- The longer time slice
  - Amortize the cost of context switching 分摊上下文切换的成本
  - Worse response time

Deciding on the length of the time slice presents a trade-off to a system designer

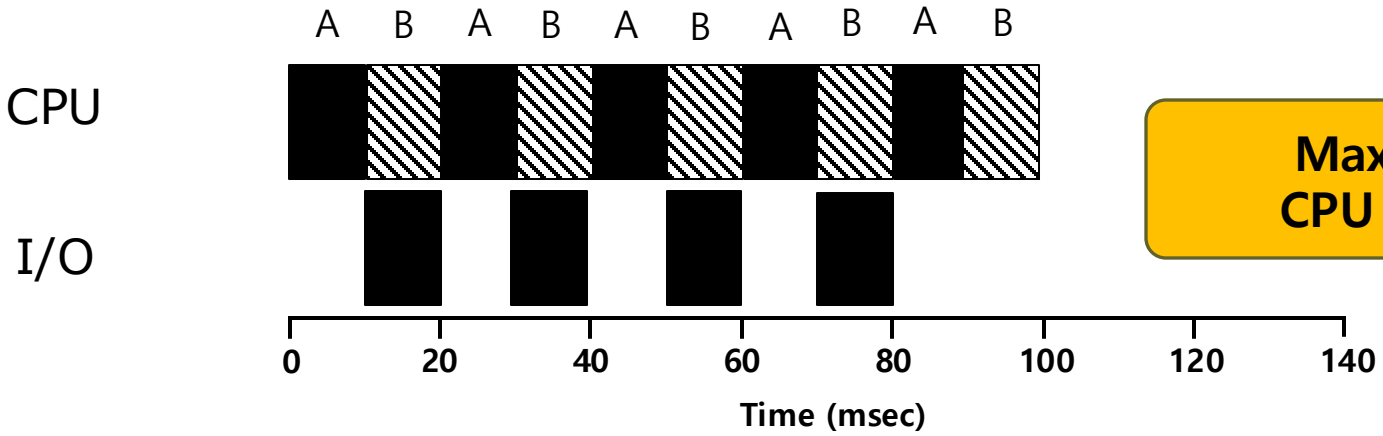
Incorporating I/O

- Let's relax assumption 3: **All programs can perform I/O**
- Example:
  - A and B need 50ms of CPU time each.
  - A runs for 10ms and then issues an I/O request
    - I/Os each take 10ms
  - B simply uses the CPU for 50ms and performs no I/O
  - 为显示I/O对调度的影响，假设使用最简化的调度算法：The scheduler runs A first, then B after

Incorporating I/O (Cont.)



Poor Use of Resources



Maximize the CPU utilization

Overlap Allows Better Use of Resources

Incorporating I/O (Cont.)

- When a job initiates an I/O request.
  - The job is blocked waiting for I/O completion. 作业被阻塞等待 I/O 完成
  - The scheduler should schedule another job on the CPU.
    调度程序应该在 CPU 上调度另一个作业
- When the I/O completes
  - An interrupt is raised.
  - The OS moves the process from blocked back to the ready state.
    操作系统将进程从阻塞状态移回就绪状态

Incorporating I/O (Cont.)

- 【例题】一个多道批处理系统中仅有P1和P2两个作业，P2比P1晚5s到达，它们的计算和I/O操作顺序如下：

P1：计算60ms，I/O：80ms，计算20ms

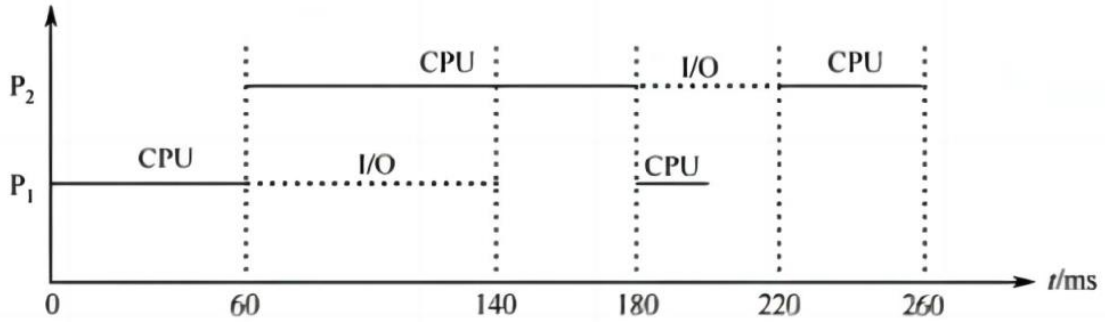
P2：计算120ms，I/O：40ms，计算40ms

若不考虑调度和切换时间，则完成两个作业需要的时间最少是（ ）。

A. 240ms　　B. 260ms　　C. 340ms　　D. 360ms

答案：B

由于P2比P1晚5ms到达，P1先占用CPU，作业运行的甘特图如下。



Module 3：调度

1. Mechanism: Limited Direct Execution

2. Policy: Scheduling Algorithms

2.1 Introduction – 基础

- 概述 – 基础
- 经典调度算法 – 基础

2.2 The Multi-Level Feedback Queue – 基础

2.3 Proportional Share – 进阶

Multi-Level Feedback Queue (MLFQ)

——多级反馈队列

- A Scheduler that learns from the past to predict the future.
- Objective (目标):
  - Optimize **turnaround time** → Run shorter jobs first
  - Minimize **response time** without *a* 先验知识 *(priori knowledge) of* 任务运行时间 *(job length)*

MLFQ: Basic Rules

- MLFQ has a number of distinct **queues**.
- Each queues is assigned a different priority level.

  每一个队列赋予不同的优先级

- A job that is ready to run is on a single queue.
  - A job **on a higher queue** is chosen to run. 不同队列，高级队列先调度
  - Use round-robin scheduling among jobs in the same queue 同队列，采用R-R调度

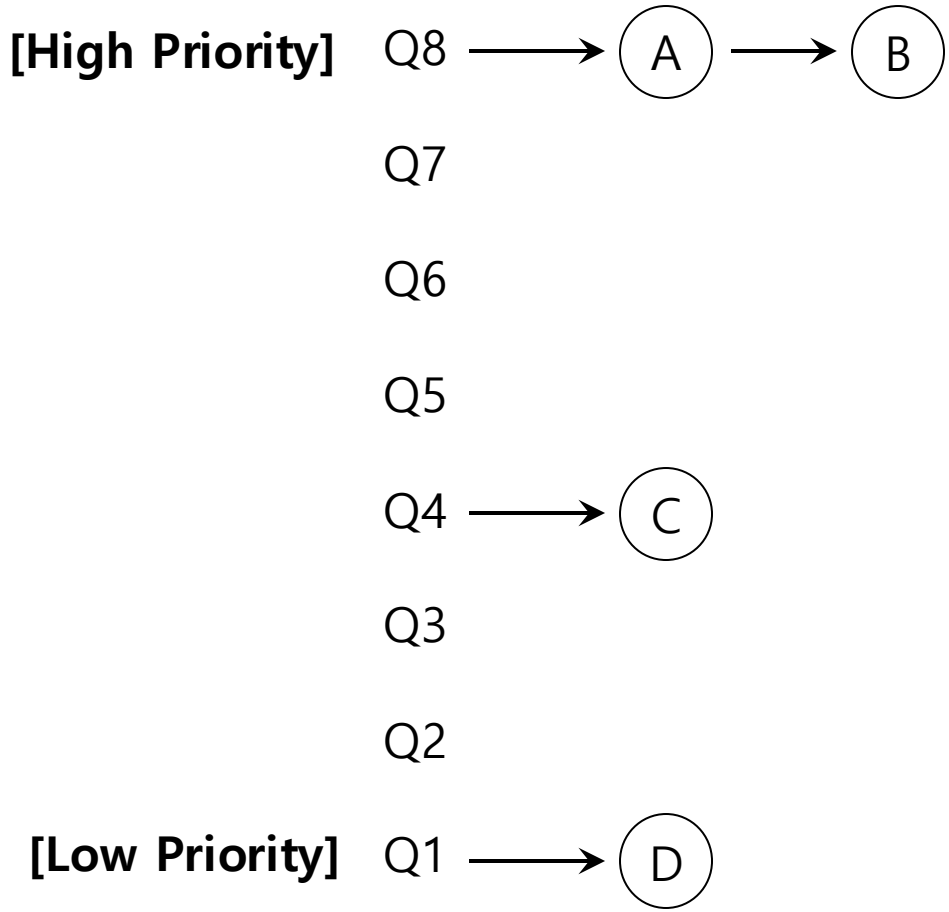
Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
Rule 2: If Priority(A) = Priority(B), A & B run in RR.

MLFQ: Basic Rules (Cont.)

- MLFQ varies the priority of a job based on its observed behavior. 任务的优先级动态变化

- Example:
  - A job repeatedly relinquishes（放弃）the CPU while waiting IOs → Keep its priority high 作业在等待 IO 时反复放弃（放弃）CPU → 保持其优先级高
  - A job uses the CPU intensively for long periods of time → Reduce its priority. 作业长时间密集使用 CPU → 降低其优先级

MLFQ Example



Attempt 1: How to Change Priority

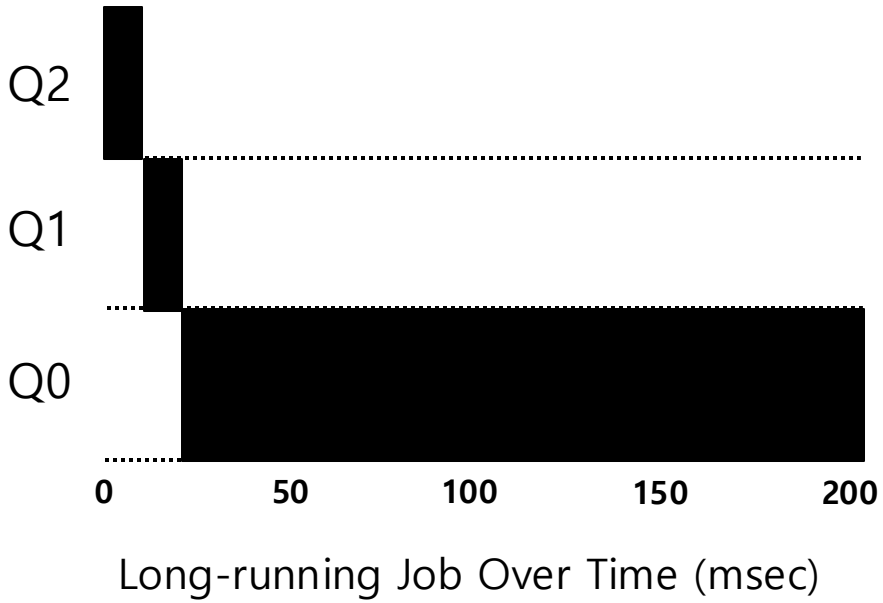
- MLFQ priority adjustment algorithm:
  - **Rule 3**: When a job enters the system, it is placed at the highest priority 当作业进入系统时，它被置于最高优先级
  - **Rule 4a**: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down on queue). 计算密集型任务，优先级逐步调低
  - **Rule 4b**: If a job gives up the CPU before the time slice is up, it stays at the same priority level 交互型任务，优先级保留

In this manner, MLFQ approximates SJF

46

Example 1: A Single Long-Running Job

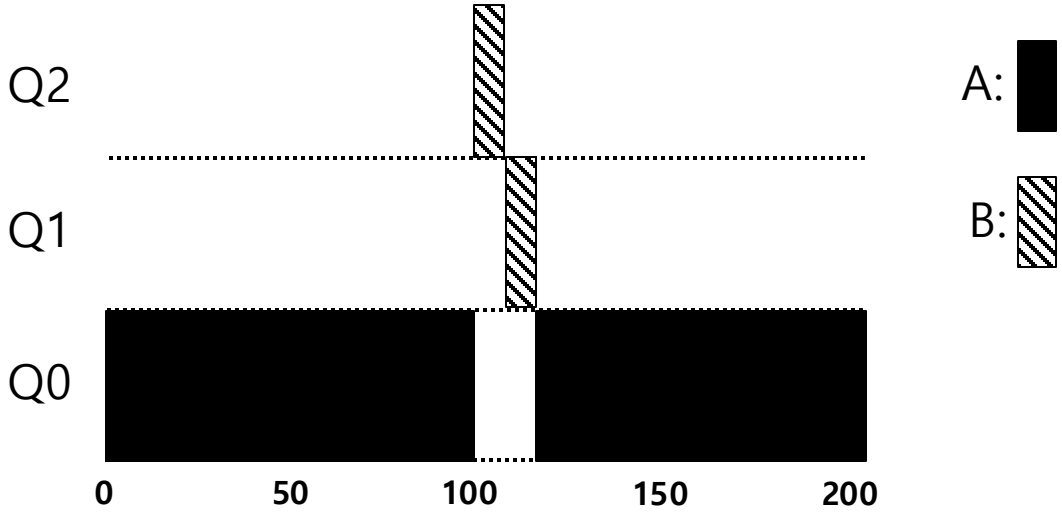
- A three-queue scheduler with time slice 10ms



Example 2: Along Came a Short Job

实例2：来了一个短任务

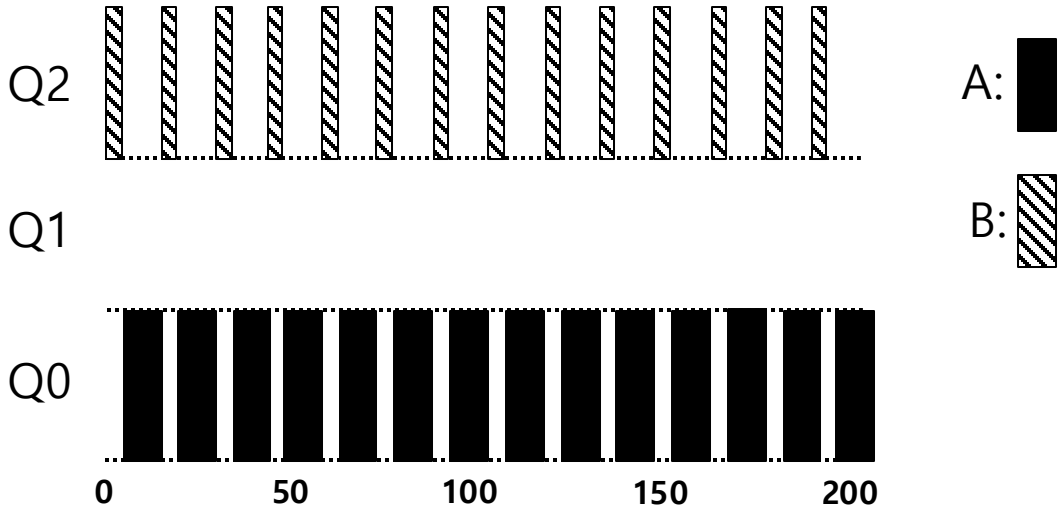
- Assumption:
  - **Job A**: A long-running CPU-intensive job
  - **Job B**: A short-running interactive job (20ms runtime)
  - A has been running for some time, and then B arrives at time T=100.



Along Came An Interactive Job (msec)

Example 3: What About I/O?

- Assumption:
  - **Job A**: A long-running CPU-intensive job
  - **Job B**: An interactive job that need the CPU only for 1ms before performing an I/O



A Mixed I/O-intensive and CPU-intensive Workload (msec)

The MLFQ approach keeps an interactive job at the highest priority

Problems with the Basic MLFQ

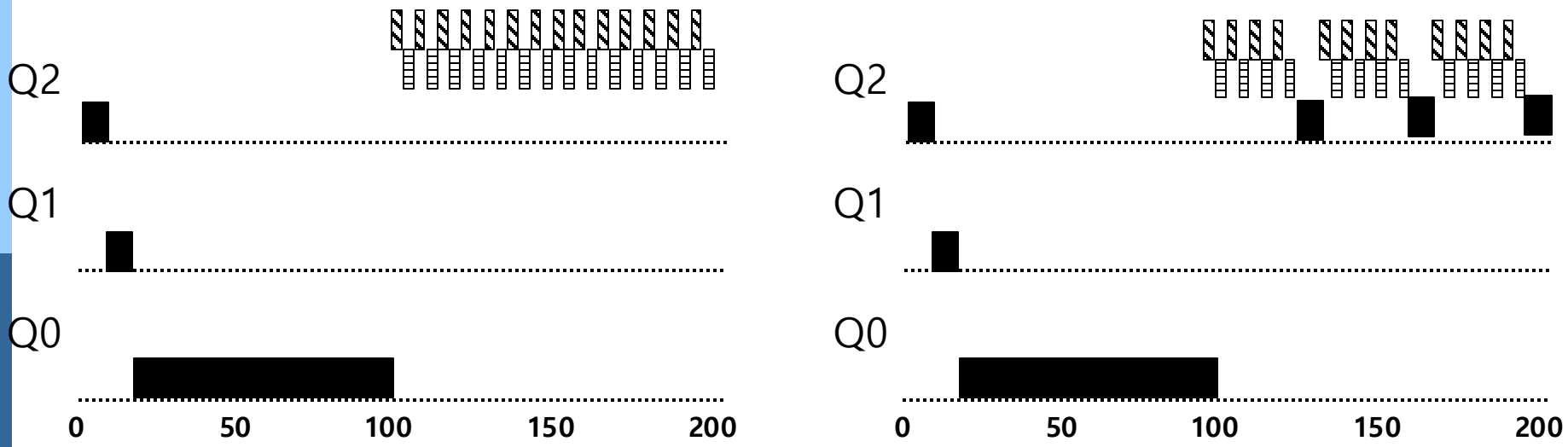
基础MLFQ的一些问题

- Starvation 首先，会有饥饿问题
  - If there are “too many” interactive jobs in the system.
  - Lon-running jobs will never receive any CPU time.
- Game the scheduler 其次，聪明的用户可能重写程序来愚弄调度程序
  - After running 99% of a time slice, issue an I/O operation.
  - The job gain a higher percentage of CPU time.
- A program may change its behavior over time. 最后，一个程序可能在不同时间段表现不同
  - CPU bound process → I/O bound process

Attempt 2: The Priority Boost

尝试2：提升优先级

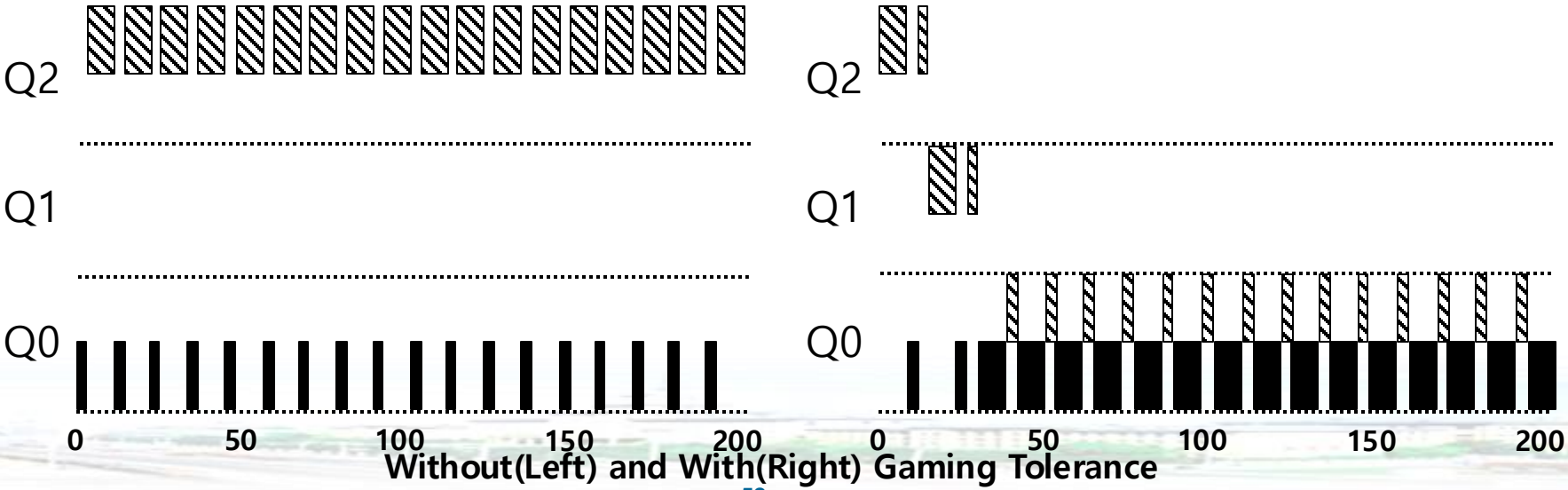
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue. 经过一段时间 S 后，将系统中的所有作业移动到最顶层队列
  - Example:
    - A long-running job(A) with two short-running interactive job(B, C)



Without(Left) and With(Right) Priority Boost A: B: C:

Attempt 3: Better Accounting

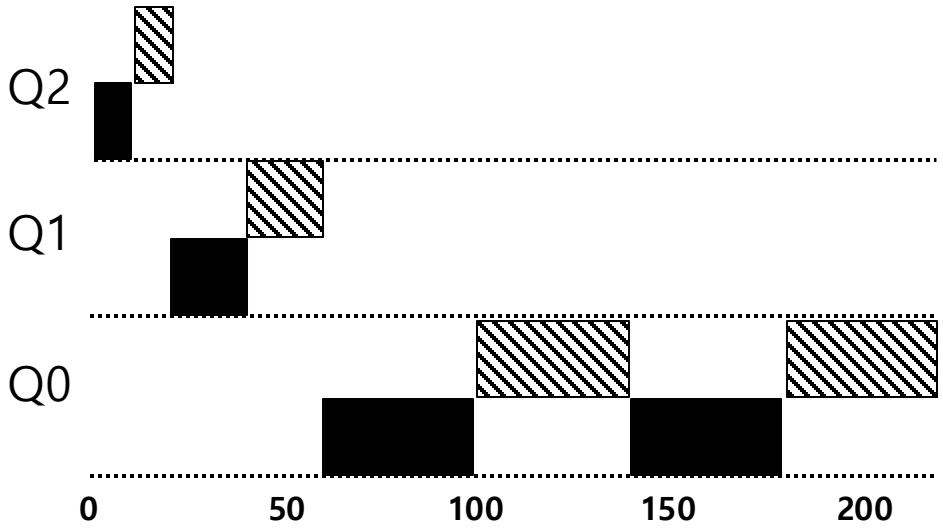
- How to prevent gaming of our scheduler?
  - Rules 4a and 4b：which let a job retain its priority by relinquishing the CPU before the time slice expires 通过在时间片到期之前放弃 CPU 让作业保留其优先级
- Solution:
  - **Rule 4** (Rewrite Rules 4a and 4b): Once a job uses up its time allotment (时间配额) at a given level (regardless of how many times it has given up the CPU), **its priority is reduced**(i.e., it moves down on queue). 一旦作业在给定级别用完其时间分配（时间损耗）（无论它放弃了多少次 CPU），它的优先级就会降低（即，它在队列中向下移动）



Tuning MLFQ And Other Issues

Lower Priority, Longer Quanta

- The high-priority queues → Short time slices
  - E.g., 10 or fewer milliseconds
- The Low-priority queue → Longer time slices
  - E.g., 100 milliseconds



Example) 10ms for the highest queue, 20ms for the middle, 40ms for the lowest

The Solaris MLFQ implementation

- For the Time-Sharing scheduling class (TS)
  - 60 Queues
  - Slowly increasing time-slice length
    - The highest priority: 20msec
    - The lowest priority: A few hundred milliseconds
  - Priorities boosted around every 1 second or so.

MLFQ: Summary

- The refined set of MLFQ rules:
  - **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
  - **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.
  - **Rule 3:** When a job enters the system, it is placed at the highest priority.
  - **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced(i.e., it moves down on queue).
  - **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

MLFQ

【例题】系统采用二级反馈队列调度算法进行进程调度。就绪队列Q1采用时间片轮转调度算法，时间片为10ms；就绪队列Q2采用短进程优先调度算法；系统优先调度Q1队列中的进程，当Q1为空时系统才会调度Q2中的进程；新创建的进程首先进入Q1；Q1中的进程执行一个时间片后，若未结束，则转入Q2。若当前Q1、Q2为空，系统依次创建进程P1、P2后即开始进程调度，P1、P2需要的CPU时间分别为30ms和20ms，则进程P1、P2在系统中的平均等待时间为（）。

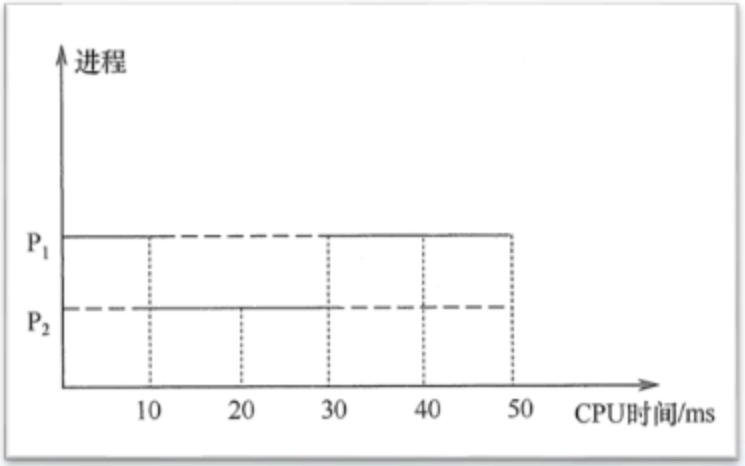
A. 25ms　　B. 20ms　　C. 15ms　　D. 10ms

答案：C

解析：进程P1、P2依次进入Q1后，将依次被分配10ms的CPU时间，两个进程执行完一个时间片后转入Q2队列。此时P1还需要20ms，P2还需要10ms，根据Q2的SJF算法，P2会被优先调度进入Q1，10ms后P2执行结束，之后P1再被调度执行，20ms后执行结束。

进程P1、P2的等待时间为图中的虚横线部分。

平均等待时间 = (P1等待时间 + P2等待时间)/2 = (20 + 10)/2 = 15



Module 3：调度

1. Mechanism: Limited Direct Execution

2. Policy: Scheduling Algorithms

2.1 Introduction – 基础

- 概述 – 基础
- 经典调度算法 – 基础

2.2 The Multi-Level Feedback Queue – 基础

2.3 Proportional Share – 进阶

Proportional Share Scheduler

比例份额调度

- Fair-share scheduler 公平份额调度程序
  - Guarantee that each job obtain *a certain percentage* of CPU time.
    保证每个作业获得一定比例的CPU时间
  - Not optimized for turnaround or response time
    未针对周转时间或响应时间进行优化

Basic Concept

- Tickets（彩票数）
  - Represent the share of a resource that a process should receive
    表示进程应接收的资源份额
  - The percent of tickets represents its share of the system resource in question. 彩票数的百分比表示其在相关系统资源中的份额

- Example
  - There are two processes, A and B.
    - Process A has 75 tickets → receive 75% of the CPU
    - Process B has 25 tickets → receive 25% of the CPU

Lottery scheduling 彩票调度

- The scheduler picks a winning ticket.
  - Load the state of that *winning process* and runs it.
    加载该获胜进程的状态并运行它
- Example
  - There are 100 tickets
    - Process A has 75 tickets: 0 ~ 74
    - Process B has 25 tickets: 75 ~ 99

Scheduler's winning tickets:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Resulting scheduler:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

The longer these two jobs compete,
The more likely they are to achieve the desired percentages.

Ticket Mechanisms 彩票机制

- Ticket currency 彩票货币
  - A user allocates tickets among their own jobs in whatever currency they would like. 拥有彩票的用户将彩票分配给他们的不同工作。
  - The system converts the currency into the correct global value.

    之后，系统会自动地将货币兑换为正确的全局变量。
  - Example
    - There are 200 tickets (Global currency)
    - User A has 100 tickets
    - User B has 100 tickets

User A → *500* (A's currency) to A1 → *50* (global currency)
→ *500* (A's currency) to A2 → *50* (global currency)

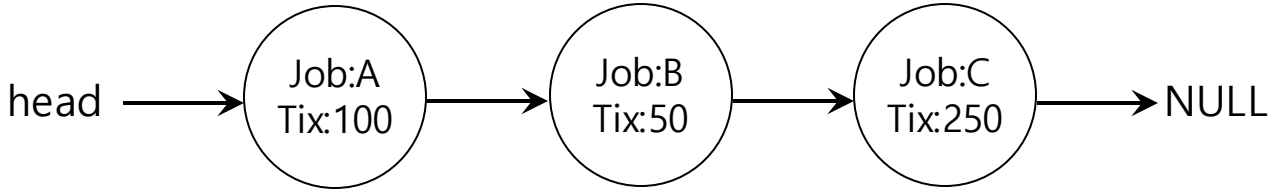
User B → *10* (B's currency) to B1 → *100* (global currency)

Ticket Mechanisms 彩票机制 (Cont.)

- Ticket transfer 彩票转让
  - A process can temporarily hand off *its tickets* to another process.
    一个进程可以暂时将它的彩票交给另一个进程。
- Ticket inflation 彩票通胀
  - A process can temporarily raise or lower the number of tickets is owns.
    进程可以临时增加或减少拥有的彩票数量。
  - If any one process needs *more CPU time*, it can boost its tickets.
    如果任何一个进程需要更多的 CPU 时间，它可以增加它的彩票。

Implementation 实现

- Example: There are three processes, A, B, and C.
  - Keep the processes in a list:



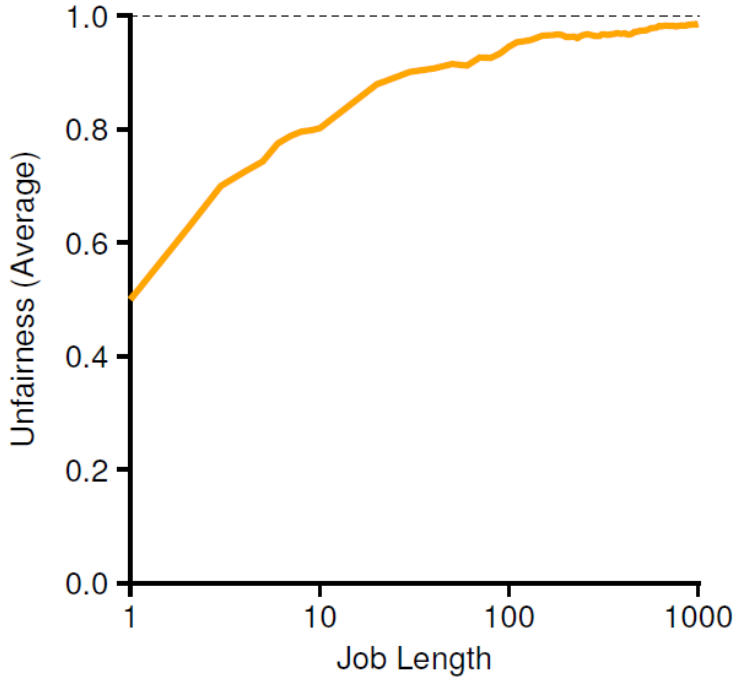

```
1         // counter: used to track if we've found the winner yet
2         int counter = 0;
3
4         // winner: use some call to a random number generator to
5         // get a value, between 0 and the total # of tickets
6         int winner = getrandom(0, totaltickets);
7
8         // current: use this to walk through the list of jobs
9         node_t *current = head;
10
11        // loop until the sum of ticket values is > the winner
12        while (current) {
13                  counter = counter + current->tickets;
14                  if (counter > winner)
15                            break; // found the winner
16                  current = current->next;
17        }
18        // 'current' is the winner: schedule it...
```

Example: Lottery Scheduling

- U: unfairness metric 不公平指标
  - The time the first job completes divided by the time that the second job completes. 两个任务完成时刻相除得到U的值
- Example:
  - There are two jobs, each jobs has runtime 10.
    - First job finishes at time 10
    - Second job finishes at time 20
  - $U = \frac{10}{20} = 0.5$
  - U will be close to 1 when both jobs finish at nearly the same time. 当两个工作几乎同时完成时，U 将接近 1

Lottery Fairness Study 彩票公平性研究

- There are two jobs.
  - Each jobs has the same number of tickets (100).



When the job length is not very long, average unfairness can be quite severe.

Stride Scheduling 步长调度

- Stride of each process 每个进程的步长(与彩票的数量成反比):
  - (A large number) / (the number of tickets of the process)
  - Example: A large number = 10,000
    - Process A has 100 tickets → stride of A is 100
    - Process B has 50 tickets → stride of B is 200
    - Process C has 250 tickets → stride of C is 40

- A process runs, increment a counter(=pass value) for it by its stride.
  进程运行后，计数器(称为行程 pass)值每次增加它的步长大小
  - Pick the process to run that has the lowest pass value

```
current = remove_min(queue);           // pick client with minimum pass
schedule(current);                     // use resource for quantum
current->pass += current->stride;      // compute next pass using stride
insert(queue, current);                // put back into the queue
```

A pseudo code implementation

Stride Scheduling Example

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

If new job enters with pass value 0,
It will monopolize the CPU!