操作系统
Operating Systems

刘川意 教授

哈尔滨工业大学（深圳）

2021年9月

Module 3：调度

- Mechanism: Limited Direct Execution
- Policy: Scheduling Algorithms
  - Introduction
  - The Multi-Level Feedback Queue
  - Proportional Share

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libray

Problem 1: Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as …
  - Issuing an I/O request to a disk
  - Gaining access to more system resources such as CPU or memory

- **Solution**: Using 保护模式（protected control transfer）
  - User mode: Applications do not have full access to hardware resources.
  - Kernel mode: The OS has access to the full resources of the machine

System Call

- Allow the kernel to carefully expose certain key pieces of functionality to user program, such as …
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory

- **Trap** instruction
  - Jump into the kernel
  - Raise the privilege level to kernel mode

- **Return-from-trap** instruction
  - Return into the calling user program
  - Reduce the privilege level back to user mode

Limited Direction Execution Protocol

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of syscall handler

OS @ run (kernel mode)	Hardware	Program (user mode)
- Create entry for process list - Allocate memory for program - Load program into memory - Setup user stack with argv - Fill kernel stack with reg/PC - **return-from -trap**		
	- restore regs from kernel stack - move to user mode - jump to main	
		- 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 ➢ move to kernel mode ➢ jump to trap handler	
➢ Handle trap ➢ Do work of syscall ➢ **return-from-trap**		
	➢ restore regs from kernel stack ➢ move to user mode ➢ jump to PC after trap	
		➢ … ➢ 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*?
  - A cooperative Approach: **Wait for system calls**
  - A Non-Cooperative Approach: **The OS takes control**

A cooperative Approach: Wait for system calls

- Processes periodically give up the CPU by making **system calls** such as `yield`.
  - The OS decides to run some other task.
  - Application also transfer control to the OS when they do something illegal.
    - Divide by zero
    - Try to access memory that it shouldn't be able to access
  - Example: Early versions of the Macintosh OS, The old Xerox Alto system

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.
  - The timer raise an interrupt every so many milliseconds.
  - When the interrupt is raised :
    - The currently running process is halted.
    - Save enough of the state of the program
    - A pre-configured interrupt handler in the OS runs.

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

Saving and Restoring Context

- Scheduler makes a decision:
  - Whether to continue running the **current process**, or switch to a **different one**.
  - If the decision is made to switch, the OS executes context switch.

Context Switch

- A low-level piece of assembly code
  - **Save a few register values** for the current process onto its kernel stack
    - General purpose registers
    - PC
    - kernel stack pointer
  - **Restore a few** for the soon-to-be-executing process from its kernel stack
  - **Switch to the kernel stack** for the soon-to-be-executing process

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

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
        # Save old registers
        movl 4(%esp), %eax          # put old ptr into eax
        popl 0(%eax)                # save the old IP
        movl %esp, 4(%eax)          # and stack
        movl %ebx, 8(%eax)          # and other registers
        movl %ecx, 12(%eax)
        movl %edx, 16(%eax)
        movl %esi, 20(%eax)
        movl %edi, 24(%eax)
        movl %ebp, 28(%eax)

        # Load new registers
        movl 4(%esp), %eax          # put new ptr into eax
        movl 28(%eax), %ebp         # restore other registers
        movl 24(%eax), %edi
        movl 20(%eax), %esi
        movl 16(%eax), %edx
        movl 12(%eax), %ecx
        movl 8(%eax), %ebx
        movl 4(%eax), %esp          # stack is switched here
        pushl 0(%eax)               # return addr put in place
        ret                         # finally return into new ctxt
```

Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations:
  - **Disable interrupts** during interrupt processing
  - Use a number of sophisticated **locking** schemes to protect concurrent access to internal data structures.

Module 3：调度

- Mechanism: Limited Direct Execution
- Policy: Scheduling Algorithms
  - Introduction
  - The Multi-Level Feedback Queue
  - Proportional Share

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.**
  2. All jobs **arrive** at the same time.
  3. All jobs only use the **CPU** (i.e., they perform no I/O).
  4. The **run-time** of each job is known.

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

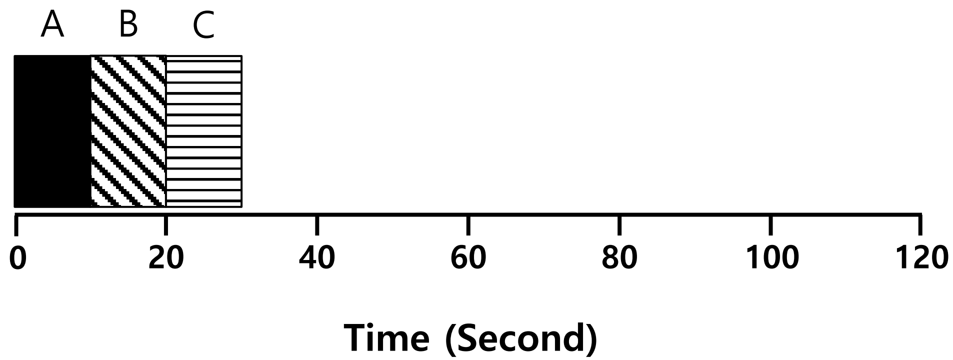
  响应时间是指:从任务到达系统到首次运行(首次被调度)的时间

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

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

First In, First Out 先进先出 (FIFO)

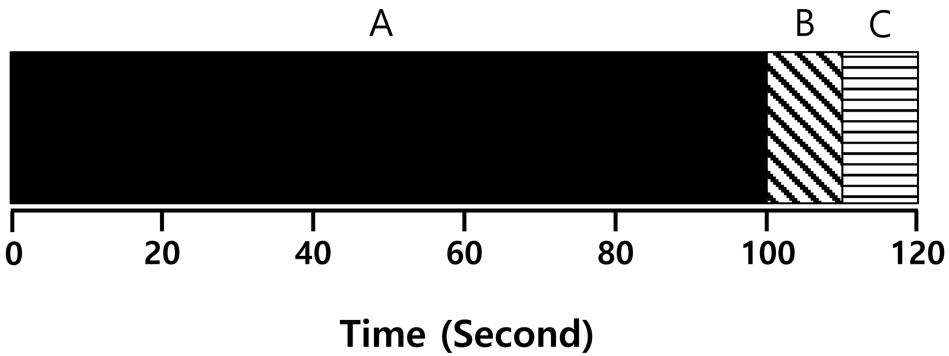
- First Come, First Served (FCFS)
  - Very simple and easy to implement
- Example:
  - A arrived just before B which arrived just before C.
  - Each job runs for 10 seconds.



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

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

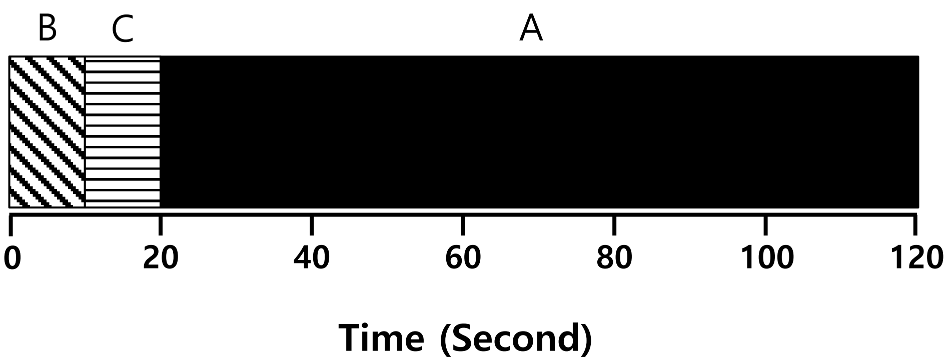
- Let's relax assumption 1: Each job **no longer** runs for the same amount of time.
- Example:
  - A arrived just before B which arrived just before C.
  - A runs for 100 seconds, B and C run for 10 each.



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

Shortest Job First (最短任务优先 SJF)

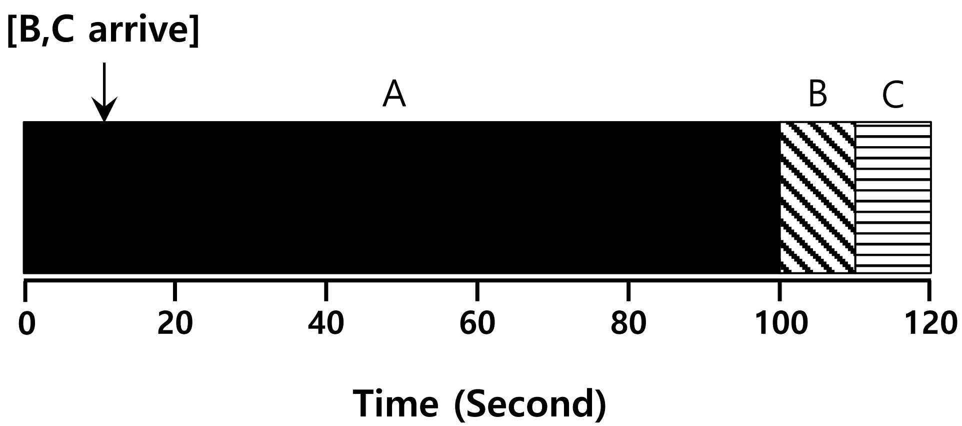
- Run the shortest job first, then the next shortest, and so on
  - Non-preemptive（非抢占式）scheduler
- Example:
  - A arrived just before B which arrived just before C.
  - A runs for 100 seconds, B and C run for 10 each.



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

SJF with Late Arrivals from B and C

- Let's relax assumption 2: Jobs can arrive at any time.
- 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



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

Shortest Time-to-Completion First

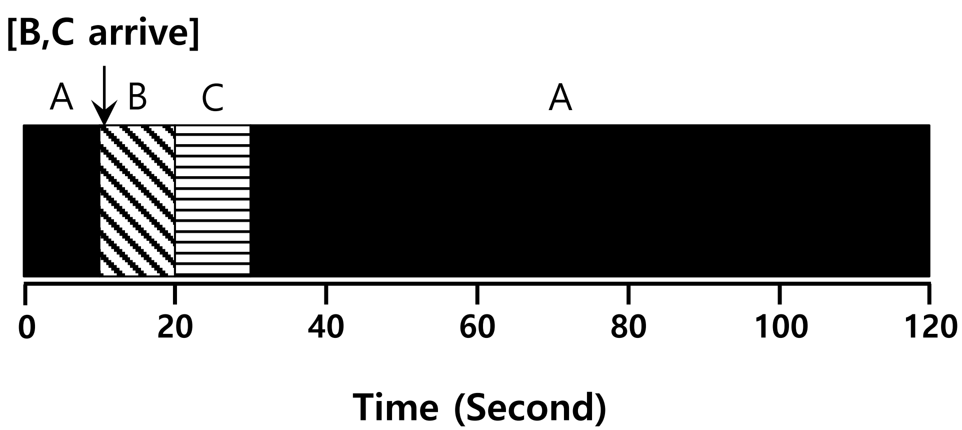
—— 最短完成时间优先 STCF

- Add preemption（抢占）to SJF
  - Also knows as Preemptive Shortest Job First (PSJF)
- A new job enters the system:
  - Determine of the remaining jobs and new job
  - Schedule the job which has the lest time left

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



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

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.

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

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$(3T_1+2T_2+T_3)/3$。

- 【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。

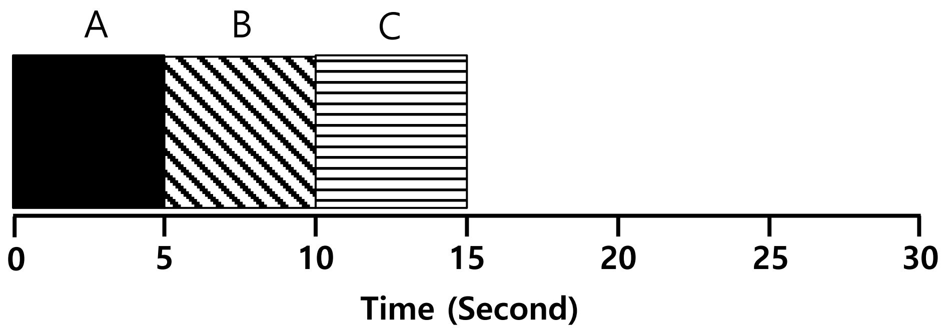
Round Robin (RR) Scheduling

——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）.

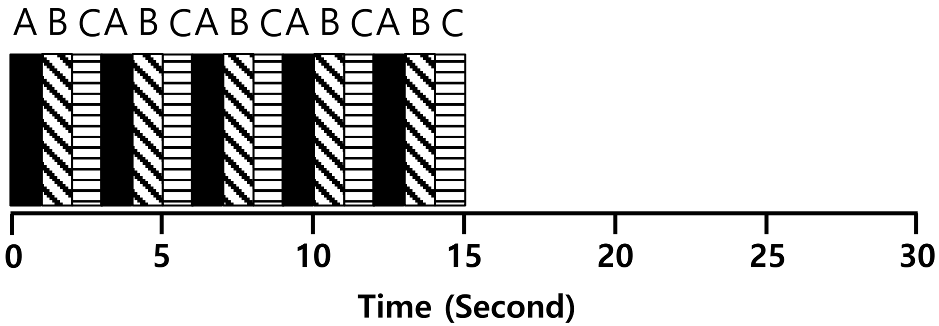
RR Scheduling Example

- 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.
    - 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）.

RR is fair, but performs poorly on metrics such as turnaround time

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。

The length of the time slice is critical.

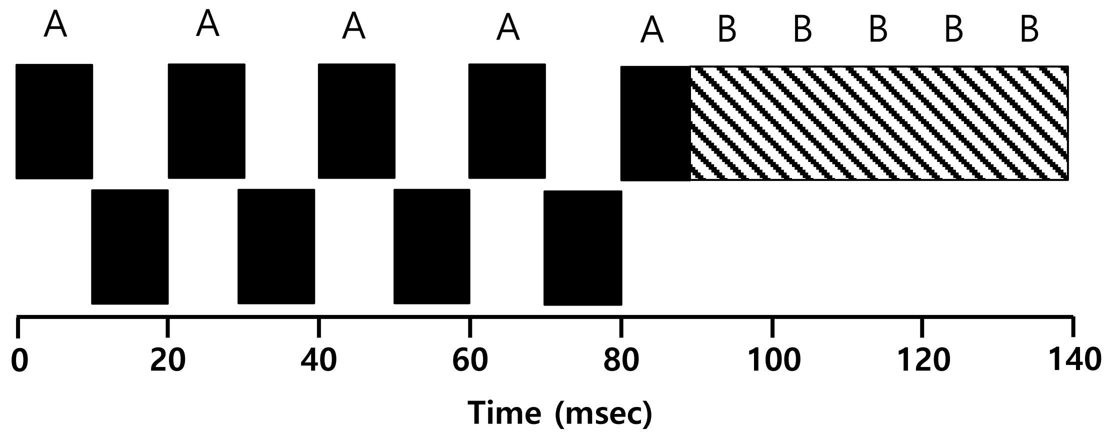
- The shorter time slice
  - Better response time
  - The cost of context switching will dominate overall performance.
- The longer time slice
  - Amortize the cost of context switching 分摊上下文切换的成本
  - Worse response time

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

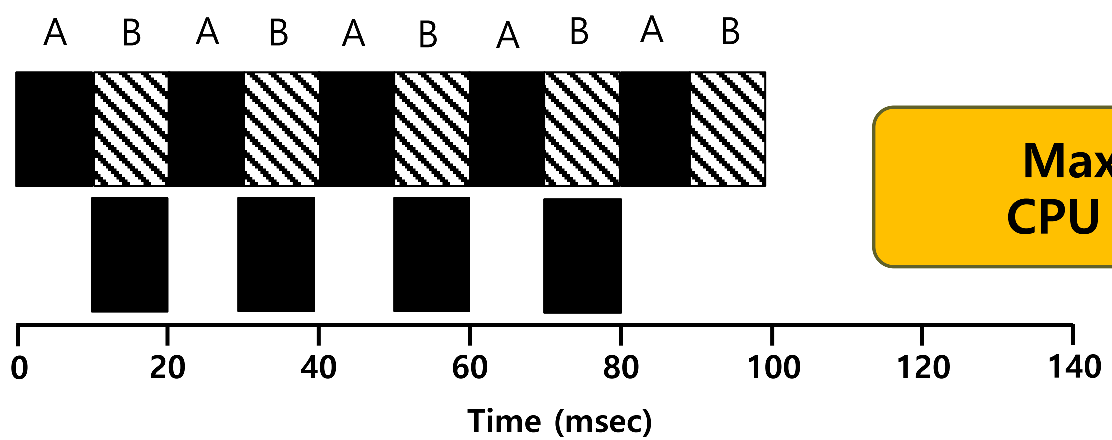
Incorporating I/O

- 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



Overlap Allows Better Use of Resources

Maximize the CPU utilization

Incorporating I/O (Cont.)

- When a job initiates an I/O request.
  - The job is blocked waiting for I/O completion.
  - The scheduler should schedule another job on the CPU.

- When the I/O completes
  - An interrupt is raised.
  - The OS moves the process from blocked back to the ready state.

Module 3：调度

- Mechanism: Limited Direct Execution
- Policy: Scheduling Algorithms
  - Introduction
  - The Multi-Level Feedback Queue
  - Proportional Share

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 priori knowledge of 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.
  - Use round-robin scheduling among jobs in the same queue

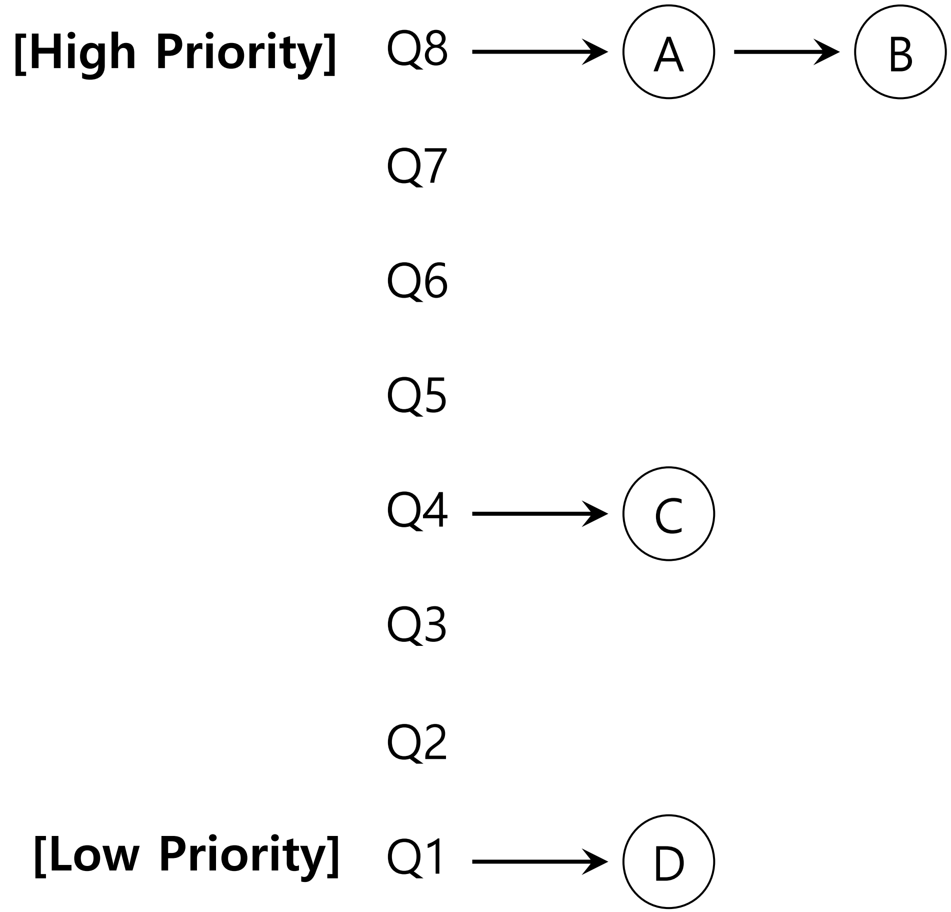
Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
Rule 2: If Priority(A) = Priority(B), A & B run in RR.

MLFQ: Basic Rules (Cont.)

- MLFQ varies the priority of a job based on its observed behavior.

- Example:
  - A job repeatedly relinquishes（放弃）the CPU while waiting IOs → Keep its priority high
  - A job uses the CPU intensively for long periods of time → Reduce its priority.

MLFQ Example



Attempt 1: How to Change Priority

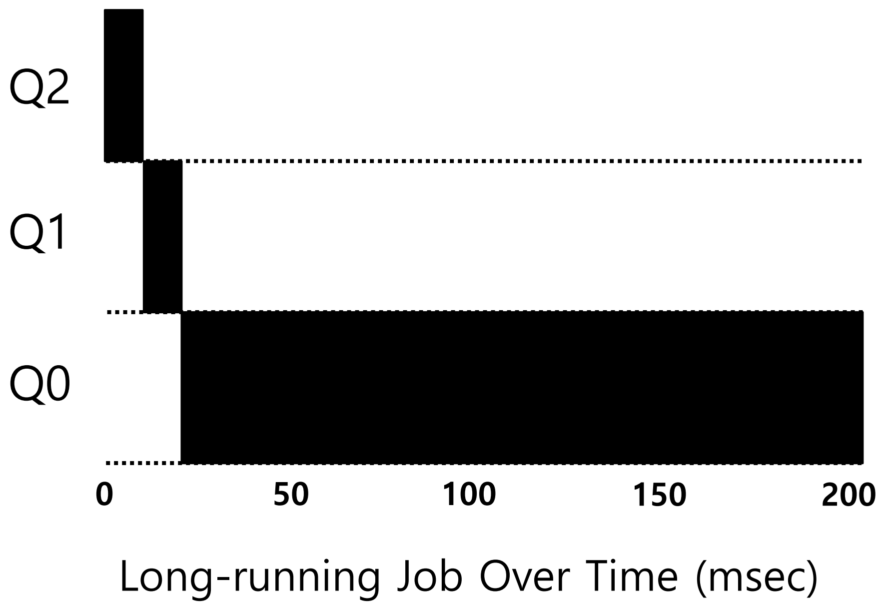
- MLFQ priority adjustment algorithm:
  - **Rule 3**: When a job enters the system, it is placed at the highest priority
  - **Rule 4a**: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down on queue).
  - **Rule 4b**: If a job gives up the CPU before the time slice is up, it stays at the same priority level

In this manner, MLFQ approximates SJF

43

Example 1: A Single Long-Running Job

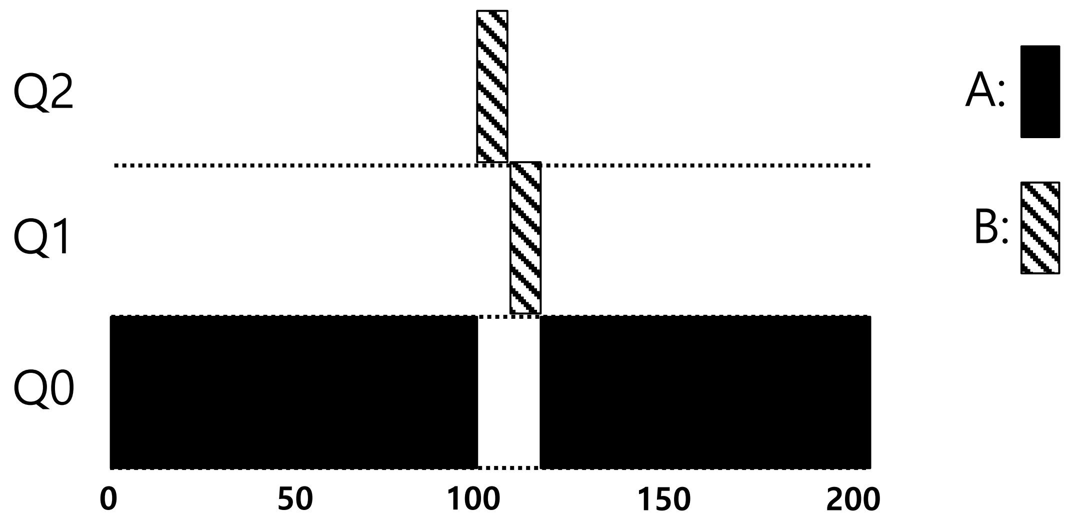
- A three-queue scheduler with time slice 10ms



Example 2: Along Came a Short Job

实例2：来了一个短任务

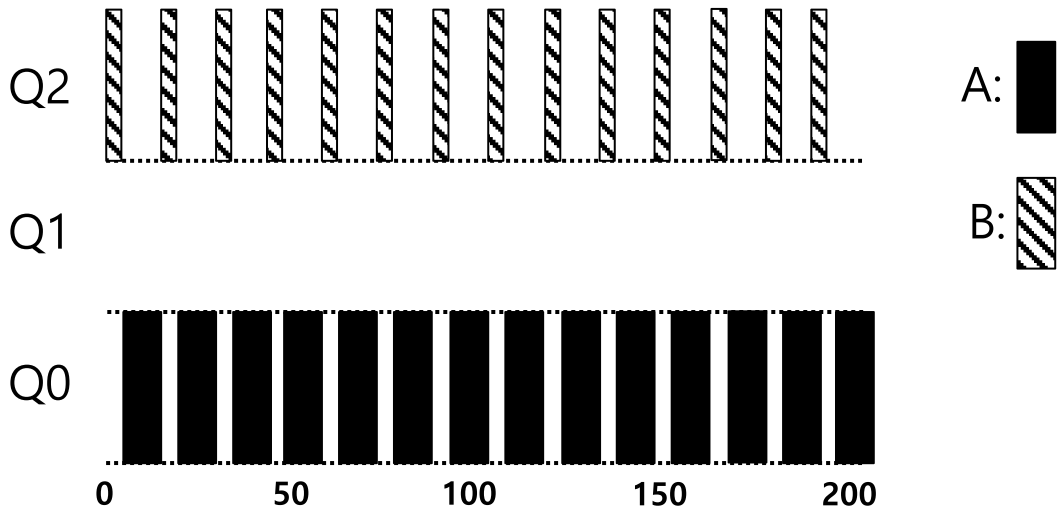
- Assumption:
  - **Job A**: A long-running CPU-intensive job
  - **Job B**: A short-running interactive job (20ms runtime)
  - A has been running for some time, and then B arrives at time T=100.



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.

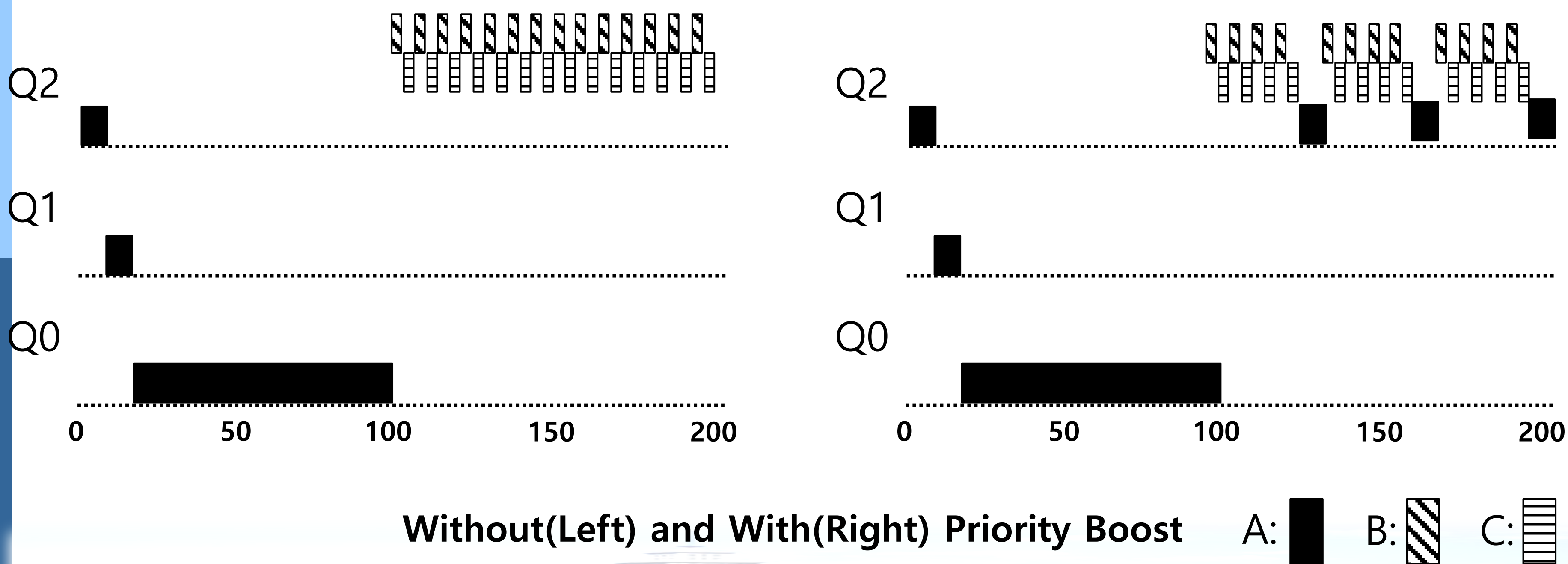
- Game the scheduler 其次，聪明的用户可能重写程序来愚弄调度程序
  - After running 99% of a time slice, issue an I/O operation.
  - The job gain a higher percentage of CPU time.

- 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

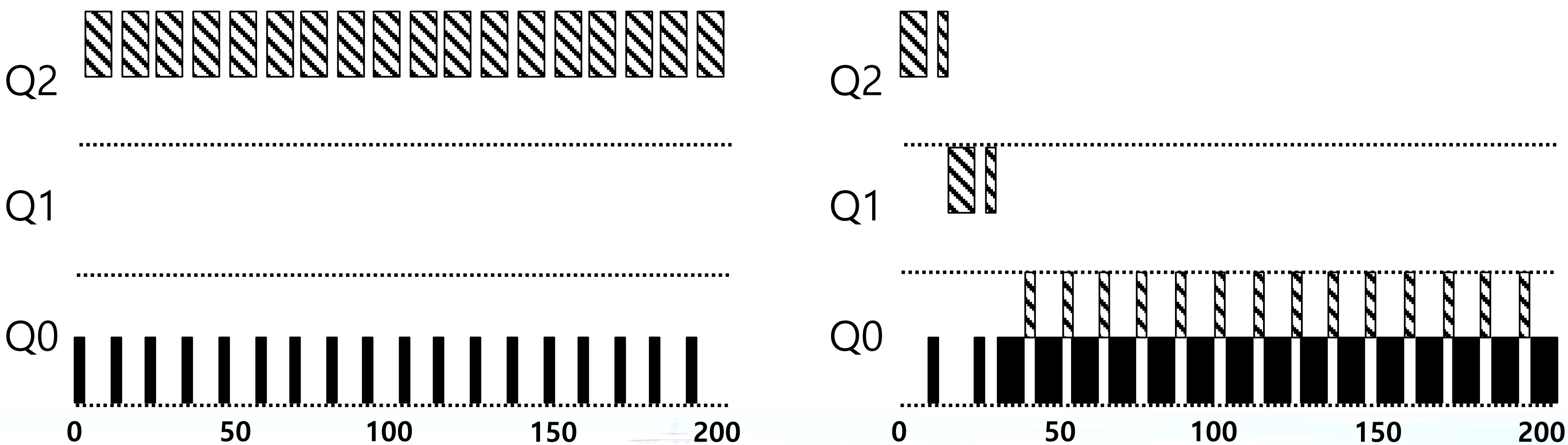
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.
  - Example:
    - A long-running job(A) with two short-running interactive job(B, C)



Without(Left) and With(Right) Priority Boost

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
- 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).

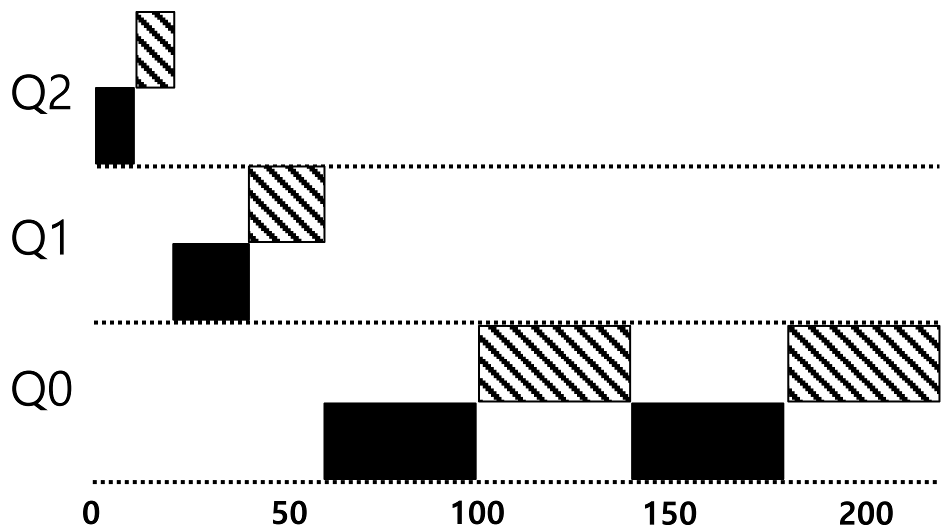


Without(Left) and With(Right) Gaming Tolerance

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.

【例题】若每个任务只能建立一个进程，为了照顾短任务用户，应采用（ ）；为了照顾紧急任务用户，应采用（ ）；
为了能更好地实现人机交互，应采用( )；
而能使短任务、长任务和交互型任务用户都满意，应采用（ ）。

A. FCFS调度算法　　B. SJF调度算法　　C. RR调度算法

D. MLFQ调度算法　　E. 基于优先级的抢占式调度算法

答案：B、E、C、D

Module 3：调度

- Mechanism: Limited Direct Execution
- Policy: Scheduling Algorithms
  - Introduction
  - The Multi-Level Feedback Queue
  - Proportional Share

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.
  - Not optimized for turnaround or response time

Basic Concept

- Tickets（彩票数）
  - Represent the share of a resource that a process should receive
  - The percent of tickets represents its share of the system resource in question.

- 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.
- 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. 用户(拥有彩票的用户)以某种货币的方式，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彩票
  - 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)

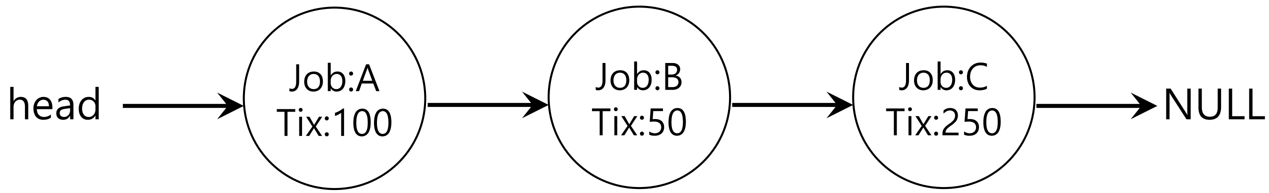
User B → *10* (B's currency) to B1 → *100* (global currency)

Ticket Mechanisms 彩票机制 (Cont.)

- Ticket transfer 彩票转让
  - A process can temporarily hand off *its tickets* to another process.
- Ticket inflation 彩票通胀
  - A process can temporarily raise or lower the number of tickets is owns.
  - If any one process needs *more CPU time*, it can boost its tickets.

Implementation 实现

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



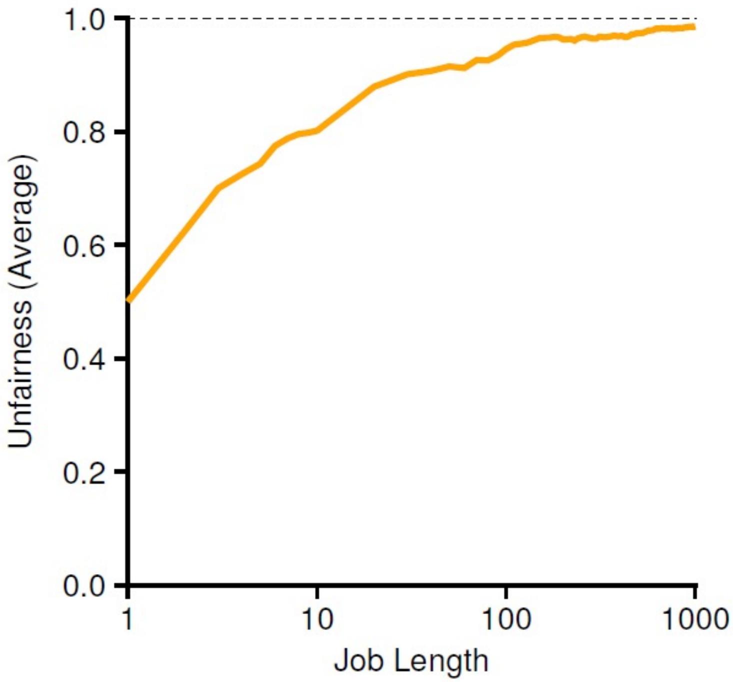

```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
// get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
        counter = counter + current->tickets;
        if (counter > winner)
                break; // found the winner
        current = current->next;
}
// '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.

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

哈爾濱工業大學 HARBIN INSTITUTE OF TECHNOLOGY

- 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!