CISC 3320

C14c. CPU Scheduling: Operating System Examples

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- These slides are a revision of the slides provided by the authors of the textbook via the publisher of the textbook

Outline

- Operating Systems Examples
  - Linux scheduling
  - Windows scheduling
  - Solaris scheduling

- Algorithm Evaluation

Linux Scheduling

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order $O(1)$ scheduling time
- "Complete Fair Scheduler" since version 2.6.23

Linux Variation of Unix Scheduling

- Not designed with SMP systems in mind
  - Did not adequately support systems with multiple processors.
  - In addition, it resulted in poor performance for systems with a large number of runnable processes.

Linux “O(1)” Scheduler

- Preemptive, priority based
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- Map into global priority with numerically lower values indicating higher priority
- Higher priority gets larger quantum
- Task run-able as long as time left in time slice (**active**)
- If no time left (**expired**), not run-able until all other tasks use their slices
- All run-able tasks tracked in per-CPU **runqueue** data structure
  - Two priority arrays (active, expired)
  - Tasks indexed by priority
  - When no more active, arrays are exchanged
- Poor response times for interactive processes

Linux "CFS" Scheduler

- Scheduling class
- Time quantum
- Virtual runtime
- Data structure for the "ready queue"
- Real-time scheduling
- Priority and nice value

Scheduling Class

- Each process has specific priority
- Scheduler picks highest priority task in highest scheduling class
- Rather than quantum based on fixed time allotments, based on proportion of CPU time
- 2 scheduling classes included, others can be added
    1. default
    2. real-time

Quantum

- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases

Virtual Rumtime

- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority (nice = 0) yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

“Fair” Scheduling: Example

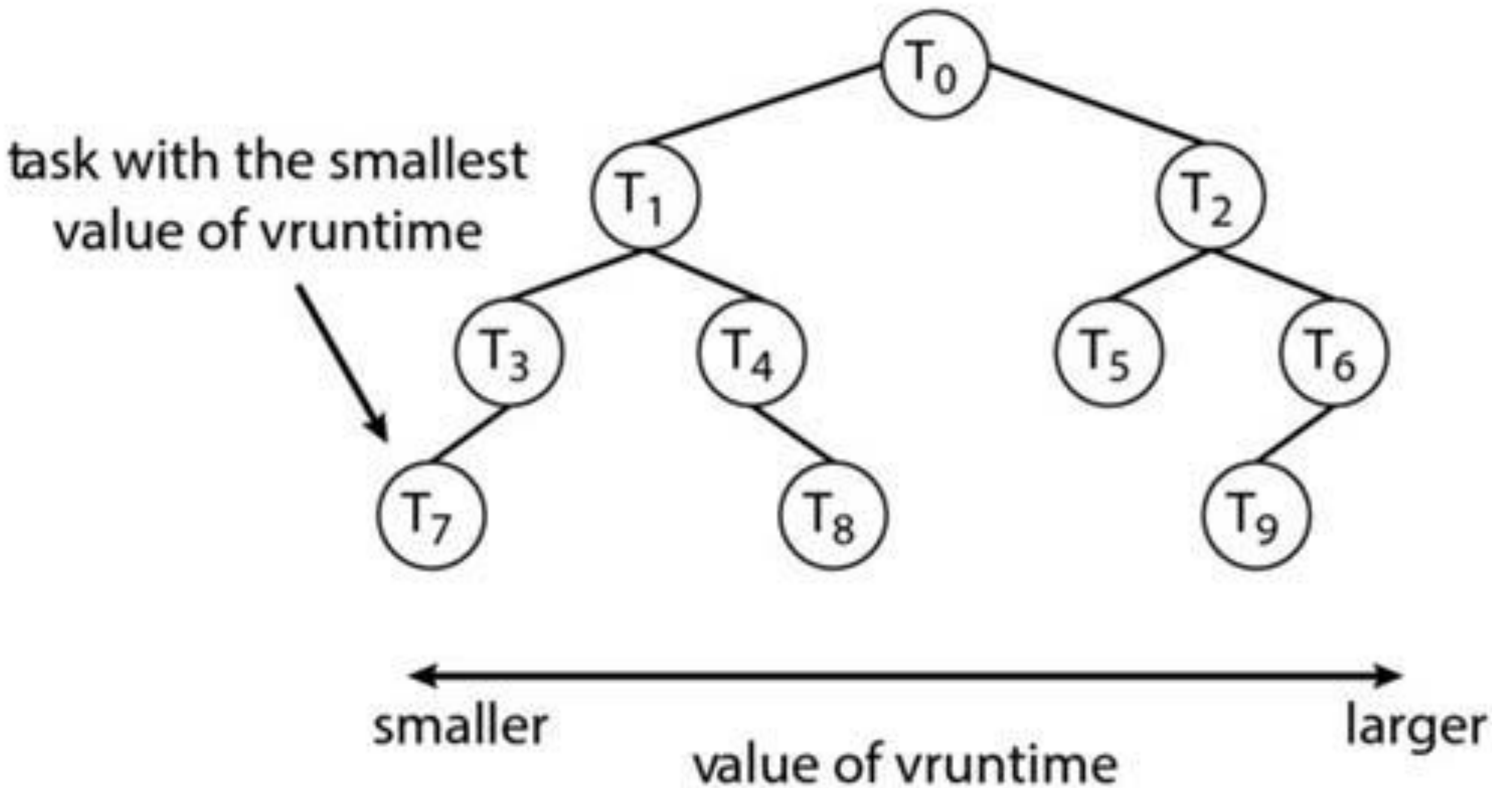
- Two processes have the same nice values
  - P1: I/O bound
  - P2: CPU bound
- Observation
  - I/O bound task run only for short periods before blocking for additional I/O
  - CPU-bound task will exhaust its time period whenever it has an opportunity to run on a CPU
- Result
  - P1 will smaller vruntime than P2
  - P1 will have higher priority than P1
  - When P1 is fulfilled an I/O request, P1 will empty P2 (P1 waited long enough for I/O)

CFS Queue

- Runnable tasks (i.e., processes in the Ready state) are placed in a balanced binary search tree whose key is based on the value of vruntime (a black-red tree).

Selecting a Task

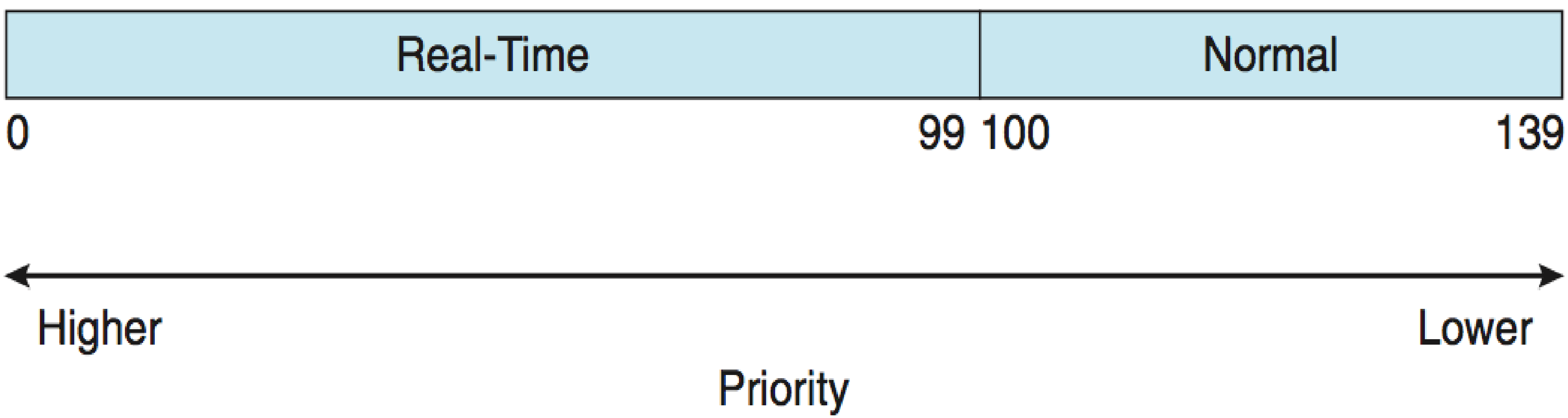
- Taks O(n) time, where n is the number of tasks



Linux Real-time Scheduling

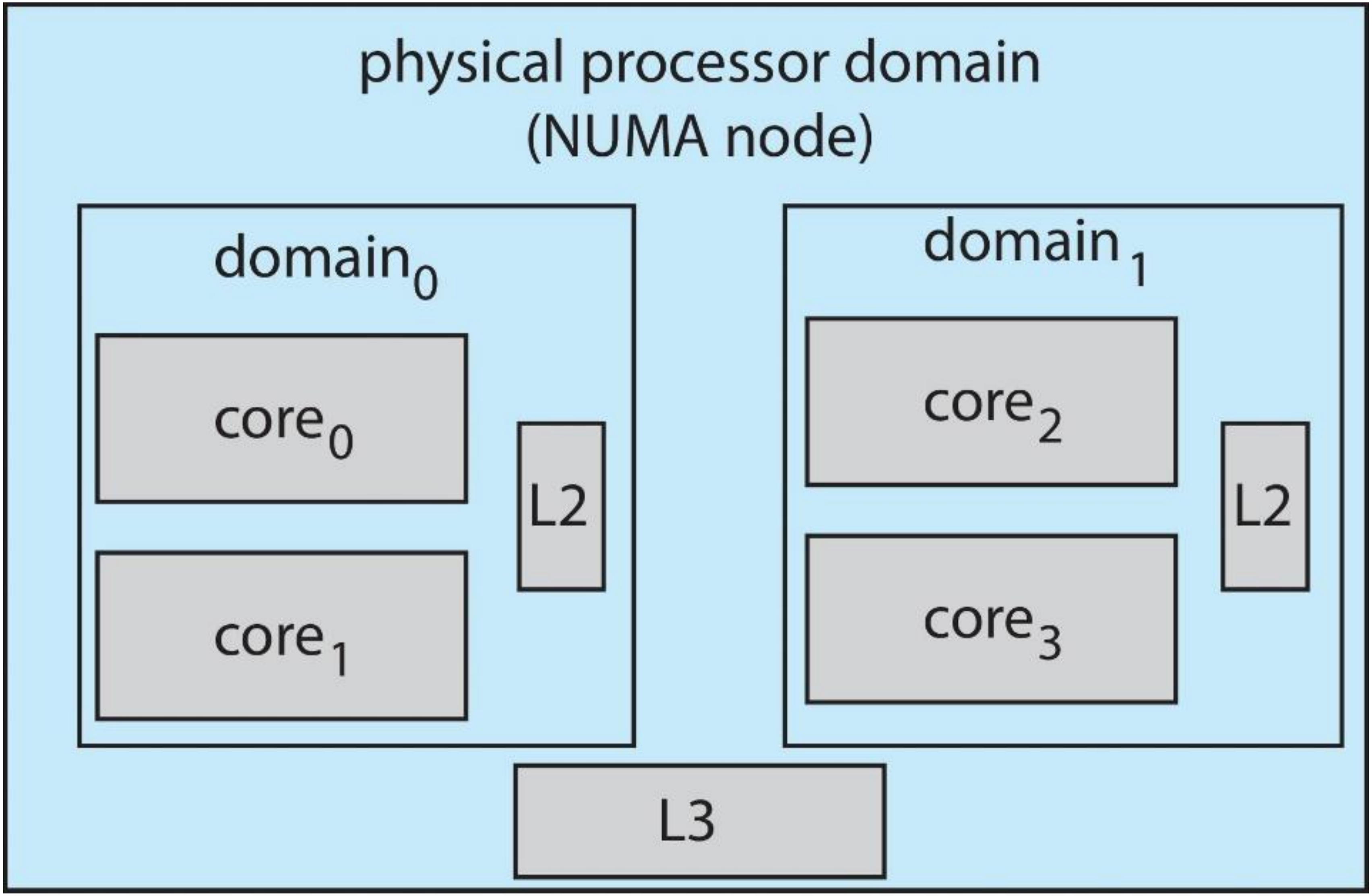
- Real-time scheduling according to POSIX.1b
- Two real-time scheduling policies
  - SCHED_FIFO or the SCHED_RR
- Real-time tasks have static priorities, ranging 0 ~ 99 (vs. normal tasks 100 ~ 139)
- Normal tasks are assigned a priority based on their nice values
  - Nice value of -20 maps to global priority 100
  - Nice value of +19 maps to priority 139

Linux Priority



Linux Load Balancing

- Linux supports load balancing, but is also NUMA-aware.
- **Scheduling domain** is a set of CPU cores that can be balanced against one another.
- Domains are organized by what they share (i.e. cache memory.)
  - Goal is to keep threads from migrating between domains.


physical processor domain
(NUMA node)
domain$_0$
core$_0$
core$_1$
L2
domain$_1$
core$_2$
core$_3$
L2
L3

Questions?

- Evolution of Linux scheduling
- Linux CFS scheduling
  - Scheduling class
  - Time quantum
  - Virtual runtime
  - Data structure for the “ready queue”
  - Real-time scheduling
  - Priority and nice value

Windows Scheduling

- In Windows kernel, the scheduler is called "dispatcher"
- Windows uses priority-based preemptive scheduling
  - Highest-priority thread runs next
  - Thread runs until (1) blocks, (2) time quantum ends, or (3) preempted by higher-priority thread

Windows Priority

- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
  - Real-time threads can preempt non-real-time
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priority Class

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

	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

Variable Priority

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

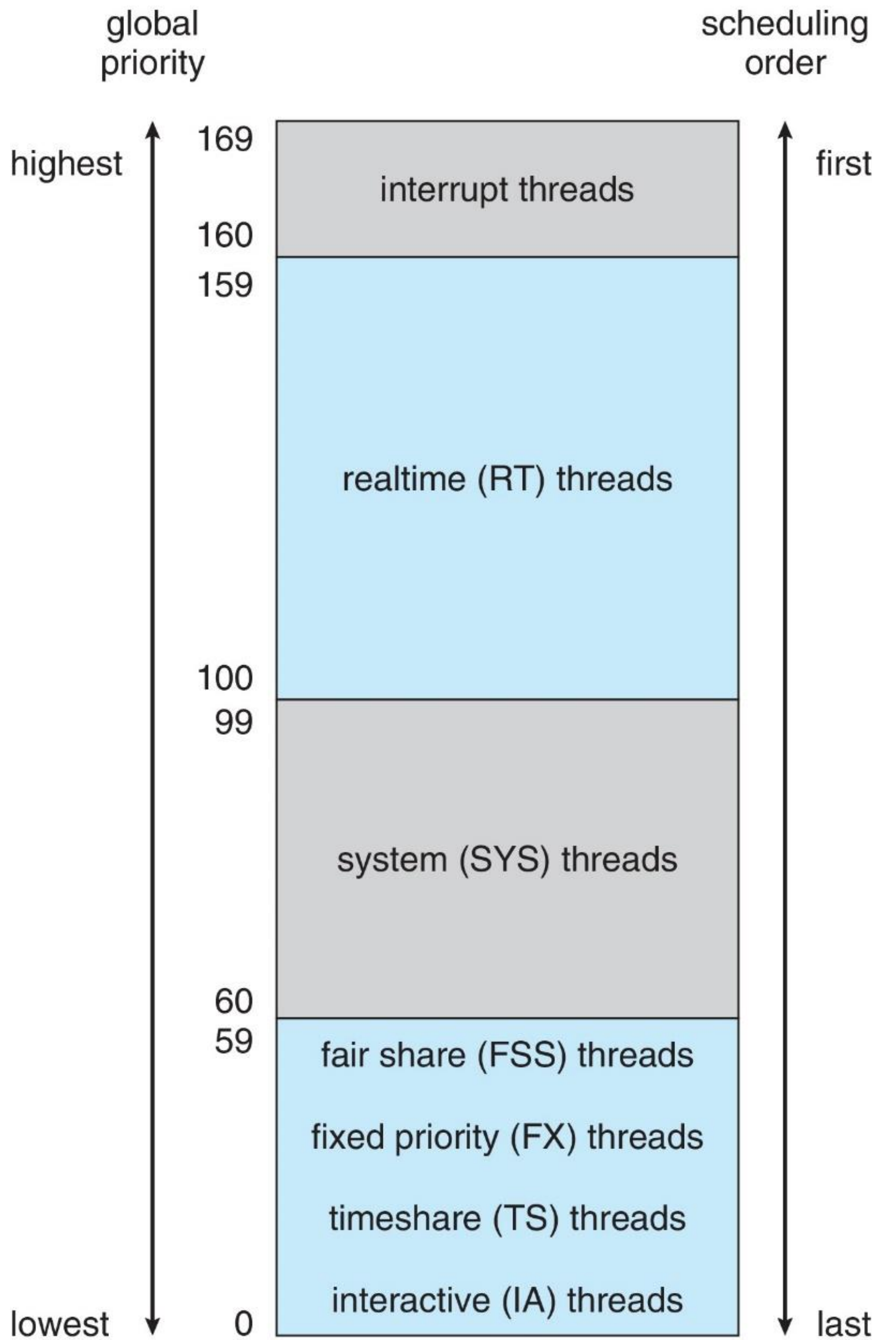
Questions?

- Windows scheduling
- Priority

Solaris

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

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


global priority
scheduling order
highest
first
169
interrupt threads
160
159
realtime (RT) threads
100
99
system (SYS) threads
60
59
fair share (FSS) threads
fixed priority (FX) threads
timeshare (TS) threads
interactive (IA) threads
lowest
0
last

Solaris Priority

- Scheduler converts class-specific priorities into a per-thread global priority
- Solaris uses priority-based preemptive scheduling
  - Highest-priority thread runs next
  - Thread runs until (1) blocks, (2) time quantum ends, or (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

Questions

- Solaris priority

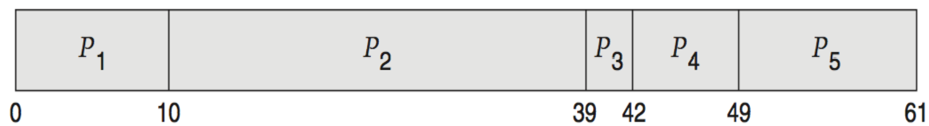
Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes redetermined workload and define ance of each algorithm for that w
- Conside s arriving at time 0:

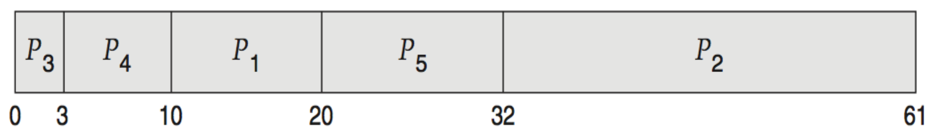
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

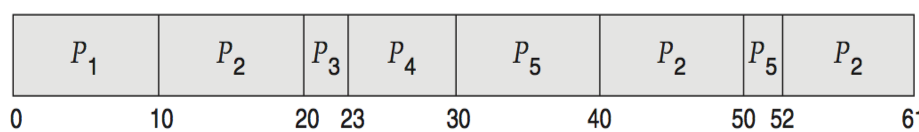
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - Nc



  - RR



Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

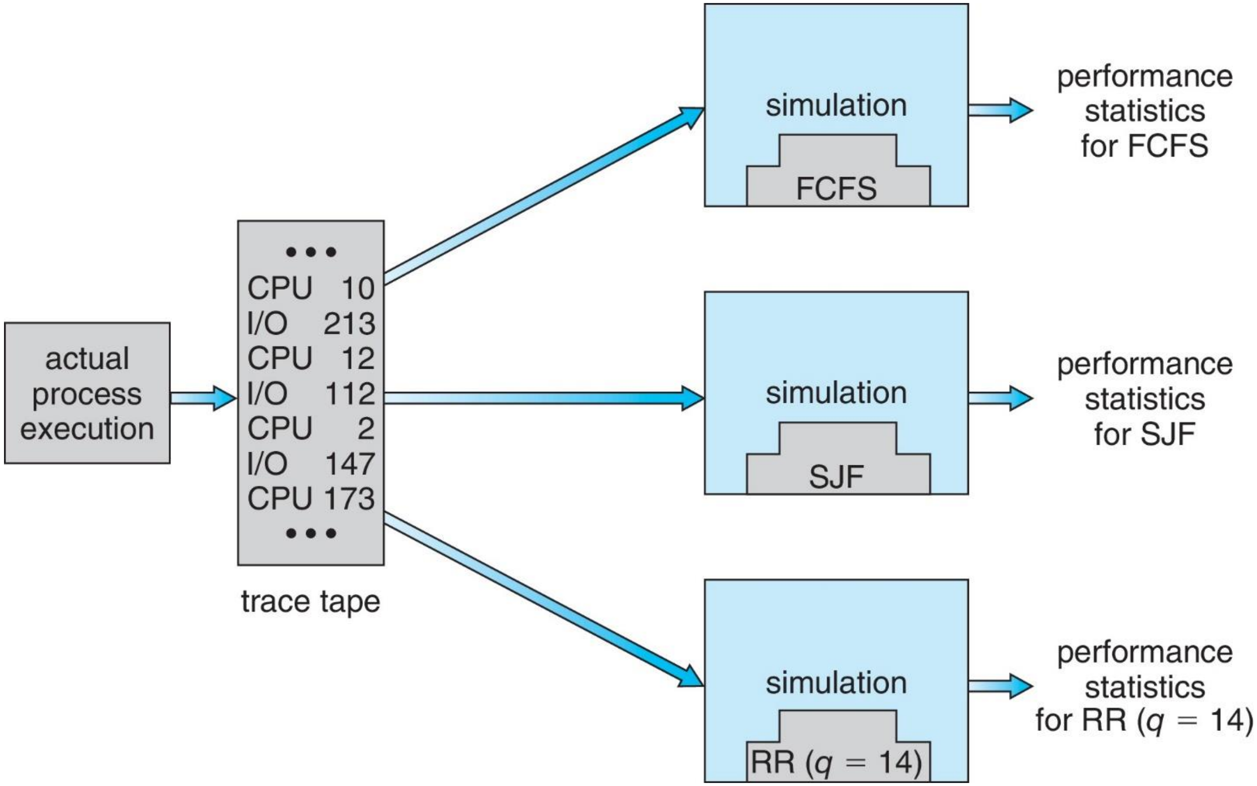
Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:
  $n = \lambda \times W$
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

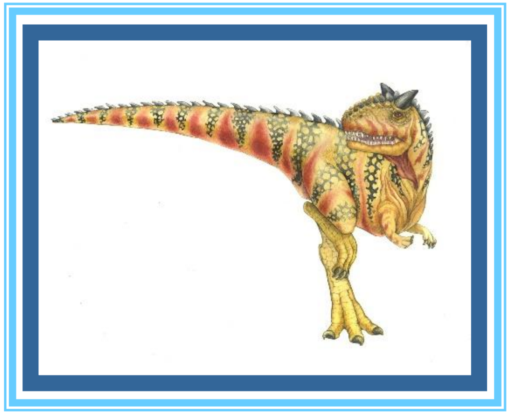
Evaluation of CPU Schedulers by Simulation



Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

End of Chapter 5



Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms