Operating Systems

7. CPU: Scheduling (2)

Hyunchan, Park

http://oslab.chonbuk.ac.kr

Division of Computer Science and Engineering

Chonbuk National University

Contents

- Multiple-Processor Scheduling
- Real-time Scheduling
- Others
  - Proportional share Scheduling
  - Scheduling in Virtualized system
  - Algorithm evaluation
- Operating system examples
  - Linux
  - Windows
  - Solaris

Multiple-Processor Scheduling

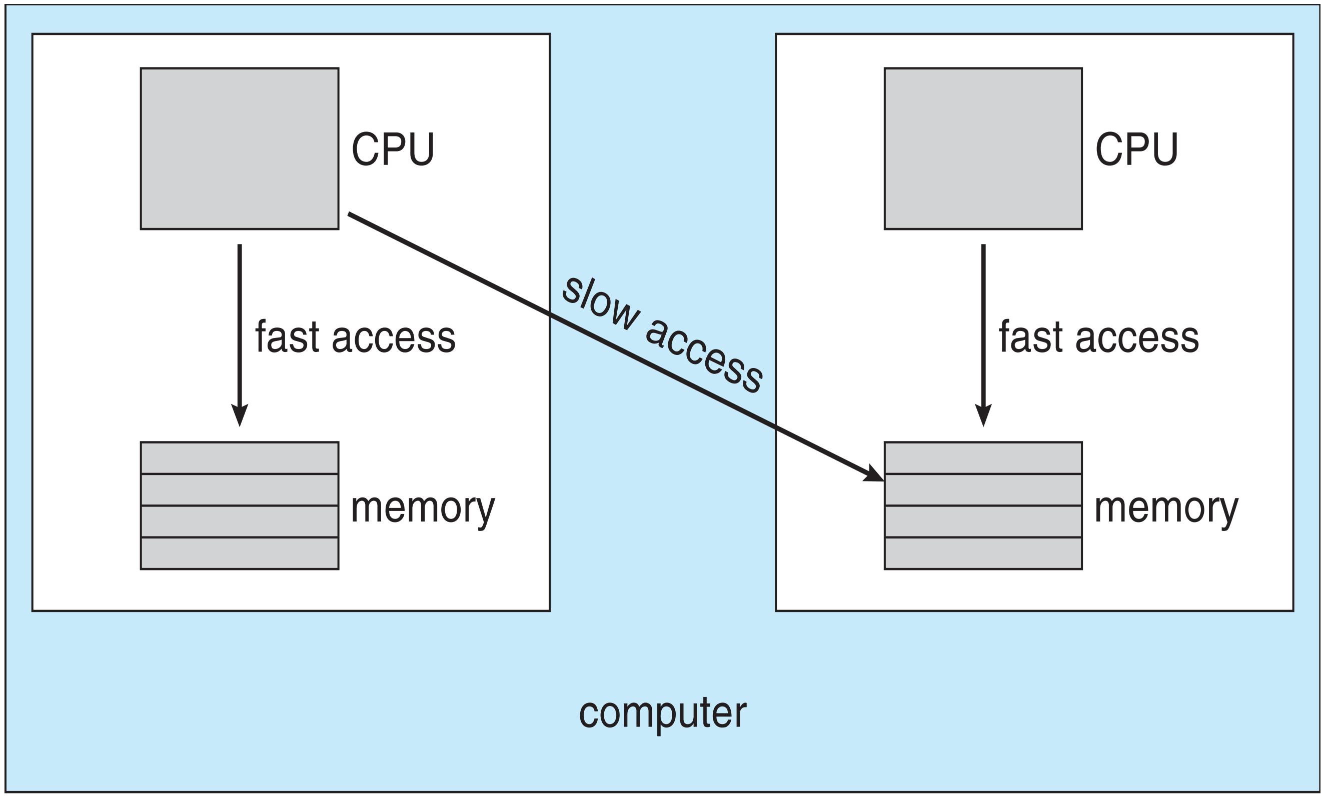
- CPU scheduling more complex when multiple CPUs are available
- Homogeneous or heterogeneous processors
  - Within a multiprocessor
- Asymmetric multiprocessing
  - Only one processor accesses the system data structures, alleviating the need for data sharing
- Symmetric multiprocessing (SMP)
  - Each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common

Multiple-Processor Scheduling

- Processor affinity
  - Process has affinity for processor on which it is currently running
  - Soft affinity
    - Not guaranteeing to keep a process running on a same processor
  - Hard affinity
    - Always on a same processor
  - Variations including processor sets

  - Pros: Load balancing
  - Cons: Performance overhead
    - because of context switches between two processors

NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity

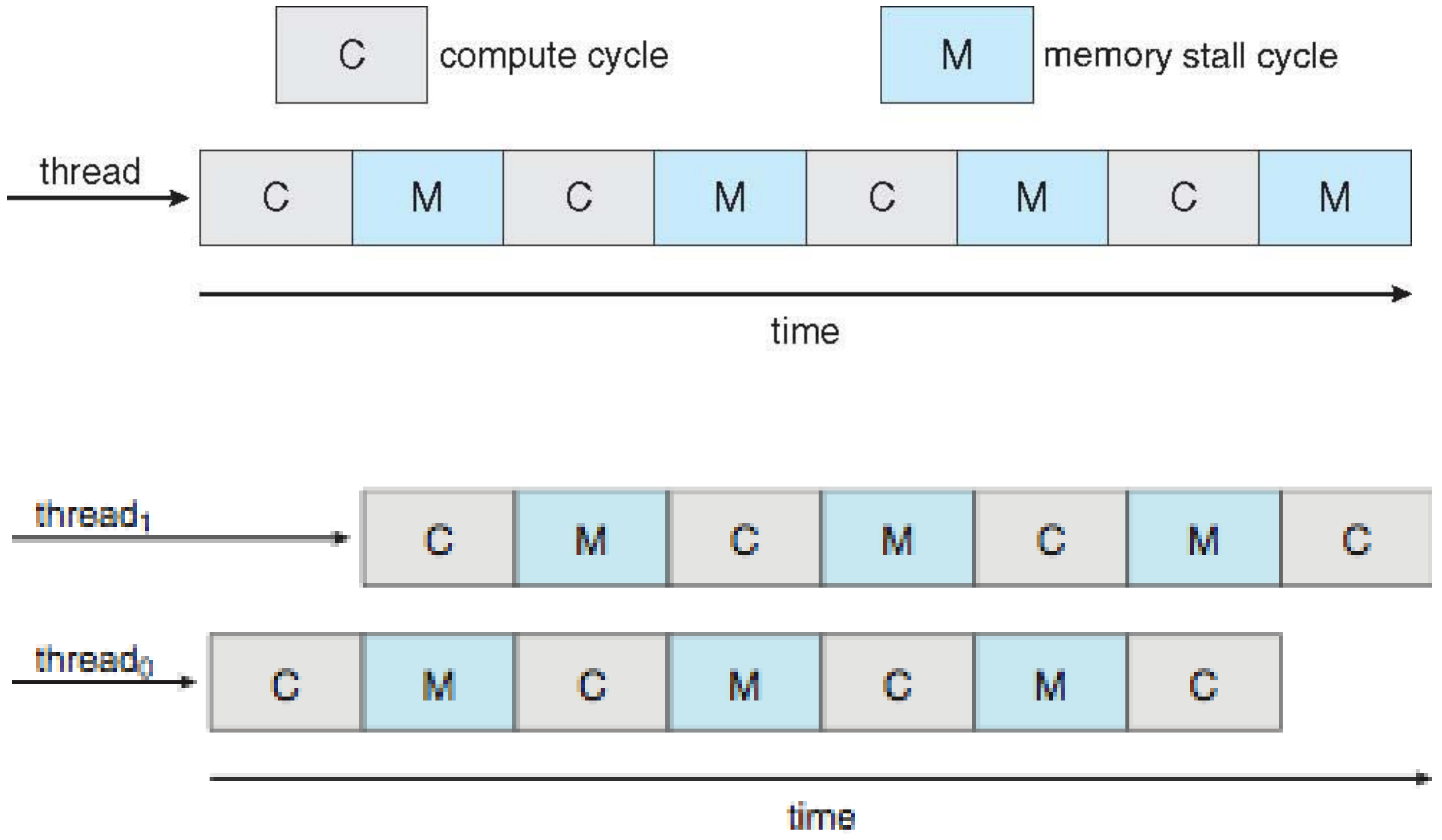
Multiple-Processor Scheduling – Load Balancing

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

Multicore Processors

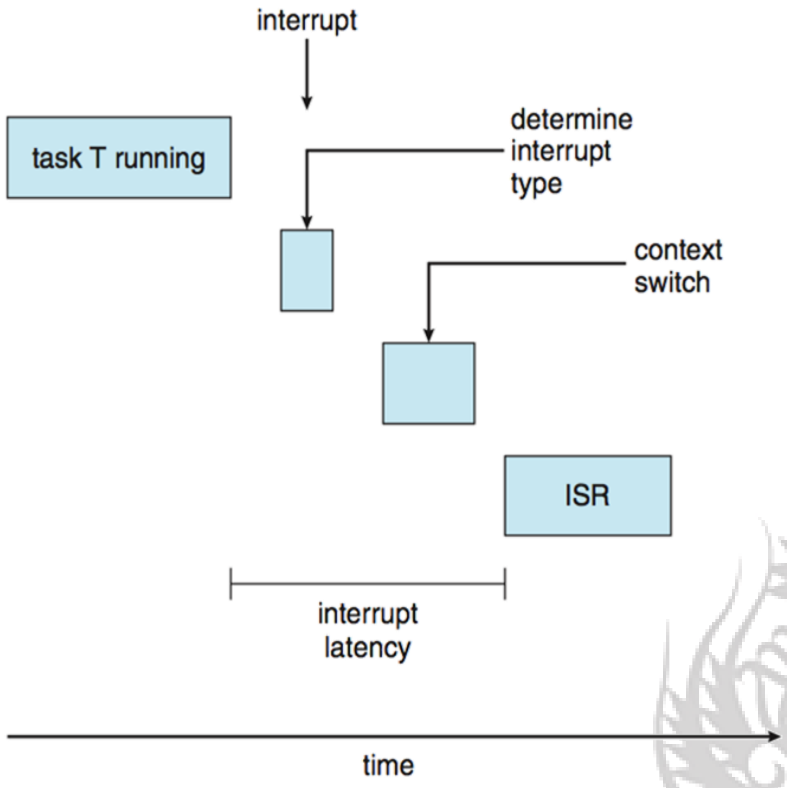
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Simultaneous multithreading (SMT) or H/W multithreading
    - E.g. Intel Hyper-threading
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System



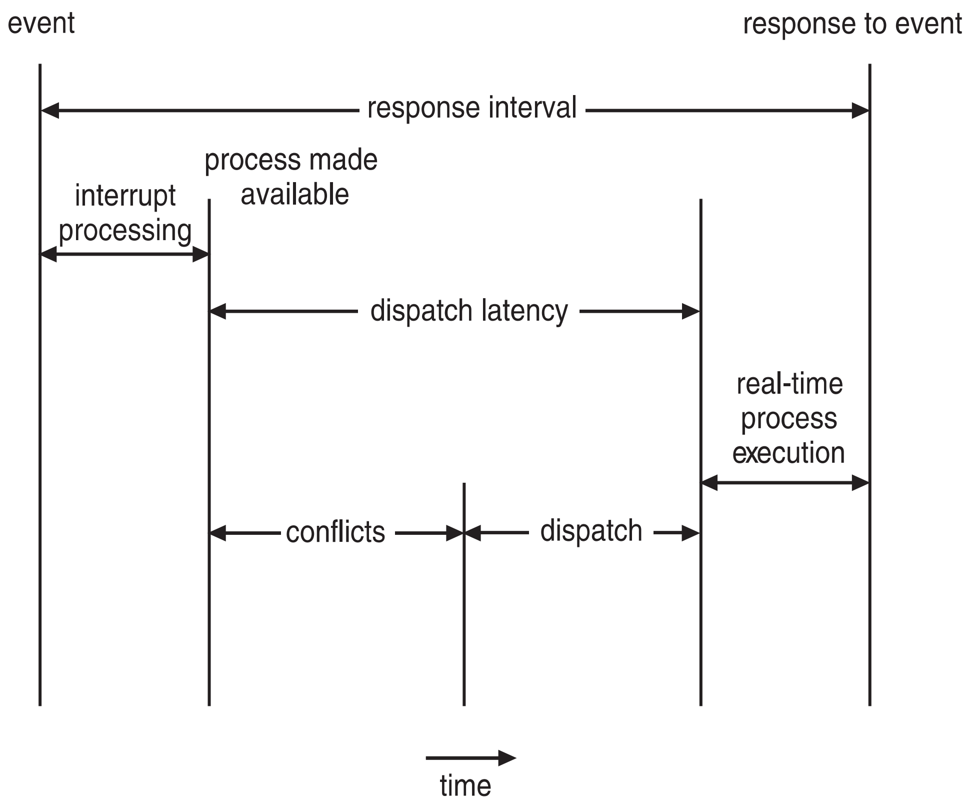
Real-Time CPU Scheduling

- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
  - e.g. streaming player
- **Hard real-time systems** – task must be serviced by its deadline
  - e.g. control of medical robot
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of rou tine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



Real-Time CPU Scheduling (Cont.)

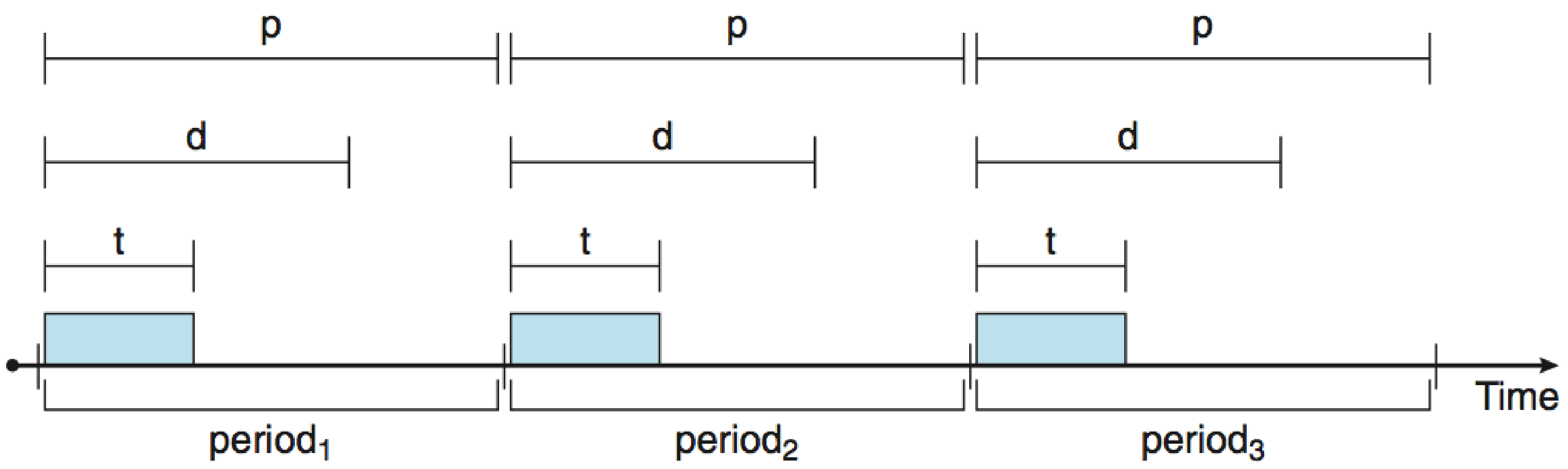
- Conflict phase of dispatch latency:
  - Preemption of any process running in kernel mode
  - Release by low-priority process of resources needed by high-priority processes



Priority-based Scheduling

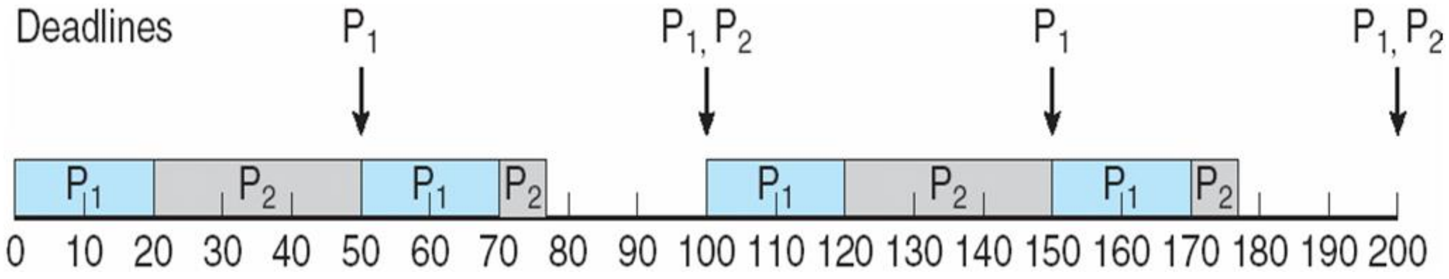
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
    - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: periodic ones require CPU at constant intervals
    - Has processing time t, deadline d, period p
    - $0 \leq t \leq d \leq p$
    - Rate of periodic task is 1/p

Priority-based Scheduling



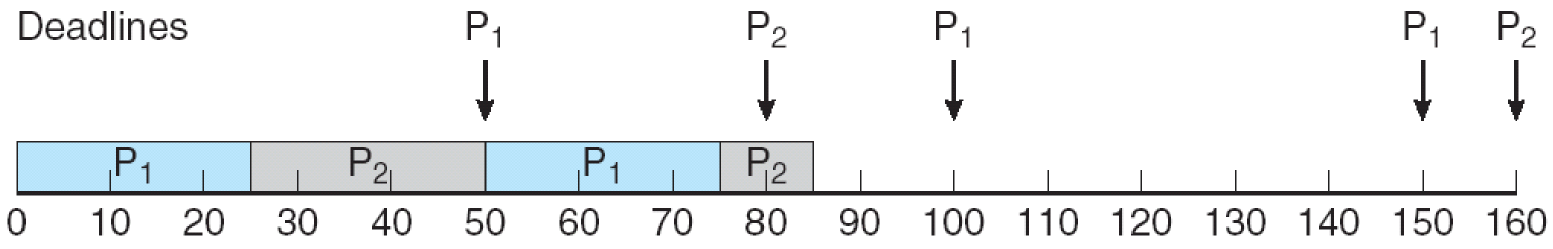
Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P1 is assigned a higher priority than P2.
  - P1: p and d = 50, t = 20 high priority (1/50)
  - P2: p and d = 100, t= 35 low priority (1/100)



Missed Deadlines with Rate Monotonic Scheduling

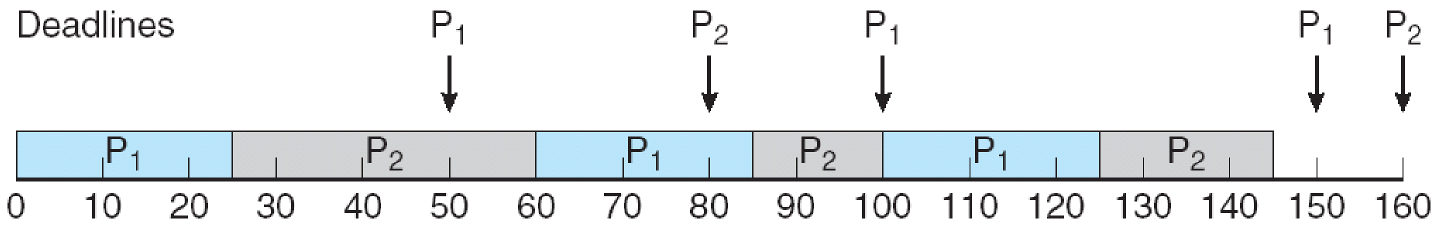
- Not suitable for hard real-time system
  - P1: p and d = 50, t = 25
  - P2: p and d = 80, t= 35



Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority

- P1: p and d = 50, t = 25
- P2: p and d = 80, t= 35



POSIX Real-Time Scheduling

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

POSIX Real-Time Scheduling API: Example

```
•   #include <pthread.h>
•   #include <stdio.h>
•   #define NUM_THREADS 5
•   int main(int argc, char *argv[])
•   {
•     int i, policy;
      pthread_t_tid[NUM_THREADS];
•     pthread_attr_t attr;
•     /* get the default attributes */
•     pthread_attr_init(&attr);
•     /* get the current scheduling policy */
      if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
•       fprintf(stderr, "Unable to get policy.\n");
•     else {
•       if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
•       else if (policy == SCHED_RR) printf("SCHED_RR\n");
•       else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
•     }
```

POSIX Real-Time Scheduling API: Example

- /* set the scheduling policy - FIFO, RR, or OTHER */
  if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
-   fprintf(stderr, "Unable to set policy.\n");
- /* 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)
  {
- /* do some work ... */
- pthread_exit(0);
- }

Proportional Share Scheduling

- T shares are allocated among all processes in the system
- An application receives N shares where N < T
- This ensures each application will receive N / T of the total processor time
- Admission control is required
  - To guarantee that a process receives its allocated shares of time
  - E.g.) if 70% of CPU utilizations are used, the system cannot accept a process with CPU requirement of more than 30%

Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
  - Scheduling entity: process -> virtual CPU
- Each guest doing its own scheduling
  - Not knowing it doesn’t own the CPUs
  - Can result in poor response time
  - Can effect clocks in guests
  - Can undo good scheduling algorithm efforts of guests

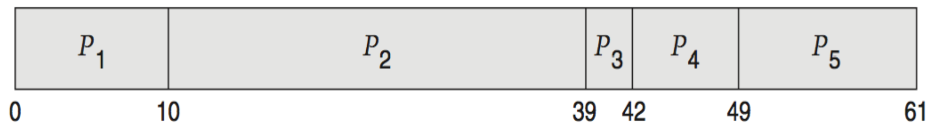
Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- Deterministic modeling
  - Type of analytic evaluation
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

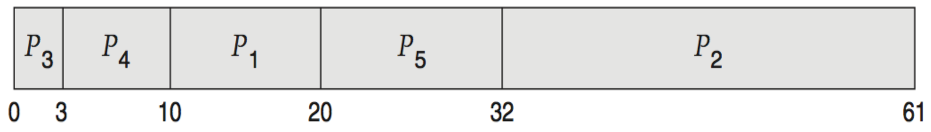
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

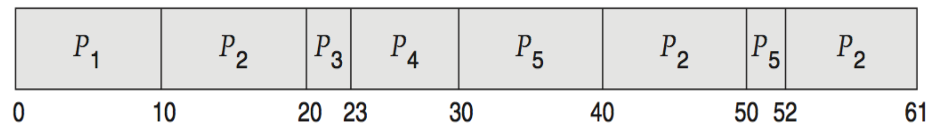
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
    - FCS is 28ms:



    - Non-preemptive SFJ is 13ms:



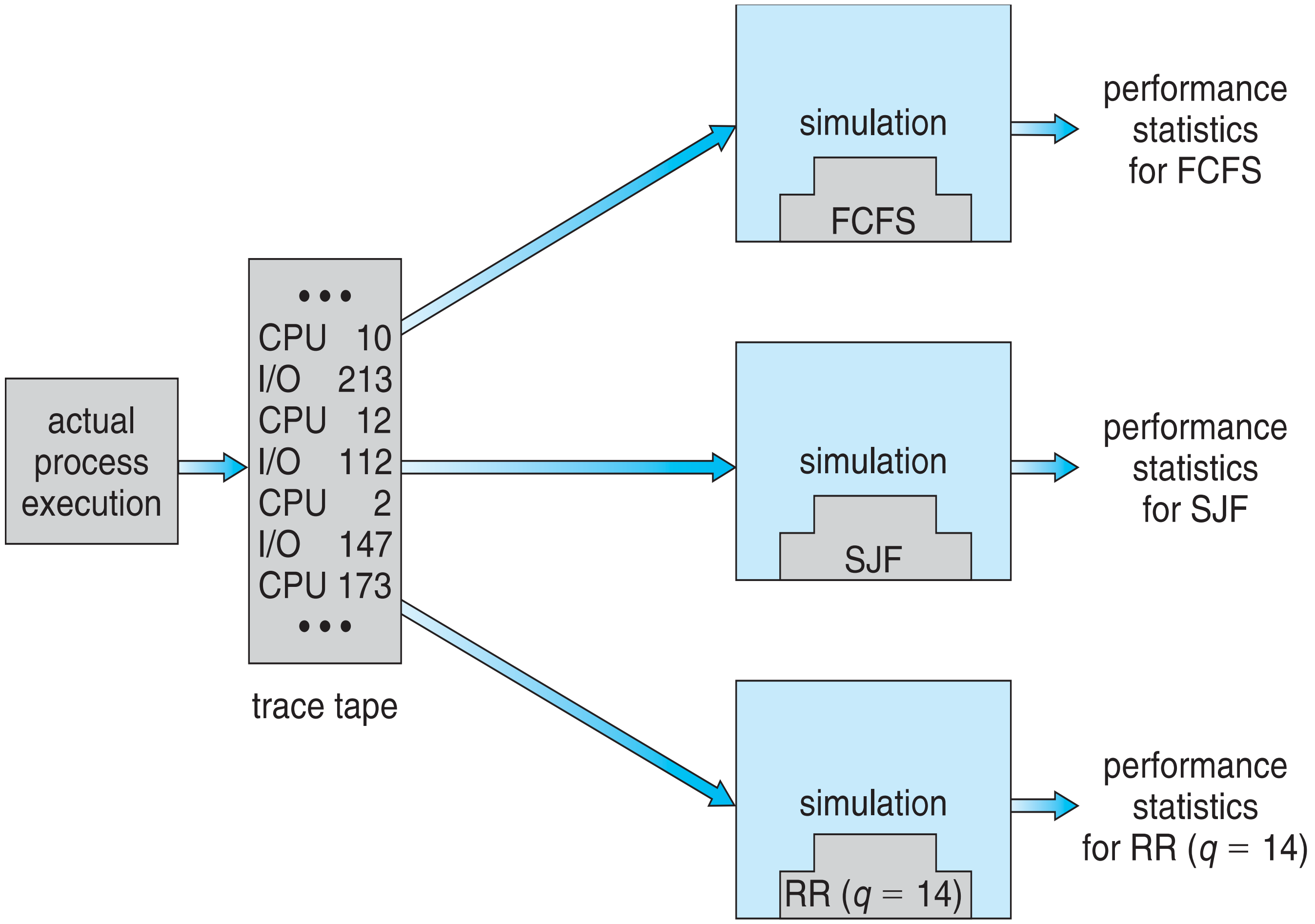
    - RR is 23ms:



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

Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling

Linux Scheduling Through Version 2.5

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

Linux Scheduling in Version 2.6.23 +

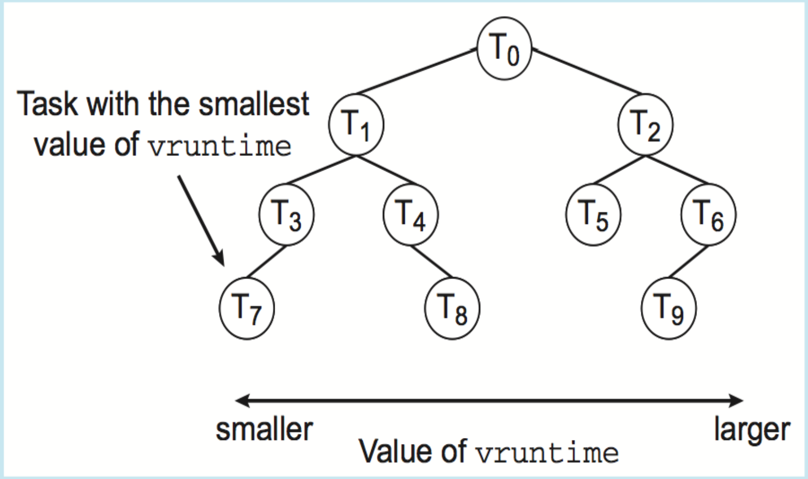
- Completely Fair Scheduler (CFS)
- Scheduling classes
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    - default
    - real-time

Linux Scheduling in Version 2.6.23 +

- 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
- 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 yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance

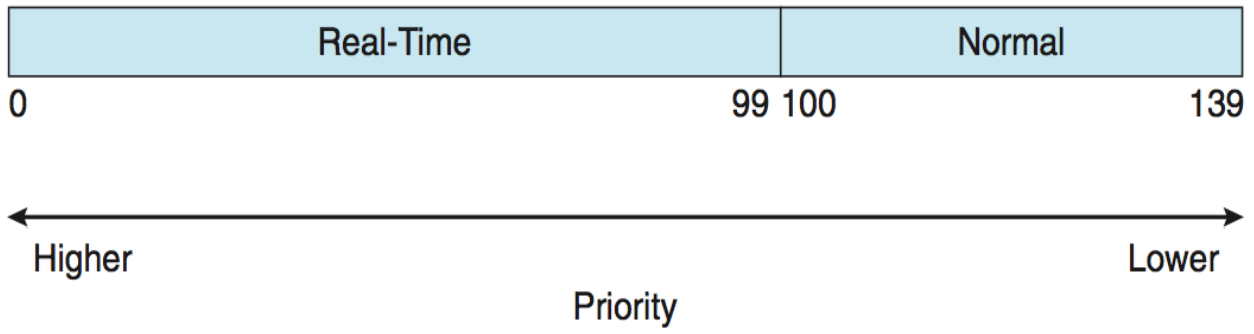
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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

Linux Scheduling (Cont.)

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



Windows Scheduling

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

Windows Priority Classes

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

Windows Priority Classes (Cont.)

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

Windows Priorities

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

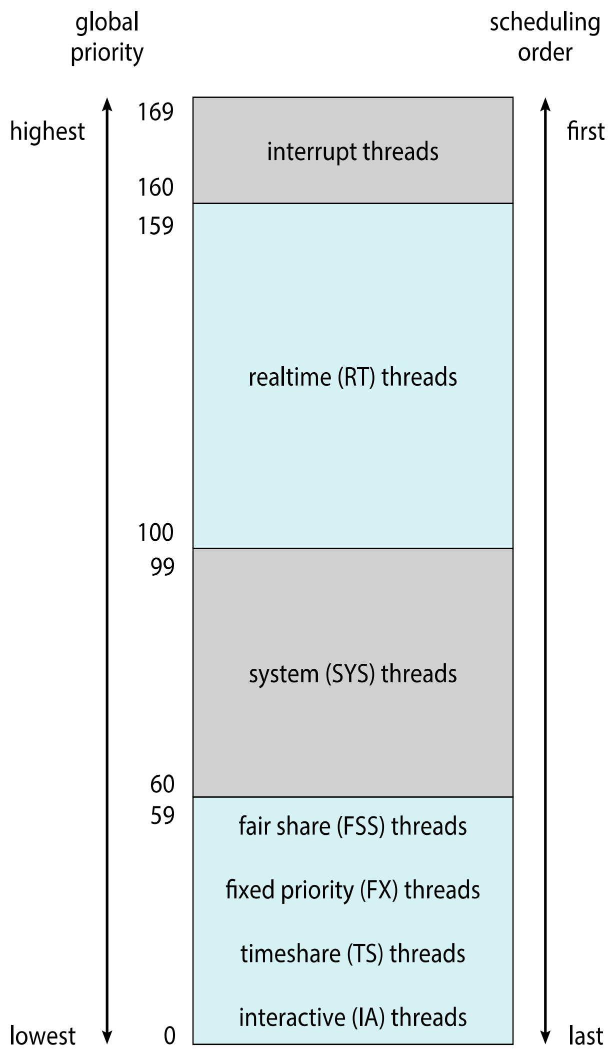
Solaris

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

Solaris Dispatch Table

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

Solaris Scheduling



Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR