

CPU Scheduling

CS 256/456
Department of Computer Science
University of Rochester

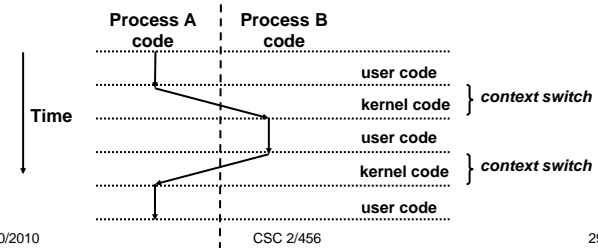
10/20/2010

CSC 2/456

28

Context Switching

- Processes are managed by a shared chunk of OS code called the *kernel*
 - Important: the kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via a *context switch*.



10/20/2010

CSC 2/456

29

Thread Scheduling: Transferring Context Blocks

Coroutines

transfer(other)

save all callee-saves registers on stack, including ra and fp

*current := sp

current := other

sp := *current

pop all callee-saves registers (including ra, but NOT sp!)

return (into different coroutine!)

10/20/2010

CSC 2/456

30

Uniprocessor Scheduling

- Use Ready List to reschedule voluntarily (cooperative threading)
 - reschedule:
 - t : cb := dequeue(ready_list)
 - transfer(t)
 - yield:
 - enqueue(ready_list, current)
 - reschedule
 - sleep_on(q):
 - enqueue(q, current)
 - reschedule

10/20/2010

CSC 2/456

31

Preemption

- Use timer interrupts or signals to trigger involuntary yields
- Protect scheduler data structures by disabling/reenabling prior to/after rescheduling

yield:

```

disable_signals
enqueue(ready_list, current)
reschedule
re-enable_signals
    
```

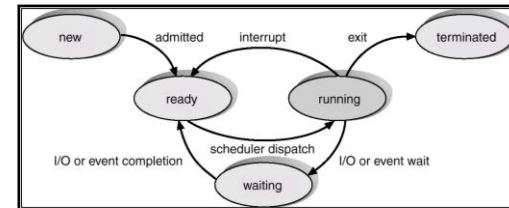
10/20/2010

CSC 2/456

32

Process State

- As a process executes, it changes *state*
 - **new**: The process is being created
 - **ready**: The process is waiting to be assigned to a process
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **terminated**: The process has finished execution



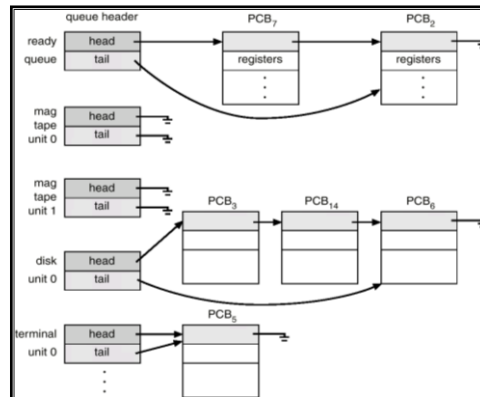
10/20/2010

CSC 2/456

33

Queues for PCBs

- Ready queue - set of all processes ready for execution.
- Device queues - set of processes waiting for an I/O device.
- Process migration between the various queues.



10/20/2010

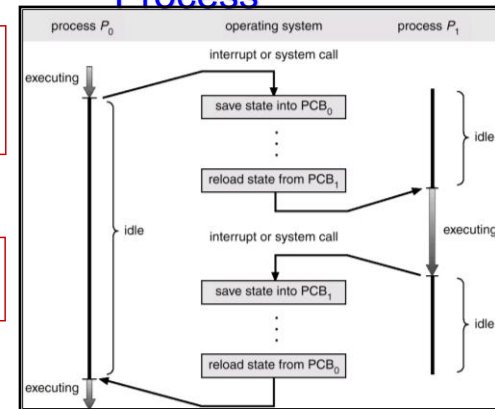
CSC 2/456

34

CPU Switch From Process to Process

When can the OS switch the CPU from one process to another?

Which one to switch to? - scheduling



10/20/2010

CSC 2/456

35

CPU Scheduling

- Selects from among the processes/threads that are ready to execute, and allocates the CPU to it
- CPU scheduling may take place at:
 1. Hardware interrupt/software exception
 2. System calls
- *Nonpreemptive*:
 - Scheduling only when the current process terminates or not able to run further
- *Preemptive*:
 - Scheduling can occur at any opportunity possible

10/20/2010

CSC 2/456

36

Scheduling Criteria

- Minimize turnaround time - amount of time to execute a particular process
- Maximize throughput - # of processes that complete their execution per time unit
- Maximize CPU utilization - the proportion of the CPU that is not idle
- Minimize response time - amount of time it takes from when a request was submitted until the first response is produced (interactivity)
- Fairness: avoid starvation

10/20/2010

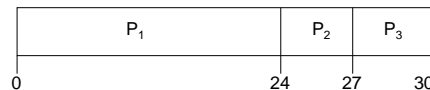
CSC 2/456

37

First-Come, First-Served (FCFS) Scheduling

Process	CPU 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 schedule is:



- Turnaround time for $P_1 = 24$; $P_2 = 27$; $P_3 = 30$
- Average turnaround time: $(24 + 27 + 30)/3 = 27$

10/20/2010

CSC 2/456

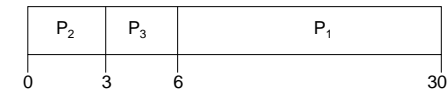
38

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2, P_3, P_1.$$

- The schedule is:



- Turnaround time for $P_1 = 30$; $P_2 = 3$; $P_3 = 6$
- Average turnaround time: $(30 + 3 + 6)/3 = 13$
- Much better than previous case.
- Short process delayed by long process: *Convoy effect*

10/20/2010

CSC 2/456

39

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its CPU time. Use these lengths to schedule the process with the shortest CPU time
- Two variations:
 - Non-preemptive - once CPU given to the process it cannot be taken away until it completes
 - preemptive - if a new process arrives with CPU time less than remaining time of current executing process, preempt
- Preemptive SJF is optimal - gives minimum average turnaround time for a given set of processes
- Problem:
 - don't know the process CPU time ahead of time

10/20/2010

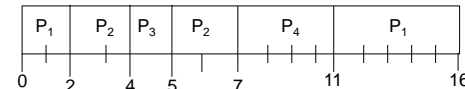
CSC 2/456

40

Example of Preemptive SJF

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

- SJF (preemptive)



- Average turnaround time = $(16 + 5 + 1 + 6)/4 = 7$

10/20/2010

CSC 2/456

41

Priority Scheduling

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

10/20/2010

CSC 2/456

42

Round Robin (RR)

- Each process gets a fixed unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units
- Performance
 - q small \Rightarrow fair, starvation-free, better interactivity
 - q large \Rightarrow FIFO
 - q must be large with respect to context switch cost, otherwise overhead is too high

10/20/2010

CSC 2/456

43

Cost of Context Switch

- Direct overhead of context switch
 - saving old contexts, restoring new contexts,
- Indirect overhead of context switch
 - caching, memory management overhead

10/20/2010

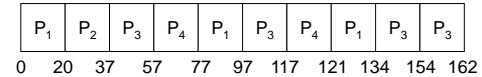
CSC 2/456

44

Example of RR with Quantum = 20

<u>Process</u>	<u>CPU Time</u>
P_1	53
P_2	17
P_3	68
P_4	24

- The schedule is:



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

10/20/2010

CSC 2/456

45

Multilevel Scheduling

- Ready tasks are partitioned into separate classes:
 - foreground (interactive)
 - background (batch)
- Each class has its own scheduling algorithm,
 - foreground - RR
 - background - FCFS
- Scheduling must be done between the classes.
 - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation
 - Time slice - each class gets a certain amount of CPU time which it can schedule amongst its processes; e.g.,
 - 80% to foreground in RR
 - 20% to background in FCFS

10/20/2010

CSC 2/456

46

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

10/20/2010

CSC 2/456

47

Example of Multilevel Feedback Queue

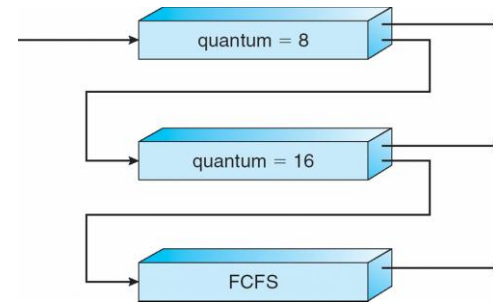
- Three queues:
 - Q_0 – RR with time quantum 8 milliseconds
 - Q_1 – RR time quantum 16 milliseconds
 - Q_2 – FCFS
- Scheduling
 - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
 - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 .

10/20/2010

CSC 2/456

48

Multilevel Feedback Queues



10/20/2010

CSC 2/456

49

Lottery Scheduling

- Give processes lottery tickets for various system resources
- Choose ticket at random and allow process holding the ticket to get the resource
- Hold a lottery at periodic intervals
- Properties
 - Chance of winning proportional to number of tickets held (highly responsive)
 - Cooperating processes may exchange tickets
 - Fair-share scheduling easily implemented by allocating tickets to users and dividing tickets among child processes

10/20/2010

CSC 2/456

50

Real-Time Scheduling

- Hard real-time systems – required to complete a critical task within a guaranteed amount of time
- Soft real-time computing – requires that critical processes receive priority over less fortunate ones
- EDF – Earliest Deadline First Scheduling

10/20/2010

CSC 2/456

51

Linux Task Scheduling

- Linux 2.5 and up uses a preemptive, priority-based algorithm with two separate priority ranges:
 - A time-sharing class/range for fair preemptive scheduling (nice value ranging from 100-140)
 - A real-time class that conforms to POSIX real-time standard (0-99)
- Numerically lower values indicate higher priority
- Higher-priority tasks get longer time quanta (200-10 ms)
- One runqueue per processor (logical or physical); load balancing phase to equally distribute tasks among runqueues
- Runqueue indexed by priority and contains two priority arrays - **active** and **expired**
- Choose task with highest priority on active array; switch active and expired arrays when active is empty
- Time-sharing tasks are assigned the nice value +/- 5

10/20/2010

CSC 2/456

52

CPU Scheduling on Multi-Processors

- Cache affinity
 - Keep a task on a particular processor as much as possible
- Resource contention
 - prevent resource-conflicting tasks from running simultaneously on sibling processors

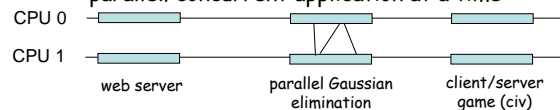
10/20/2010

CSC 2/456

53

Multiprocessor Scheduling

- Timesharing
 - similar to uni-processor scheduling - one queue of ready tasks (protected by synchronization), a task is dequeued and executed when a processor is available
- Space sharing
- cache affinity
 - affinity-based scheduling - try to run each process on the processor that it last ran on
- caching sharing and synchronization of parallel/concurrent applications
 - gang/cohort scheduling - utilize all CPUs for one parallel/concurrent application at a time



10/20/2010

CSC 2/456

54

Anderson et al. 1989 (IEEE TOCS)

- Raises issues of
 - Locality (per-processor data structures)
 - Granularity of scheduling tasks
 - Lock overhead
 - Tradeoff between throughput and latency
 - Large critical sections are good for best-case latency (low locking overhead) but bad for throughput (low parallelism)

10/20/2010

CSC 2/456

55

Performance Measures

- Latency
 - Cost of thread management under the best case assumption of no contention for locks
- Throughput
 - Rate at which threads can be created, started, and finished when there is contention

10/20/2010

CSC 2/456

56

Optimizations

- Allocate stacks lazily
- Store deallocated control blocks and stacks in free lists
- Create per-processor ready lists
- Create local free lists for locality
- Queue of idle processors (in addition to queue of waiting threads)

10/20/2010

CSC 2/456

57

Ready List Management

- Single lock for all data structures
- Multiple locks, one per data structure
- Local freelists for control blocks and stacks, single shared locked ready list
- Queue of idle processors with preallocated control block and stack waiting for work
- Local ready list per processor, each with its own lock

10/20/2010

CSC 2/456

58

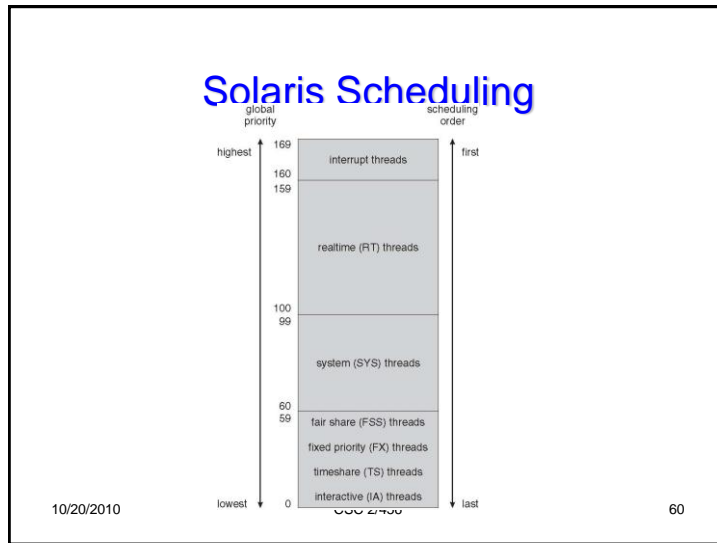
Solaris Dispatch Table

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

10/20/2010

CSC 2/456

59



- ### Multiprocessor Scheduling in Linux 2.6
- One ready task queue per processor
 - scheduling within a processor and its ready task queue is similar to single-processor scheduling
 - One task tends to stay in one queue
 - for cache affinity
 - Tasks move around when load is unbalanced
 - e.g., when the length of one queue is less than one quarter of the other
 - which one to pick?
 - No native support for gang/cohort scheduling or resource-contention-aware scheduling
- 10/20/2010 61

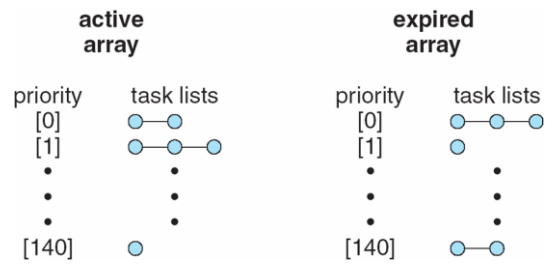
- ### Linux Scheduling
- Constant order $O(1)$ scheduling time
 - Two priority ranges: time-sharing and real-time
 - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- 10/20/2010 62

Priorities and Time-slice length

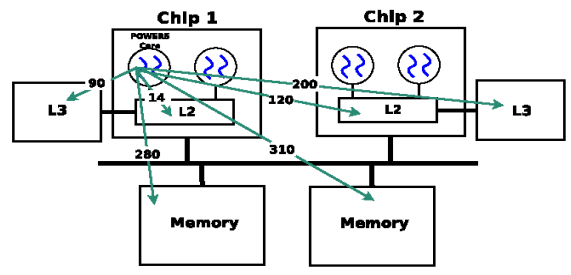
numeric priority	relative priority	time quantum
0	highest	200 ms
•		real-time tasks
•		
99		other tasks
100		
•		
•		10 ms
140	lowest	

10/20/2010 63

List of Tasks Indexed According to Priorities



SMP-CMP-SMT Multiprocessor

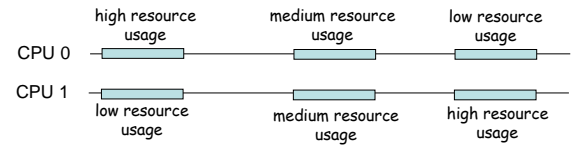


Resource Contention-Aware Scheduling I

- Hardware resource sharing/contention in multi-processors
 - SMP processors share memory bus bandwidths
 - Multi-core processors share L2 cache
 - SMT processors share a lot more stuff
- An example: on an SMP machine
 - a web server benchmark delivers around 6300 reqs/sec on one processor, but only around 9500 reqs/sec on an SMP with 4 processors
- Contention-reduction scheduling
 - co-scheduling tasks with complementary resource needs (a computation-heavy task and a memory access-heavy task)
 - In [Fedorova et al. USENIX2005], IPC is used to distinguish computation-heavy tasks from memory access-heavy tasks

Resource Contention-Aware Scheduling II

- What if contention on a resource is unavoidable?
- Two evils of contention
 - high contention ⇒ performance slowdown
 - fluctuating contention ⇒ uneven application progress over the same amount of time ⇒ poor fairness
- [Zhang et al. HotOS2007] Scheduling so that:
 - very high contention is avoided
 - the resource contention is kept stable



Disclaimer

- Parts of the lecture slides were derived from those by Kai Shen, Willy Zwaenepoel, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).

10/20/2010

CSC 2/456

68