Preemptive Scheduling

- Again in the context of CPU scheduling
- A thread or process is chosen to run
- It runs until either it yields
- Or the OS decides to interrupt it
- At which point some other process/thread runs
- Typically, the interrupted process/thread is restarted later

Implications of Forcing Preemption

- A process can be forced to yield at any time
  - If a higher priority process becomes ready
    - Perhaps as a result of an I/O completion interrupt
  - If running process's priority is lowered
    - Perhaps as a result of having run for too long
- Interrupted process might not be in a "clean" state
  - Which could complicate saving and restoring its state
- Enables enforced "fair share" scheduling
- Introduces gratuitous context switches
  - Not required by the dynamics of processes
- Creates potential resource sharing problems

Implementing Preemption

- Need a way to get control away from process
  - E.g., process makes a sys call, or clock interrupt
- Consult scheduler before returning to process
  - Has any ready process had its priority raised?
  - Has any process been awakened?
  - Has current process had its priority lowered?
- Scheduler finds highest priority ready process
  - If current process, return as usual
  - If not, yield on behalf of current process and switch to higher priority process

Clock Interrupts

- Modern processors contain a clock
- A peripheral device
  - With limited powers
- Can generate an interrupt at a fixed time interval
- Which temporarily halts any running process
- Good way to ensure that runaway process doesn't keep control forever
- Key technology for preemptive scheduling

Round Robin Scheduling Algorithm

- Goal - fair share scheduling
  - All processes offered equal shares of CPU and experience similar queue delays
- All processes are assigned a nominal time slice
  - Usually the same sized slice for all
- Each process is scheduled in turn
  - Runs until it blocks, or its time slice expires
  - Then put at the end of the process queue
- Then the next process is run
- Eventually, each process reaches front of queue

Properties of Round Robin Scheduling

- All processes get relatively quick chance to do some computation
  - At the cost of not finishing any process as quickly
  - A big win for interactive processes
- Far more context switches
  - Which can be expensive
- Runaway processes do relatively little harm
  - Only take $1/n^{th}$ of the overall cycles

Round Robin and I/O Interrupts

- Processes get halted by round robin scheduling if their time slice expires
- If they block for I/O (or anything else) on their own, the scheduler doesn't halt them
- Thus, some percentage of the time round robin acts no differently than FIFO
  - When I/O occurs in a process and it blocks

Round Robin Example

Assume a 50 msec time slice (or *quantum*)

Dispatch Order: 0, 1, 2, 3, 4, 0, 1, 2, . . .

Process	Length	1st	2nd	3d	4th	5th	6th	7th	8th	Finish	Switches
0	350	0	250	475	650	800	950	1050		1100	7
1	125	50	300	525						525	3
2	475	100	350	550	700	850	1000	1100	1250	1275	10
3	250	150	400	600	750	900				900	5
4	75	200	450							475	2
										1275	**27**

Average waiting time: 100 msec

First process completed: 475 msec

Comparing Example to Non-Preemptive Examples

- Context switches: 27 vs. 5 (for both FIFO and SJF)
  - Clearly more expensive
- First job completed: 475 msec vs.
  - 75 (shortest job first)
  - 350 (FIFO)
  - Clearly takes longer to complete some process
- Average waiting time: 100 msec vs.
  - 350 (shortest job first)
  - 595 (FIFO)
  - For first opportunity to compute
  - Clearly more responsive

Choosing a Time Slice

- Performance of a preemptive scheduler depends heavily on how long time slice is
- Long time slices avoid too many context switches
  - Which waste cycles
  - So better throughput and utilization
- Short time slices provide better response time to processes
- How to balance?

Costs of a Context Switch

- Entering the OS
  - Taking interrupt, saving registers, calling scheduler
- Cycles to choose who to run
  - The scheduler/dispatcher does work to choose
- Moving OS context to the new process
  - Switch stack, non-resident process description
- Switching process address spaces
  - Map-out old process, map-in new process
- Losing instruction and data caches
  - Greatly slowing down the next hundred instructions

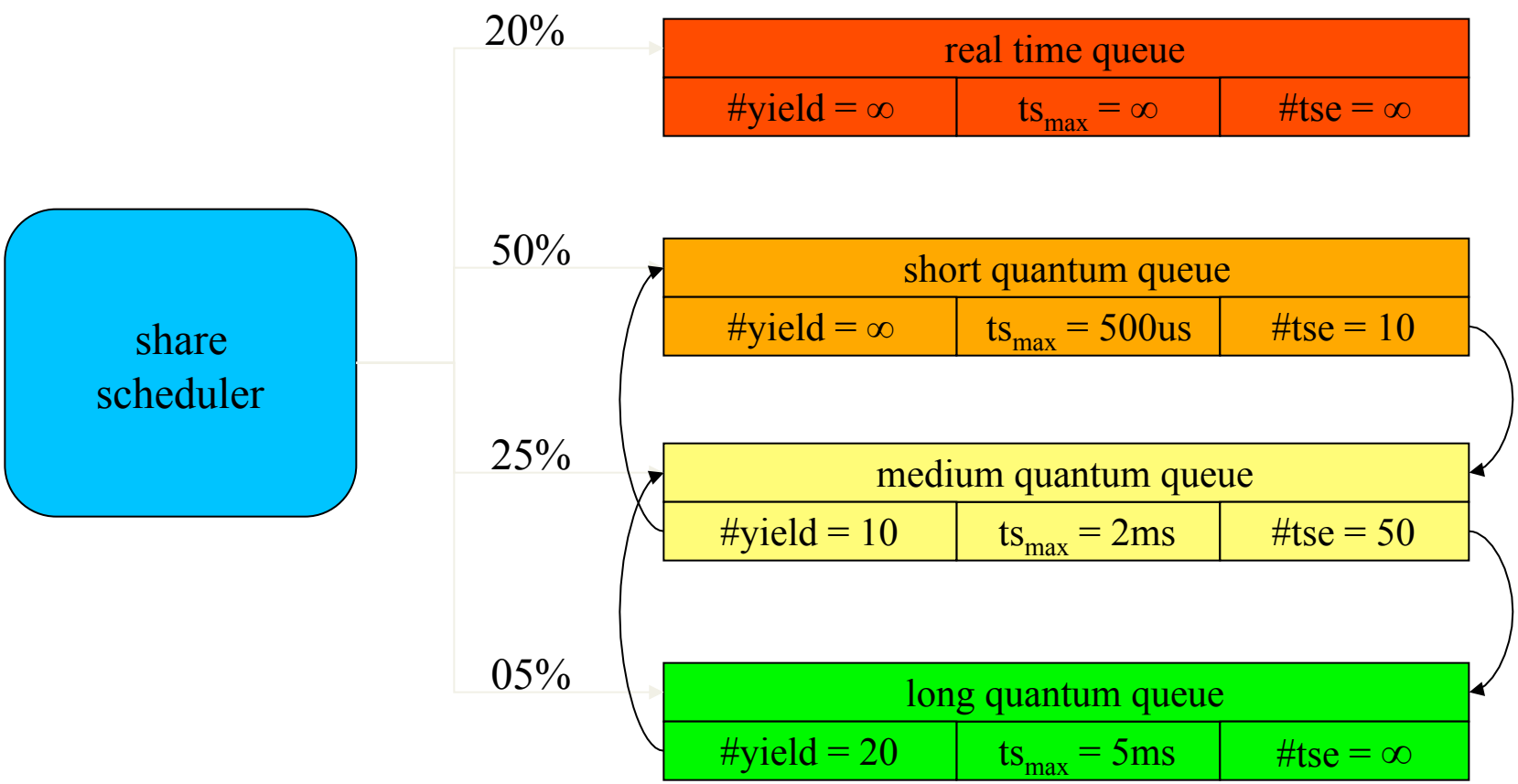
Multi-queue Scheduling

- One time slice length may not fit all processes
- Create multiple ready queues
  - Short quantum (foreground) tasks that finish quickly
    - Short but frequent time slices, optimize response time
  - Long quantum (background) tasks that run longer
    - Longer but infrequent time slices, minimize overhead
  - Different queues may get different shares of the CPU

How Do I Know What Queue To Put New Process Into?

- Start all processes in short quantum queue
  - Move downwards if too many time-slice ends
  - Move back upwards if too few time slice ends
  - Processes dynamically find the right queue
- If you also have real time tasks, you know what belongs there
  - Start them in real time queue and don't move them

Multiple Queue Scheduling



Priority Scheduling Algorithm

- Sometimes processes aren't all equally important
- We might want to preferentially run the more important processes first
- How would our scheduling algorithm work then?
- Assign each job a priority number
- Run according to priority number

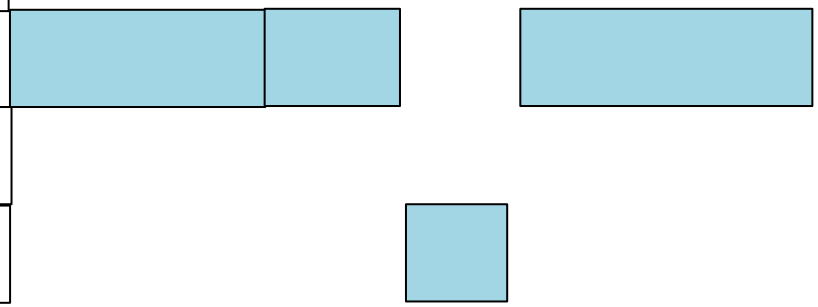
Priority and Preemption

- If non-preemptive, priority scheduling is just about ordering processes
- Much like shortest job first, but ordered by priority instead
- But what if scheduling is preemptive?
- In that case, when new process is created, it might preempt running process
  - If its priority is higher

Priority Scheduling Example

550 **Time**

Process	Priority	Length
0	10	350
1	30	125
2	40	475
3	20	250
4	50	75



Process 4 completes

So we go back to process 2

Process 3's priority is lower than running process

Process 4's priority is higher than running process

Problems With Priority Scheduling

- Possible starvation
- Can a low priority process ever run?
- If not, is that really the effect we wanted?
- May make more sense to adjust priorities
  - Processes that have run for a long time have priority temporarily lowered
  - Processes that have not been able to run have priority temporarily raised

Priority Scheduling in Linux

- Each process in Linux has a priority
  - Called a *nice* value
  - A soft priority describing share of CPU that a process should get
- Commands can be run to change process priorities
- Anyone can request lower priority for his processes
- Only privileged user can request higher