CS-534: Packet Switch Architecture
Spring 2005

Department of Computer Science

Exercise Set 12:
Output Scheduling for QoS

Assigned: Wed. 1 June 2005 (week 13) - Due on the Final Exam Day

12.1 Conservation Law for Work-Conserving Schedulers

Consider a work-conserving scheduler on an output port of a CIOQ switch with an internal speedup of 2. The queue(s) behind this scheduler (output-side queues) are filled at a rate up to 2 cells per time slot (given the speedup factor of 2), and they are emptied at a rate of 1 cell per time slot whenever they are non-empty. Assume that the cell arrivals are as follows:

> c01; -; b03, d03; -; -; a06; c07, d07; b08, c08; a09; c10; b11; a12, d12; -; -; c15; a16, b16; -; c18, d18; b19; a20; -; b22; -; d24; -; -; b27, d27; c28; a29; -; -;

In this list, each cell is identified by a letter and a number. The letter (a, b, c, or d) indicates the switch input that this cell came from. The number indicates the time slot during which the cell was delivered to the output-side queue(s) of the CIOQ switch. Thus, cells b03 (from input b) and d03 (from input d) traversed the crossbar and arrived at the queue(s) in front of our scheduler at the end of time slot 3; in time slots 4 and 5, no cells arrived through the crossbar, as indicated by the dashes; in time slot 6, the single cell a06 (from input a) was delivered to our queue(s).

We wish to compute the average cell delay through the above queue(s) and output scheduler, separately for the following four "flows": flow A consists of all cells originating from input a, flow B consists of those originating from input b, and similarly for flow C originating from c and flow D from d. A cell that is delivered to the output queue(s) in time slot *i* and is served (i.e. departs) during time slot *j* is said to have undergone a delay of *(j-(i+1))*. Note that the earliest possible departure time for this cell is time slot *i+1*, in which case the cell is consider to depart with zero delay. We will consider the floowing two kinds of scheduler:

i. FIFO Scheduler: there is a single output queue in front of the scheduler, and cells depart in strict FIFO order.
ii. Static Priority Scheduler: there are four queues in front of the scheduler, one for each of the flows A, B, C, and D, and the scheduler serves them in strict priority order, with queue A having the highest priority, and queue D having the lowest priority.

For each of the above two schedulers, list the cell departure time slots and corresponding delays, in detail, in the form: ts02: c01 (0); ts03: -; ts04: b03 (0); ts05: d03 (1); ts06: -; ts07: a06 (0); etc. As mentioned above, cell c01, which arrived in time slot 1, cannot depart earlier than the following time slot 2, and similarly cell b03 cannot depart before time slot 4; in both of these cases, the cell delay was 0. Cell d03 arrived in time slot 3 and departed in time slot 5, hence it had a delay of 1. Based on these listings for the two schedulers, answer the following questions:

a. What is the cumulative delay of all cells of flow A for each scheduler? Similarly for flows B, C, and D. What is the cumulative delay for all cells of all flows, for each scheduler? What do you observe?
b. What is the average rate of flow A during the above interval? (The average rate is the number of cells divided by the duration of the interval). Similarly for flows B, C, and D.
c. What is the average per-cell delay for flow A, for each scheduler? Similarly for flows B, C, and D.
d. Compute the average delay of all flows, where each flow's delay is weighed by its average rate, for each scheduler. What do you observe?

12.2 Memory-Access Cost of Linked-List Round-Robin Scheduling

In section 7.3 (last transparency), we saw that round-robin scheduling of a large number of flows (queues) can be implemented by maintaining a circular linked list of the eligible flows, and serving one segment from each, in turn. We wish to estimate the time-cost of this method, in terms of number of accesses (per time slot) to the memory that holds the per-flow pointers and --possibly-- the per-flow timestamps. We assume that there is a large memory (possibly off-chip) that contains an array of size equal to the number of flows (queues). This array is indexed by the flow ID, and each array entry contains the "nextPointer" of the circular linked list (the ID of the next flow on the list); in scheme (b) below, each array entry additionally contains a "timeStamp". We wish to count the number of accesses to this memory.

Let a time slot be the time to receive or transmit one segment on an external link of the switch. Assume that our round-robin scheduler is situated on the output side of a CIOQ switch with an internal speedup of 2. Thus, in each time slot, two segments (of possibly different flows) may arrive from the crossbar and be enqueued for transmission, and one segment may be served by dequeueing and transmitting its head segment. In the worst case, the two arriving segments will belong to previously-empty flows, causing these two flows to become eligible and thus be inserted into the circular list, and the departing segment will be the last segment in its queue, causing its flow to become ineligible and be removed from the circular list.

(a) First consider the case where insertions into the circular list are performed at a "fixed" position of the list, i.e. a position that does not move when flows are served (case (c) in the above transparency). Specificly, the scheduler maintains two pointers (flow ID's) into the list (these pointers are **not** located inside the memory to which we are counting accesses):

- *lastServed*: points to the flow just before the flow that should be served next; this is the last flow that was served and was not subsequently removed from the list;
- *insertHere*: points to the flow after which new insertions should be made; this pointer advances on each insertion, so that newly eligible flows are inserted in FIFO order, but it does **not** advance upon flow serving; this pointer must also advance in case it happens to point to the flow that is being removed from the list as a result of becoming ineligible (empty) after being served.

Write, in pseudocode form, the pointer manipulations that are necessary when (i) inserting a new flow into the list, (ii) serving a flow which remains eligible and in the list, or (iii) serving a flow which becomes ineligible and is removed from the list. Count the number of accesses, per time slot, to the above memory that contains the per-flow state, in the above worst case where two insertions and one serving and removal are necessary, per time slot.

(b) Next, consider the case where each array entry additionally contains the "timeStamp" of when this flow was last served; in this case, each array entry consists of two words, and separate memory accesses are needed for each (although these may be "dependent" (sequential) accesses, we will not consider such optimizations in this exercise). The scheduler maintains a "currentTime" clock (time-stamp), and an estimate "circScanTime" of the current duration of one circular scan around the list. Insertions occur at (after) the *lastServed* point, but the *lastServed* pointer is or is not advanced to point to the newly inserted flow according to the criterion discussed in class and described in the comments at the end of section 7.3.

Write, in pseudocode form, the timeStamp and pointer comparisons and manipulations that are necessary in this case, when performing operations (i), (ii), or (iii) above. Count, again, the number of accesses, per time slot, to the per-flow state memory, in the above worst case.

12.3 WRR with Variable-Size Packets using Next-Service-Time

---Optional Exercise: Read and fully understand this exercise; then, if you do not heve enough time to perform what is asks, you are allowed to only do part of it (or none at all?)---

In section 7.4 we saw a method to implement (relatively) smooth weighted round-robin (WRR) scheduling using, for each flow, a "next service (virtual) time" and a "service interval". This method can be easily adapted to variable-size packets, by interpreting the service interval value to be the **per-byte** or **per-word** or **per-segment** service interval. The adaptation is as follows: after serving a flow (queue) by transmitting its head packet, we reschedule this flow to be served again after a (virtual) time interval equal to the flow's (per-size-unit) service interval multiplied by the size of the packet that was just transmitted. In other words, the rate of serving the flow is inversely proportional to the size of the packets that the flow transmits.

In this exercise, we will assume, for simplicity, that the service interval is **per-segment**, and we will consider packet sizes to be integer multiples of segments. Consider that there exist three flows (queues):

- flow **A**: weight=**1**, service_interval=**30**;
- flow **B**: weight=**3**, service_interval=**10**;
- flow **C**: weight=**6**, service_interval= **5**;

Assume that flows A and B are "persistent", i.e. their queues are always non-empty, while flow C is originally inactive (empty queue), and becomes active at virtual time 200, and remains active thereafter. You are to simulate by hand the operation of the WRR scheduler. Your simulation starts at virtual time 0; queues A and B contain the packets listed below, at that time. Queue C is empty until virtual time 200, and thereafter fills up faster than it is being emptied, with the packets listed below:

- queue **A** at virtual_time=0: a1-3, a2-1, a3-2, a4-4, a5-2, a6-1, a7-2,....
- queue **B** at virtual_time=0: b1-1, b2-1, b3-5, b4-2, b5-8, b6-3, b7-9, b8-6, b9-7,....
- queue **C** after virtual_time=200: c1-4, c2-8, c3-1, c4-2, c5-7, c6-8, c7-5, c8-8, c9-4,....

In the above lists, queue heads are on the left. Each packet is named by one letter and two numbers. The letter identifies the flow (queue) that the packet belongs to. The first number is the (sequential) packet ID inside its flow. The second number is the size of the packet, measured in segments. Thus, packet "b6-3" is the sixth packet of flow B, and its size is 3 segments; when naming individual segments of each packet, name them like this: b6.1, b6.2, and b6.3.

You are to simulate, by hand, the operation of the WRR scheduler, starting at real time slot 1 and virtual time 0, and stopping when virtual time reaches 400. Real time advances in time slots; one time slot is the time it takes the sceduler to serve (transmit) one segment. For each (real) time slot of your simulation interval, list the following:

- *the time slot number;*
  (sequentially counting 1, 2, 3,...).
- *the segment being served (transmitted) during that time slot;*
  Just prior to time slot 1, the next-service-time of both flows A and B was 0. When multiple flows have identical next-service-time's, give priority to A over the others, or to B over C; thus, during time slot 1, packet a1-3 starts being served, so the segment being served is a1.1. The server always serves *entire packets*, never interleaving their segments with segments of other packets, so once packet a1-3 has started being served, we will continue by serving segment a1.2 in time slot 2, and segment a1.3 in time slot 3; after that, the scheduler needs to choose the next flow to be served.
- *flow A's next_service_time at the end of this time slot;*
  The next_service_time of the flow being served is incremented by its service_interval during each time slot when the flow receives service. The number of time slots that it takes to serve a packet is equal to the packet size in number of segments; hence, at the end of serving the entire packet, the next_service_time of the flow will have been incremented by the flow's service_interval multiplied by the packet size, which is the desired behavior. Thus, for flow A,

at the end of time slot 1 its next_service_time will be 30, at the end of time slot 2 it will be 60, and at the end of time slot 3 it will be 90.

- *flow B's next_service_time at the end of this time slot;*
  The next_service_time of an active flow that is not being served stays constant; thus, for flow B, at the end of time slot 1 its next_service_time will still be 0, and the same will hold at the end of time slots 2 and 3.
- *flow C's next_service_time at the end of this time slot;*
  During the first several time slots, flow C is ineligible for service (its queue is empty), and thus you may list its next_service_time as N/A (not applicable). Alternatively, given the assumption that flow C will become eligible at virtual time 200, you may list its next_service_time to be 200; this will have the effect of first serving flow C when the next_service_time's of the other flows have just reached or exceeded 200.

Stop your simulation when the next_service_time's of all flows exceed 400, and answer the following questions:

a. How long (how many real time slots) did it take for virtual time to advance from 0 to 200? "Virtual time" is the minimum of the next_service_time's of all flows. How long did it then take for virtual time to advance from 200 to 400? Why do these two real time intervals differ, even though the corresponding virtual time intervals are equal?
b. At what average rate does virtual time advance during each of the two above intervals? The desired figure is the quotient of 200 over the number of real time slots in each interval, and it is measured in "virtual time units per real time slot". How are these two rates connected to the sum of the weights of all active flows during each of the two above intervals?
c. How many segments of service did flow A receive during each of the two above intervals, and how many segments of service did flow B receive, correspondingly? Is their ratio approximately equal to the ratio of A's weight to B's weight?
d. How many segments of service did flow C receive during the second of the two above intervals? When comparing this number to the service received by A or B in the same interval, are the ratios approximately equal to the corresponding ratios of weights?

Up to the Home Page of CS-534

Last updated: 1 June 2005, by M. Katevenis.