CS-534: Packet Switch Architecture
Spring 2005

Department of Computer Science

Exercise Set 4: Linked-List Queue Management

Assigned: Thu. 24 March 2005 (week 5) - Due: Wed. 30 March 2005 (week 6)

4.1 Pointer-Access Cost of Enqueue and Dequeue Operations

In section 3.2.2 (4th transparency), transparency, we saw a table of enqueue and dequeue latencies and throughput, for various degrees of memory access parallelism, i.e. different numbers of separate memory blocks, on-chip or off-chip. This table assumed that all memory accesses shown on the previous two transparencies --both normal ("basic") and exceptional-- have to be performed during each and every enqueue or dequeue operation. This is not necessarily true, especially for the more "serial" implementations (fewer memories available, more cycles per operation): the first access will often show whether we are in the normal or in the exceptional case, and the next accesses can then be tailored accordingly.

Consider the case where all pointers are stored in a single, 1-port, off-chip memory (first line of the table). More precisely, the width of this memory is either the width of one pointer (one segment address), or that width plus 1 bit; in the latter case, we can store one head or tail pointer plus one *Empty Flag* in one word of this available memory. (Note: an alternative scheme to storing both an empty flag and a pointer in a word is to only store the pointer while reserving one specific pointer value (e.g. "NULL") to denote an empty queue; for the purposes of this exercise, the two schemes are identical). This single memory contains all next-pointers (NxtPtr), all head pointers (Hd) of all queues, all tail pointers (Tl) of all queues, and all empty flags (in case such flags are used) of all queues. Each access to this single memory is performed to a single address (it is not possible to access in parallel the pointer in word A and the empty flag in another word B), and the access either reads the whole addressed word, or writes the whole addressed word (pointer plus flag, if a flag exists).

Count the number of accesses to this memory that are necessary in order to perform each of the following operations in the cases listed:

- **enqN** (Enqueue Normal): append a given *new* segment at the tail of a given queue *qID*. You don't know a priori whether the queue was empty or not, so you have to check for that, but assume that in this case the queue was **not** empty.
- **enqE** (Enqueue Exceptional): append a given *new* segment at the tail of a given queue *qID*. You don't know a priori whether the queue was empty or not, so you have to check for that, and assume that in this case the queue **was** empty.
- **deqN** (Dequeue Normal): remove the segment at the head of a given queue *qID*, and return the address of that segment. We know that the queue was not empty (the scheduler does not issue illegal or dummy operations), so you do not have to check for that. However, we do not know a priori whether the queue will become empty after we remove the head segment, so you have to check for that, but assume that in this case the queue does **not** become empty.
- **deqE** (Dequeue Exceptional): remove the segment at the head of a given queue *qID*, and return the address of that segment. We know that the queue was not empty so you do not have to check for that. However, we do not know a priori whether the queue will become empty after we remove the head segment, so you have to check for that, and assume that in this case the queue **does** become empty.

Determine the above 4 counts of memory accesses separately in each of the following 3 queue implementation schemes:

- **Empty with Head**: the queue-empty flag is stored in the same word where the queue head pointer is stored.
- **Empty with Tail**: the queue-empty flag is stored in the same word where the queue tail pointer is stored.
- **Empty with Both**: there are *two identical copies* of the empty flag for each queue; one of them is stored in the same word with the queue head pointer, and the other is stored in the same word with the queue tail pointer.

The counts of memory accesses are important because they provide an upper bound for the achievable rate of queue operations (memory access rate divided by number of accesses per queue operation). Assume that the operations performed on our queues consist of alternating enqueues and dequeues (1 enq, then 1 deq, then 1 enq, then 1 deq, etc).

- Which of the 3 queue implementation schemes (empty with head, empty with tail, empty with both) yield(s) the highest **worst-case** operation rate? The worst case is the case where all operations on the queues are of the worst of the 4 possible combinations: enqN-deqN, enqN-deqE, enqE-deqN, enqE-deqE.
- Which of the 3 queue implementation schemes (empty with head, empty with tail, empty with both) yield(s) the highest **average** operation rate? Assume that 75% of the enqueues are normal and 25% are exceptional, and similarly 75% of the dequeues are normal and 25% are exceptional. (Note that exceptional cases are not that rare: higher-priority queues will usually be empty, and, when per-flow queueing is used, flows that are bottlenecked upstream of this link will have usually-empty queues).

4.2 Queue Operation Latency and Throughput

The same table of section 3.2.2 (4th transparency) considered in the previous exercise assumed fall-through off-chip SRAM, and that an off-chip dependent access (address of second access = read data of first access) can occur, at the earliest, two cycles after the access it depends on.

Assume now that the off-chip SRAM chip(s) are synchronous (pipelined), QDR or ZBT SRAM's, like the ones seen in class (section section 2.3). In that case, dependent off-chip accesses can occur, at the earliest, three cycles after one another: if address A1 is valid on the address bus on clock edge number 1, read data D(A1) will be valid on the data bus on clock edge number 3; assuming that the controller chip is fast enough, it can then drive these data D(A1) back on the address bus during the immediately succeding clock cycle --the clock cycle between edges 3 and 4-- so that, after the drive delay, they are valid and stable on clock edge number 4.

Under this new assumption, recalculate all the latency values for the enqueue and dequeue operation in most of the system configurations shown on that 4th transparency, as described below. To be precise, use the following definition of latency: the latency of an operation is the time, in clock cycles, between initiations of two successive similar operations, assuming that no operation overlap occurs, i.e. assuming that the first memory access for the second operation occurs after the last memory access for the first operation.

For each operation (enqueue or dequeue) and for each of the system configurations below, show the schedule of memory access, i.e. which memory(ies) are accessed in each clock cycle, at which address. For the QDR/ZBT off-chip SRAM's, the cycle during which you show the access corresponds to the cycle during which the controller chip drives the address on the address bus; this address is valid on the address bus on the clock edge that terminates this clock cycle. If you do not have the time to derive this schedule of memory access for all of the system configurations shown on that 4th transparency, do it at least for the following configurations: *(i)* all of Head, Tail, and NxtPtr in a single, 1-port, off-chip memory; *(ii)* like case *(i)*, but with the Empty flags in an on-chip memory; *(iii)* Empty, Head, and Tail in 1-port, on-chip memories, and NxtPtr in 1-port, off-chip memory.

Up to the Home Page of CS-534