1. Consider a switch in which each output port has two buffers, one for low priority packets and one for high priority packets. Show how such a dual queue system can be modeled using a two-dimensional finite state model. Assume that during each time step, at most one of the following events occurs: (1) a high priority packet arrives, (2) a low priority packet arrives or (3) a packet leaves. The probabilities of these three mutually exclusive events are $\lambda_1$, $\lambda_2$, and $\mu$ respectively. Draw a diagram for a finite state model in which there is space for a total of three packets. Assume that an arriving high priority packet is discarded only if when it arrives, all the available storage space is occupied by other high priority packets.

The finite state model is shown below. The state $(i,j)$ corresponds to the situation where there are $i$ low priority packets and $j$ high priority packets.

![Diagram of the finite state model](diagram.png)

Write down the steady-state balance equations for your model and describe in words, a method for solving them.

The balance equations appear below. Any nine of these equations, plus the equation $\sum_{i,j} \pi(i,j) = 1$ forms a set of 10 independent linear equations in the ten steady-state probabilities. They can be solved using Gaussian elimination, or any other general method for solving linear equations.
\[\pi(0,0)(\lambda_1 + \lambda_2) = \pi(0,1)\mu + \pi(1,0)\mu\]
\[\pi(0,1)(\lambda_1 + \lambda_2 + \mu) = \pi(0,0)\lambda_i + \pi(0,2)\mu\]
\[\pi(0,2)(\lambda_1 + \lambda_2 + \mu) = \pi(0,1)\lambda_i + \pi(0,3)\mu\]
\[\pi(0,3)\mu = \pi(0,2)\lambda_i + \pi(1,2)\lambda_i\]
\[\pi(1,0)(\lambda_1 + \lambda_2 + \mu) = \pi(0,0)\lambda_i + \pi(1,1)\mu + \pi(2,0)\mu\]
\[\pi(1,1)(\lambda_1 + \lambda_2 + \mu) = \pi(1,0)\lambda_i + \pi(0,1)\lambda_i + \pi(1,2)\mu\]
\[\pi(1,2)(\lambda_1 + \mu) = \pi(0,2)\lambda_i + \pi(1,1)\lambda_i + \pi(2,1)\lambda_i\]
\[\pi(2,0)(\lambda_1 + \lambda_2 + \mu) = \pi(1,0)\lambda_i + \pi(2,1)\mu + \pi(3,0)\mu\]
\[\pi(2,1)(\lambda_1 + \mu) = \pi(1,0)\lambda_i + \pi(2,0)\lambda_i + \pi(3,0)\lambda_i\]
\[\pi(3,0)(\lambda_1 + \mu) = \pi(2,0)\lambda_i\]

Give an expression for the probability that a high priority packet is discarded. Give an expression for the probability that a low priority packet is discarded.

A high priority packet is discarded, if when it arrives, the high priority queue contains three packets. This occurs with probability \(\pi(0,3)\). A low priority packet is discarded, if when any packet arrives, there is no space for the new packet and either there is a packet in the low priority queue, or the arriving packet is a low priority packet. This occurs with probability \((\pi(3,0)+\pi(2,1)+\pi(1,2)) + \pi(0,3)\lambda_2/(\lambda_1+\lambda_2))\).

2. The diagram below shows a set of four queues, each containing several packets. The letters on the packets are just labels; the numbers on the packets indicate their lengths. The numbers in the boxes at the right indicate the number of “credits” available for each queue at time zero, assuming a Deficit Round Robin (DRR) packet scheduler. Assume that each queue gets an allocation of 6 credits on each pass through the queue. If the packet scheduler makes one complete pass through the set of queues, starting with the top queue, which packets get sent on the link, and in what order are they sent. How many credits does each queue have after the pass completes?

<table>
<thead>
<tr>
<th>B,5</th>
<th>A,3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>L,2</td>
<td>K,2</td>
<td>F,3</td>
</tr>
<tr>
<td>M,2</td>
<td>H,6</td>
<td>G,4</td>
</tr>
<tr>
<td>P,2</td>
<td>N,2</td>
<td>J,2</td>
</tr>
</tbody>
</table>

Packets A, C, D, E, F, G, I, J, N, P get sent, in that order. The number of credits at the end of the pass are 3, 1, 3 and 1 (from top to bottom).
3. The diagram below shows a set of four queues, each containing several packets. The letters on the packets are just labels; the first number on each packet indicates its length and the second indicates its virtual start time in an SCFQ packet scheduler. Assuming that all queues have a weight of $1/4$, which packet gets selected for transmission next? What is the virtual time value, following this selection? At what value of real-time does the packet complete, assuming that the initial value is 8? Suppose that right after this packet begins transmission on the link, we get two new packet arrivals. The first packet, $Q$, gets placed in the first queue, while the second packet, $R$, gets placed in a queue that has been empty, ever since the last time the link was idle. What start time is assigned to each of these packets? For the original set of packets, list the sequence in which they get selected for transmission on the link and show the virtual time after each selection, and the real time at which they complete.

![Diagram of queues](image)

The first packet to be selected is $I$, since its finish time is $3+2/25=11$, which is the smallest of the finish times among the packets waiting to be sent. After $I$ is selected, the virtual time becomes 11, and on completion, the real time is 10. Packet $Q$ is assigned a start time of 37 and packet $R$ is assigned a start time of 11. The total sequence of packet transmissions, together with the virtual time following the selection and the real-time when each packet completes is $I-11-10$, $C-15-12$, $A-17-15$, $D-19-16$, $J-19-18$, $E-23-19$, $G-25-23$, $N-27-25$, $P-35-27$, $F-35-30$, $B-37-35$, $K-43-37$, $H-49-43$, $L-51-45$, $M-57-47$.

4. Repeat the previous problem, assuming that the packet scheduler uses the Worst-Case, Weighted Fair Queueing algorithm. Assume that the initial value of the virtual time variable is 6. Which queues are eligible for selection, initially?

Initially, the top and bottom queues are the only ones eligible. After this point, all the non-empty queues remain eligible. The first packet to be selected is $I$, since its finish time is $3+2/25=11$, which is smaller than the finish time of packet $A$, which is the only other one that is eligible for consideration at this point. After $I$ is selected, the virtual time becomes 11, and on completion, the real time is 10. Packet $Q$ is assigned a start time of 37 and packet $R$ is assigned a start time of 11. The total sequence of packet transmissions, together with the virtual time following the selection is $I-11-10$, $C-15-12$, $A-17-15$, $D-19-16$, $J-19-18$, $E-23-19$, $G-25-23$, $N-27-25$, $P-35-27$, $F-35-30$, $B-37-35$, $K-43-37$, $H-49-43$, $L-51-45$, $M-57-47$.

5. (50 points) Write a program to simulate a link with per flow queues and a weighted Deficit Round Robin packet scheduler. Your program should read an input file that specifies the packet arrivals. The first line of the file will consist of an integer $n$ that specifies the number of queues followed by $n$ queue “weights” that specify the relative fraction of the link capacity assigned to each of the $n$ queues. Each subsequent line will have four numbers. The
first specifies the arrival time of a packet (where the time unit is the time it takes to send one byte on the link), the second specifies the queue the packet should be placed in, the third specifies the length of the packet (in bytes) and the fourth is a sequence number. The simulation should terminate when all packets specified in the input file have been sent on the link. Assume that there is no limit on the available buffer space and that the maximum packet size is 1000 bytes.

Your program should have three different output modes. In verbose mode it should generate a line of output, every time that a packet is sent on the link. Each line should contain the time at which the packet starts transmission on the link, the time the packet arrived, the queue it came from, its length and its sequence number. In trace mode, your program should periodically output the current time, the length of each queue (in bytes) and the fraction of the link capacity used by each queue during the preceding period. In terse mode, your program should output nothing until the end. Then for each of the queues, it should output the queue weight, the average length of the queue over the simulation run, the maximum length, the minimum and maximum delay for its packets, the total length of all of its packets and the fraction of the total simulation time used to forward its packets on the link.

Write a second version of your program that implements an SCFQ scheduler instead of a weighted DRR scheduler. You may use the dheap data structure, which you will find on the course web site to implement the SCFQ scheduler.

Run both of your programs on the two files packets1 and packets2 that you will find on the web site. For packets1, turn in the printouts from all three output modes. For the trace mode, use a time period equal to 5% of the total simulation time. For packets2, use the results from the trace mode to generate time history charts of the queue length and the fraction of the link capacity used by each of the queues. Display the results cumulatively – that is, have one curve for the length of queue 1, a second curve that shows the sum of the lengths of the first two queues, etc. Choose a time period no larger than .5% of the simulation time (so there will be at least 200 data points per curve in your time history plot). Turn in these plots, together with the terse mode printout.

The program that simulates drr is shown below. It consists of a main program, which reads the input packets, a class that does the drr simulation and a second class that implements a generic data structure representing the set of packet queues.

```plaintext
// usage:
// drrSim printmode tp
//
// DrrSim simulates a link with a DRR packet scheduler. The printmode argument is one of "verbose", "trace" and "terse". The tp argument is an integer that determines the time period between print outs when in trace mode.
//
// Input data is read from stdin. The first line consists of an integer n that specifies the number of queues followed by n queue "weights" that specify the relative fraction of the link capacity assigned to each of the n queues.
// Each subsequent line has four numbers. The first specifies the arrival time of a packet (where the time unit is the time it takes to send one bit on the link), the second specifies the queue the packet should be placed in, the third specifies the length of the packet (in bytes) and the fourth is a sequence number.
```
// fourth is a sequence number. The simulation should terminate
// when all packets specified in the input file have been sent
// on the link. The buffer space is set to one billion bytes and
// the number of packets that can be queued is one million.
// The maximum packet size is assumed to be 1000 bytes.
// Note: this program has not been bullet-proofed. Caveat emptor.

#include "stdinc.h"
#include "dlist.h"
#include "qSet.h"
#include "drr.h"

char* skip(char*);

main(int argc, char* argv[]) {
    printMode pMode;
    int i, m, queue, plen, seq, atim, tp;
    char buf[1002], *p;
    double wt[1001];

    if (argc != 3 || sscanf(argv[2], "%d", &tp) != 1)
        fatal("usage: drrSim printMode tp");
    else if (strcmp(argv[1], "verbose") == 0) pMode = verbose;
    else if (strcmp(argv[1], "trace") == 0) pMode = trace;
    else if (strcmp(argv[1], "terse") == 0) pMode = terse;
    else fatal("usage: drrSim printMode tp");

    if (fgets(buf, 1000, stdin) != NULL) {
        if (sscanf(buf, "%d", &m) != 1 || m > 1000)
            fatal("drrSim: malformed input");
        p = skip(buf);
        for (i = 1; i <= m; i++)
            if (sscanf(p, "%lf", &wt[i]) != 1)
                fatal("drrSim: malformed input");
        p = skip(p);
    }
    drr sched(1000000, m, 1000000000, 1000, wt, pMode, tp);
    while (fgets(buf, 990, stdin) != NULL) {
        if (sscanf(buf, "%d %d %d %d", &atim, &queue, &plen, &seq) != 4)
            fatal("drrSim: malformed input");
        sched.arrival(queue, plen, atim, seq);
    }
    sched.arrival(queue, plen, -1, seq);
    sched.summary();
}

char* skip(char *s) {
// Return pointer to first space character following the first
// non-space character at location s or later. If you reach EOS
// then just return pointer to the EOS character.
    while (isspace(*s)) s++;
    while (*s != EOS && !isspace(*s)) s++;
    return s;
}
Here is the class that implements the DRIR packet scheduler.

// Data structure for simulating a set of queues with a DRR // packet scheduler.

enum printMode { verbose, trace, terse };

class drr {
    int m;          // number of queues
    qSet *qs;       // queue set data structure
    dlist active;   // list of non-empty queues
    int cq;         // current queue;
    int maxlen;     // max packet length
    int *credits;   // credits[i] = # of credits for queue i
    double *wt;     // wt[i] = weight for queue i
    int *qquant;    // qquant[i] = quantum for queue i
    int ftim;       // time when current packet completes
    printMode pmode; // print mode
    int tperiod;    // time between trace printouts
    int tptim;      // time of next trace printout
    int *busy;      // time that i has been sending since
                    // last trace printout
    double *avgqlen; // average queue length
    int *maxqlen;   // maximum queue length
    int *mindelay;  // minimum packet delay
    int *maxdelay;  // maximum packet delay
    int *tbusy;     // total time queue i has been sending

public: drr(int N, int m1, int B, int maxlen, double wt1[], printMode pmode1, int tp) {
    // Allocate storage and initialize data structure.

    m = m1;     tperiod = tp; pmode = pmode1;
    qs = new qSet(N, m, B);
    active.reset(m);
    credits = new int[m]; wt = new double[m]; qquant = new int[m];
    busy = new int[m]; avgqlen = new double[m]; maxqlen = new int[m];
    mindelay = new int[m]; maxdelay = new int[m]; tbusy = new int[m];

    int i;
    double w = 1;
    for (i = 1; i <= m; i++) {
        credits[i] = 0;
    }

}
busy[i] = tbusy[i] = 0;
avgqilen[i] = 0;
maxqilen[i] = mindelay[i] = maxdelay[i] = 0;
w[i] = wt1[i]; w = min(w,wt[i]);
}
for (i = 1; i <= m; i++) {
    qquant[i] = maxlen*(wt[i]/w);
}
cq = 0; // indicates no active queues
ftim = 0;
tptim = tperiod;
}

drr::~drr() { delete qs; delete [] credits; delete [] busy; }

void drr::arrival(int i, int len, int t, int seq) {
// Simulate arrival of packet of length len, with sequence
// number seq, for queue i at time t. If t<0 then advance
// the simulation until all queues are empty.

    int p, q, stim, rp, s;

    while (!qs->empty() && (t < 0 || ftim <= t)) {
        p = qs->first(cq); // p is packet that finishes at ftim
        stim = ftim - qs->plen(p); // time p started transmission
        while (pmode == trace && tptim <= ftim) {
            printf("%6d: ",tptim);
            for (q = 1; q <= m; q++) {
                if (qs->atim(qs->last(q)) <= tptim) {
                    printf(" %5d",qs->qlen(q));
                    continue;
                }

                s = 0;
                for (rp=qs->first(q);rp!=Null;rp=qs->next(rp)) {
                    if (qs->atim(rp) > stim)
                        s += qs->plen(rp);
                }
                printf(" %5d",qs->qlen(q)-s);
            }
            printf("\n        ");
            for (q = 1; q <= m; q++) {
                if (q == cq)
                    busy[q] += min(ftim-stim,
                        min(tperiod,
                            max(0,tptim-stim)
                        )
                    );
                if (q == cq) busy[q] += min(tperiod,tptim-stim);
                printf(" %5.3f",double(busy[q])/tperiod);
                busy[q] = 0;
            }
            printf("\n    ");
            tptim += tperiod;
        }
    }

    busy[cq] += ftim - max((tptim-tperiod),stim);
tbusy[cq] += qs->plen(p);
    if (mindelay[cq] == 0)
mindelay[cq] = ftim - qs->atim(p);
else
    mindelay[cq] = min(mindelay[cq], ftim - qs->atim(p));
maxdelay[cq] = max(maxdelay[cq], ftim - qs->atim(p));
for (q = 1; q <= m; q++) {
    if (qs->atim(qs->last(q)) <= stim) {
        avglqlen[q] += qs->plen(p) * qs->qlen(q);
        continue;
    }
    s = 0;
    for (rp = qs->first(q); rp != Null; rp = qs->next(rp)) {
        if (qs->atim(rp) > stim)
            s += qs->plen(rp)*(ftim-qs->atim(rp));
    }
    avglqlen[q] += qs->plen(p) * qs->qlen(q) - s;
}
credits[cq] -= qs->plen(p);
qs->deq(cq);
p = qs->first(cq);
if (p != Null && credits[cq] >= qs->plen(p)) {
    ftim += qs->plen(p);
} else {
    q = active.suc(cq);
    if (p == Null) {
        active -= cq; credits[cq] = 0;
    }
    cq = (q != Null ? q : active(1));
    if (cq != Null) {
        p = qs->first(cq);
        credits[cq] += qquant[cq];
        ftim += qs->plen(p);
    }
}
if (p != Null && pmode == verbose) {
    printf("%8d, %8d, %4d, %6d, %8d\n", ftim, qs->atim(p), cq, qs->plen(p), qs->seq(p));
}
if (t < 0) return;
p = qs->enq(i, len, t, seq);
if (p == Null) fatal("drr: no space for arriving packet");
maxqlen[i] = max(maxqlen[i], qs->qlen(i));
if (qs->first(i) == p) {
    active &= i;
    if (active(1) == i) {
        cq = i; credits[cq] = qquant[cq];
        ftim = t + qs->plen(p);
        if (pmode == verbose) {
            printf("%8d, %8d, %4d, %6d, %8d\n", ftim, qs->atim(p),
                cq, qs->plen(p), qs->seq(p));
        }
    }
}
}
void drr::summary() {
    int q;

    if (pmode != terse) return;

    printf("  weights:");
    for (q = 1; q <= m; q++) printf("%8.6f ", wt[q]);
    printf("\n");

    printf(" avg qlen:");
    for (q = 1; q <= m; q++) printf("%8.0f ", avgqlen[q]/ftim);
    printf("\n");

    printf(" max qlen:");
    for (q = 1; q <= m; q++) printf("%8d ", maxqlen[q]);
    printf("\n");

    printf("min delay:");
    for (q = 1; q <= m; q++) printf("%8d ", mindelay[q]);
    printf("\n");

    printf("max delay:");
    for (q = 1; q <= m; q++) printf("%8d ", maxdelay[q]);
    printf("\n");

    printf(" avg load:");
    for (q = 1; q <= m; q++) printf("%8.6f ", double(tbusy[q])/ftim);
    printf("\n");
}

And here is the part that implements the set of queues.

// Data structure for simulating a set of queues. This version
// uses tail discard (arriving packet gets discarded if no space
// is available).

class qSet {
    int N;   // max number of packets
    int m;   // number of queues
    int b;   // amount of buffer space in use
    int B;   // total amount of buffer space
    int *ql; // ql[i] is space used by queue i
    int *fp; // fp[i] is first packet in queue i
    int *lp; // lp[i] is last packet in queue i
    struct packet {
        int pl;   // length in bytes
        int at;   // time of arrival (time unit is byte time)
        int sq;   // sequence number of packet
        int nxt;  // index of next packet in queue
    } *pkt;    // array of packets
    int free;  // start of free packet list

public: qSet(int=10000, int=100, int=1000000);
    ~qSet();
    int enq(int,int,int,int);  // add a packet to a queue
    int deq(int);   // remove first packet from a queue
    int first(int); // return index of first packet in queue
int last(int);  // return index of last packet in queue
int next(int);  // return index of next packet in queue
int qlen(int);  // return length of a queue
int plen(int);  // return length of packet
int empty();  // return 1 if nothing is queued
int freeSpace(); // return amount of free space in buffer
int atim(int);  // return arrival time of packet
int seq(int);  // return sequence number of packet

// Return index of first packet in queue
inline int qSet::first(int i) { return (1 <= i && i <= m) ? fp[i] : Null; }

// Return index of last packet in queue
inline int qSet::last(int i) { return (1 <= i && i <= m) ? lp[i] : Null; }

// Return index of next packet in queue
inline int qSet::next(int p) { return (1 <= p && p <= N) ? pkt[p].nxt : Null; }

// Return space used by queue
inline int qSet::qlen(int i) { return (1 <= i && i <= m) ? ql[i] : Null; }

// Return length of packet
inline int qSet::plen(int j) { return (1 <= j && j <= N) ? pkt[j].pl : Null; }

// Return free space in buffer
inline int qSet::freeSpace() { return B-b; }

// Return 1 if nothing is queued.
inline int qSet::empty() { return b>0 ? 0 : 1; }

// Return the arrival time of the packet.
inline int qSet::atim(int j) { return (1 <= j && j <= N) ? pkt[j].at : Null; }

// Return the sequence number of the packet.
inline int qSet::seq(int j) { return (1 <= j && j <= N) ? pkt[j].sq : Null; }

#include "stdinc.h"
#include "qSet.h"

qSet::qSet(int N1, int m1, int B1) {
    // Allocate storage and initialize qSet
    N = N1; m = m1; B = B1; b = 0;
    pkt = new packet[N+1];
    ql = new int[m+1]; fp = new int[m+1]; lp = new int[m+1];
    for (int i = 1; i < N; i++) pkt[i].nxt = i+1;
    free = 1; pkt[N].nxt = Null;
}

qSet::~qSet() { delete [] pkt; delete [] ql; delete [] fp; delete [] lp; }

int qSet::enq(int i, int len, int now, int seq) {
// Add a packet of length len to queue i at time now.
// If there is no space to add a new packet, return Null;
// Otherwise, return the index of the packet.
// The sequence number of the packet is seq.
if (free == Null || b + len > B) return Null;
if (fp[i] == Null)
    fp[i] = lp[i] = free;
else
    lp[i] = pkt[lp[i]].nxt = free;
pkt[free].pl = len; pkt[free].at = now; pkt[free].sq = seq;
free = pkt[free].nxt;
pkt[lp[i]].nxt = Null;
ql[i] += len; b += len;
return lp[i];

int qSet::deq(int i) {
// Dequeue and return the index of the first packet in queue i.
// Return Null if queue is already empty.
int j;
if (fp[i] == Null) return Null;
j = fp[i];
fp[i] = pkt[j].nxt;
if (fp[i] == Null) lp[i] = Null;
pkt[j].nxt = free;
ql[i] -= pkt[j].pl; b -= pkt[j].pl;
free = j;
return j;
}

The terse, verbose and trace results from packet1 are shown below.

weights:0.500000 0.200000 0.100000 0.050000
avg qlen:  1404  1270  831  944
max qlen:  5000  1900  1000  1000
min delay: 1000  5699  7398  8397
max delay: 4984  6891  7894  8893
avg load: 0.561798 0.213483 0.112360 0.112360

1000,  0,  1,  1000,  0
2000,  4,  1,  1000,  4
3000,  8,  1,  1000,  8
4000, 12,  1,  1000, 10
5000, 16,  1,  1000, 11
5700,  1,  2,  700,  1
6400,  5,  2,  700,  5
6900,  9,  2,  500,  9
7400,  2,  3,  500,  2
7900,  6,  3,  500,  6
8400,  3,  4,  500,  3
8900,  7,  4,  500,  7

400:  5000  1900  1000  1000
     1.000 0.000 0.000 0.000
800:  5000  1900  1000  1000
     1.000 0.000 0.000 0.000
1200: 4000  1900  1000  1000
<table>
<thead>
<tr>
<th>Time</th>
<th>Queues</th>
<th>Weights</th>
<th>Avg QLen</th>
<th>Max QLen</th>
<th>Min Delay</th>
<th>Max Delay</th>
<th>Avg Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1600</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>2400</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>2800</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>3200</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>3600</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>4000</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>4400</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>4800</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>5200</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>5600</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>6000</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>6400</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>6800</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>7200</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>7600</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>8000</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>8400</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>8800</td>
<td>1000</td>
<td>0.0000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

The terse mode printout for packets 2 appears below, followed by the two charts.

weights:0.010000 0.040000 0.070000 0.100000 0.130000 0.160000 0.190000 0.220000
avg qlen:  9188  10847  1067  1917  16672  13803  5522  14368
max qlen: 924068 713466 78631 111773 632641 141131 66437 151561
min delay: 40 40 40 40 40 40 40 40
max delay: 924068 713466 78631 111773 632641 141131 66437 151561
avg load: 0.019908 0.040919 0.086440 0.111776 0.137175 0.140742 0.111924 0.184159
Here is an additional pair of curves that shows the first million time units.
The code for the SCFQ simulation appears below. The main program is omitted since it is practically identical to the one for DRR.

// Data structure for simulating a set of queues with a SCFQ packet scheduler.

enum printMode { verbose, trace, terse };

class scfq {
    int m;   // number of queues
    qSet *qs;   // queue set data structure
    dheap *active;  // heap of non-empty queues
    int cq;   // current queue;
    int maxlen;   // max packet length
    int *vft;   // vft[i] = virt. fin. time for queue i
    double *wt;   // wt[i] = weight for queue i
    int ftim;   // time when current packet completes
    printMode pmode;  // print mode
    int tperiod;  // time between trace printouts
    int tptim;   // time of next trace printout
    int *busy;   // time that i has been sending since
                   // last trace printout
    double *avgqlen;  // average queue length
    int *maxqlen;  // maximum queue length
    int *mindelay;  // minimum packet delay
    int *maxdelay;  // maximum packet delay
    int *tbusy;   // total time queue i has been sending

    public: scfq(int, int, int, int, double*, printMode, int);
    ~scfq();
    void arrival(int,int,int,int); // simulate new packet arrival
    void summary();  // print summary in terse mode
};

#include "stdinc.h"
#include "dheap.h"
#include "qSet.h"
#include "scfq.h"

scfq::scfq(int N,int m1,int B,int maxlen,double wt1[],printMode pmode1,int tp) {
    // Allocate storage and initialize data structure.
    m = m1; tperiod = tp; pmode = pmode1;
    qs = new qSet(N,m,B);
    active = new dheap(m,2);
    vft = new int[m]; wt = new double[m];
    busy = new int[m]; avgqlen = new double[m]; maxqlen = new int[m];
    mindelay = new int[m]; maxdelay = new int[m]; tbusy = new int[m];

    int i;
    for (i = 1; i <= m; i++) {
        vft[i] = 0;
        busy[i] = tbusy[i] = 0;
        avgqlen[i] = 0;
        maxqlen[i] = mindelay[i] = maxdelay[i] = 0;
    }
}
wt[i] = wt1[i];
}
cq = 0; // indicates no active queues
ftim = 0;
tptim = tperiod;
}

scfq::~scfq() { delete qs; delete active; delete [] vft; delete [] busy; }

void scfq::arrival(int i, int len, int t, int seq) {
// Simulate arrival of packet of length len, with sequence
// number seq, for queue i at time t. If t<0 then advance
// the simulation until all queues are empty.

int p, q, stim, rp, s;

while (!qs->empty() && (t < 0 || ftim <= t)) {
    p = qs->first(cq); // p is packet that finishes at ftim
    stim = ftim - qs->plen(p); // time p started transmission
    while (pmode == trace && tptim <= ftim) {
        printf("%6d: ", tptim);
        for (q = 1; q <= m; q++) {
            if (qs->atim(qs->last(q)) <= tptim) {
                printf(" %5d", qs->qlen(q));
                continue;
            }
            s = 0;
            for (rp=qs->first(q); rp!= Null; rp=qs->next(rp)) {
                if (qs->atim(rp) > stim)
                    s += qs->plen(rp);
            }
            printf(" %5d", qs->qlen(q) - s);
        }
        printf("\n        ");
        for (q = 1; q <= m; q++) {
            if (q == cq)
                busy[q] += min(ftim-stim,
                                min(tperiod,
                                max(0, tptim-stim))
                        );
            printf("%5.3f", double(busy[q])/tperiod);
        }
        busy[q] = 0;
    }
    printf("\n\n");
    tptim += tperiod;
}
}

busy[cq] += ftim - max((tptim-tperiod), stim);
tbusy[cq] += qs->plen(p);
if (mindelay[cq] == 0)
    mindelay[cq] = ftim - qs->atim(p);
else
    mindelay[cq] = min(mindelay[cq], ftim - qs->atim(p));
maxdelay[cq] = max(maxdelay[cq], ftim - qs->atim(p));
for (q = 1; q <= m; q++) {
    if (qs->atim(qs->last(q)) <= stim) {
        avgqlen[q] += qs->plen(p) * qs->qlen(q);
continue;
}

s = 0;
for (rp = qs->first(q); rp != Null; rp = qs->next(rp)) {
    if (qs->atim(rp) > stim)
        s += qs->plen(rp)*(ftim-qs->atim(rp));
}

avgqlen[q] += qs->plen(p) * qs->qlen(q) - s;
}
qs->deq(cq);
if (qs->first(cq) == Null) {
    active->remove(cq);
} else {
    vft[cq] += qs->plen(qs->first(cq))/wt[cq];
    active->changekey(cq,vft[cq]);
}
if (!active->empty()) {
    cq = active->findmin();
    p = qs->first(cq);
    ftim += qs->plen(p);
} else {
    cq = Null;
    if (p != Null & pmode == verbose) {
        printf("%8d, %8d, %4d, %6d, %8d\n", ftim, qs->atim(p), cq, qs->plen(p), qs->seq(p));
    }
}
if (t < 0) return;
p = qs->enq(i, len, t, seq);
if (p == Null) fatal("scfq: no space for arriving packet");
maxqlen[i] = max(maxqlen[i], qs->qlen(i));
if (qs->first(i) == p) {
    if (active->empty()) {
        vft[i] = qs->plen(p)/wt[i];
        cq = i;
        ftim = t + qs->plen(p);
        if (pmode == verbose) {
            printf("%8d, %8d, %4d, %6d, %8d\n", ftim, qs->atim(p),
                    cq, qs->plen(p), qs->seq(p));
        }
    } else {
        vft[i] = max(vft[i], vft[cq]) + qs->plen(p)/wt[i];
    }
    active->insert(i,vft[i]);
}
void scfq::summary() {
    int q;
    if (pmode != terse) return;

    printf(" weights:");
    for (q = 1; q <= m; q++) printf("%8.6f ", wt[q]);
    printf("\n");
The outputs for packets1 appear below.

| weights: 0.500000 0.200000 0.100000 0.050000 |
|---|---|---|---|
| avg qlen: 1667 880 568 860 |
| max qlen: 5000 1900 1000 1000 |
| min delay: 1000 2699 4198 8397 |
| max delay: 6884 7391 7894 8893 |
| avg load: 0.561798 0.213483 0.112360 0.112360 |

<table>
<thead>
<tr>
<th></th>
<th>1000,</th>
<th>0,</th>
<th>1,</th>
<th>1000,</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000,</td>
<td>4,</td>
<td>1,</td>
<td>1000,</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2700,</td>
<td>1,</td>
<td>2,</td>
<td>700,</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3700,</td>
<td>8,</td>
<td>1,</td>
<td>1000,</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4200,</td>
<td>2,</td>
<td>3,</td>
<td>500,</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5200,</td>
<td>12,</td>
<td>1,</td>
<td>1000,</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5900,</td>
<td>5,</td>
<td>2,</td>
<td>700,</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6900,</td>
<td>16,</td>
<td>1,</td>
<td>1000,</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>7400,</td>
<td>9,</td>
<td>2,</td>
<td>500,</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>7900,</td>
<td>6,</td>
<td>3,</td>
<td>500,</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8400,</td>
<td>3,</td>
<td>4,</td>
<td>500,</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8900,</td>
<td>7,</td>
<td>4,</td>
<td>500,</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>400:</th>
<th>5000</th>
<th>1900</th>
<th>1000</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>800:</td>
<td>5000</td>
<td>1900</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>1200:</td>
<td>4000</td>
<td>1900</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>1600:</td>
<td>4000</td>
<td>1900</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>2000:</td>
<td>4000</td>
<td>1900</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>2400:</td>
<td>3000</td>
<td>1900</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>2800:</td>
<td>3000</td>
<td>1200</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0.250</td>
<td>0.750</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>Time (ms)</td>
<td>V1 (mV)</td>
<td>V2 (mV)</td>
<td>V3 (mV)</td>
<td>V4 (mV)</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>3200</td>
<td>3000</td>
<td>1200</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3600</td>
<td>3000</td>
<td>1200</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4000</td>
<td>2000</td>
<td>1200</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>0.250</td>
<td>0.000</td>
<td>0.750</td>
<td>0.000</td>
</tr>
<tr>
<td>4400</td>
<td>2000</td>
<td>1200</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>0.500</td>
<td>0.000</td>
<td>0.500</td>
<td>0.000</td>
</tr>
<tr>
<td>4800</td>
<td>2000</td>
<td>1200</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5200</td>
<td>2000</td>
<td>1200</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5600</td>
<td>1000</td>
<td>1200</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6000</td>
<td>1000</td>
<td>500</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>0.250</td>
<td>0.750</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6400</td>
<td>1000</td>
<td>500</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6800</td>
<td>1000</td>
<td>500</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7200</td>
<td>0</td>
<td>500</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>0.250</td>
<td>0.750</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7600</td>
<td>0</td>
<td>0</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
<td>0.500</td>
<td>0.500</td>
<td>0.000</td>
</tr>
<tr>
<td>8000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.750</td>
<td>0.250</td>
</tr>
<tr>
<td>8400</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>1.000</td>
</tr>
<tr>
<td>8800</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>1.000</td>
</tr>
</tbody>
</table>
The terse output for packets2 appears below, followed by the charts from the trace output.

weights: 0.010000 0.040000 0.070000 0.100000 0.130000 0.160000 0.190000 0.220000
avg qlen: 9188 10847 1067 1917 16672 13803 5522 14368
max qlen: 96106 117442 7262 14989 127425 141131 66437 151561
min delay: 40 40 40 40 40 40 40 40
max delay: 924068 713466 78631 111773 632641 506883 227324 342193
avg load: 0.019908 0.040919 0.086440 0.111776 0.137175 0.140742 0.111924 0.184159

Below are more expanded charts showing the first 1,000,000 time steps. Note that SCFQ divides the bandwidth more evenly than DRR during short time periods.
6. (15 points) Consider the design of a 32-port switch, based on a sub-divided bus with knockout concentrators at the output. Assume that the links operate at 1 Gb/s and that the circuit technology can support a maximum clock rate of 100 MHz. This means that each IPP must transmit cells on at least ten parallel signal lines and that each OPP must receive cells on at least 320 input lines. The ten signal lines leaving each IPP can be used in one of two ways. Either, an IPP can send one cell at a time in parallel form over the ten signal lines, or an IPP can send ten different cells concurrently over the ten signal lines. In the first case, each knockout concentrator has 32 inputs, which are ten bits wide. In the second case, each knockout concentrator has 320 inputs, which are each one bit wide.

How many outputs must the concentrators have in each of the two cases, to produce a packet loss probability of no more than $10^{-8}$ when the input load is 100%?

In the first case, the probability that an output receives $i$ cells in a given cycle is just

$$\binom{n}{i} \left(\frac{1}{n}\right)^i \left(1-\frac{1}{n}\right)^{n-i},$$

where $n=32$. The probability that an arriving cell is discarded is

$$\sum_{i=k+1}^{n} \binom{n}{i} \left(\frac{1}{n}\right)^i \left(1-\frac{1}{n}\right)^{n-i}$$

if the knockout concentrator has $k$ output, so we need to find the smallest value of $k$ for which

$$\sum_{i=k+1}^{n} \binom{n}{i} \left(\frac{1}{n}\right)^i \left(1-\frac{1}{n}\right)^{n-i} \leq 10^{-8}.$$ 

The smallest such $k$ is 10 when $n=32$. 


In the second case, the probability of receiving $i$ cells is 

\[
\frac{10^n}{i} \left( \frac{1-1/n}{1/n} \right)^{10n-i},
\]

so we need to find the smallest value of $k$ for which 

\[
\sum_{i=k+1}^{10n} \left( \frac{10n}{i} \left( \frac{1-1/n}{1/n} \right)^{10n-i} \right) \leq 10^{-8}.
\]

The smallest such $k$ is 32 when $n=32$.

What does this imply about the memory bandwidth required at the output ports? Compare the cost of implementing the knockout concentrators in each of the two cases. Which design do you think is the better choice? Why?

The second design requires a total of 32 concentrator outputs, so the bandwidth entering the memory is 3.2 Gb/s and the total memory bandwidth is 4.2 Gb/s. The first design requires a total memory bandwidth of 11 Gb/s, almost 3 times as much. The datapaths of the two designs have about the same complexity, with the second being perhaps a little less. There is a more control logic needed for the second design, but the overall difference between the two is likely to be fairly modest. The second design is clearly the better choice. The reduction in memory bandwidth is a very significant advantage.

7. Consider a shared buffer switch with $n$ inputs and $n$ outputs. Assume that there is no limit on the available buffer space, that the packet lengths are exponentially distributed, with mean $1/\mu$ and that the time between packet arrivals for each output is exponentially distributed, with mean $1/\lambda$. Give an expression for the probability that the shared buffer contains $i$ packets for a particular output.

Under the given conditions, each output can be modeled by an $M/M/1$ queue, so the probability that the shared buffer contains $i$ packets for a given output is $(1-\rho)^i\rho$ where $\rho = \lambda/\mu$.

Give an expression for the probability that the shared buffer contains a total of $i$ packets, assuming that the number of packets going to the different outputs are independent.

In this case, we have to consider all possible ways that we can get $i$ packets. So, the required expression is

\[
\sum_{i_1+\ldots+i_n=i} (1-\rho)^{i_1}\rho^{i_1} (1-\rho)^{i_2}\rho^{i_2} \ldots (1-\rho)^{i_n}\rho^{i_n}
\]

where the summation is over all non-negative integer values for $i_1,\ldots,i_n$ that add up to $i$. If we call this quantity $f(i,n)$, we can do the computation iteratively using the equation

\[
f(i,n) = \sum_{0\leq j\leq i} f(j,1)f(i-j,n-1)
\]

where $f(j,1)=(1-\rho)^j$.

Use this expression to estimate the amount of memory needed to ensure that the packet loss probability is no more than $10^{-6}$, when $n=16$ and the input load is 80%. Compare this to the amount of memory you would need if the memory were not shared.

A Visual Basic program to compute $f(0,n)+\cdots+f(i,n)$ is shown below. Using this, we find that the probability that the queue contains more than 61 packets is just under $10^{-6}$. For a 16 port shared buffer switch, the probability that the shared buffer has more than 183 packets is just under $10^{-6}$. So the shared buffer switch needs storage for 183 packets, while a switch with separate buffers for each
output would need $16 \times 61 = 976$, which is about 5.33 times as many as the shared buffer switch needs. Note that this method for computing the required buffer size over-estimates what's really needed by a small amount.

Function `sbuf(rho As Double, i As Integer, n As Integer) As Double`
'
' Compute the probability that a shared buffer switch with n ports
' and a uniform random input load of rho, has less than or equal to
' i packets in its buffer, assuming no limit on the number of packets
' that can be stored.
'
Dim h, j, r As Integer
Dim pi(10000, 32) As Double
Dim s As Double

pi(0, 1) = 1 - rho
For j = 1 To i
    pi(j, 1) = rho * pi(j - 1, 1)
Next j

For h = 2 To n
    For j = 0 To i
        pi(j, h) = 0
        For r = 0 To j
            pi(j, h) = pi(j, h) + pi(r, 1) * pi(j - r, h - 1)
        Next r
    Next j
Next h

s = 0
For j = 0 To i
    s = s + pi(j, n)
Next j
sbuf = s
End Function

Consider what happens when TCP traffic is applied to the shared buffer switch. Specifically, assume that $n/2$ of the outputs are receiving traffic from a large number of TCP streams, each with a large backlog of traffic. Also assume that there is a single queue in the shared buffer for each of the outputs, and that arriving packets are discarded if the queue is full. Qualitatively describe the queueing behavior of the shared buffer switch in this situation. Estimate the amount of memory needed by the switch to ensure that none of the busy output links experiences underflow, assuming that the network round trip time is 100 ms and that the output links operate at 1 Gb/s. Compare this to the amount of memory needed in a switch that does not share the memory among the different outputs.

In this situation, the different TCP flows going out on all the outputs can potentially become synchronized, since when the shared memory fills up, all the flows are likely to experience packet loss and reduce their window sizes together. To prevent the shared buffer from underflowing, we need a memory size that is comparable to the product of the output link bandwidth times the network round-trip delay. If the round trip time is 100 ms and the output links are 1 Gb/s, this comes to about 100 M bits for each congested link. If we allow for a half-second's worth of buffering, this number grows to 500 M bits per congested link. Since half of the links are congested, this reduces the memory usage by just a factor of 2, and since more than $n/2$ links could be congested, it's not clear that we would even get this reduction, in practice. Consequently for continuously backlogged TCP flows, there is little gain obtained from the shared buffer.
Based on all your answers above, evaluate the advantages and disadvantages of shared buffer switches, relative to switches with separate memories for each output.

Shared buffering works well, so long as the traffic going to different outputs is uncorrelated. Since most TCP flows complete their data transmission before going through multiple rounds of congestion avoidance, their behavior is more like that predicted by the first analysis, than by the second. This suggests that there is a significant reduction in memory that can be obtained using shared buffering. This may not always lead to a significant reduction in cost, since commercial memory components often do not have the ideal combination of IO bandwidth and memory capacity. For high speed routers, memory bandwidth is often the limiting factor, forcing a router designer to use more memory components in parallel to get the required bandwidth, leading to an over-supply of memory capacity. In this situation, there is less to be gained from sharing the memory among different ports.

Consider a crossbar-based switch with 64 ports in which each IPP has a single queue and during each arbitration cycle, the IPPs contend for the output that the first cell in the queue is addressed to. If every input has a cell in its queue and if the outputs these cells are addressed to are selected at random, what is the probability that no cells are directed to a particular output?

\[ (1 - 1/n)^n = (1 - 1/64)^{64} = 0.365 \]

What is the expected number of outputs that no cells are addressed to?

\[ 0.365n = 23.4 \]

Suppose that each IPP has two queues, one for cells addressed to even-numbered outputs and one for cells addressed to odd-numbered outputs. Assume that every IPP has a cell in each of its two queues and that the addresses of cells in the even queues are randomly selected from among the even outputs and that the addresses of cells in the odd queues are randomly selected from among the odd outputs. What is the probability that a given output has no cells addressed to it?

\[ (1 - 1/32)^{64} = 0.131 \]

What is the expected number of outputs that no cells are addressed to?

\[ 0.131n = 8.38 \]

Consider a system in which each IPP has such an odd-even queue arrangement, and during each arbitration cycle, each IPP attempts to send a cell from either of its queues. Estimate the maximum throughput possible in such a system.

In a single cell cycle, we would expect 64-8.38=55.62 cells to get through, under the given conditions, compared to about 40 for the system in which each IPP has a single queue. In subsequent cell cycles, the addresses of cells will not be independent, leading to a degradation in the throughput, but the degradation should be less in the case of an odd-even queue system than in the case of a single queue system. The actual maximum throughput should be between 80% and 85% of 64, or roughly 51 to 55 cells per cycle.
8. The time-slotted arbitration ring, described on page 3-12, is attractive because it is very simple to implement. It can be generalized to apply to systems with VOQs and to allow scheduling over multiple time steps. In this generalized version, the bit \( z_t \) is replaced by a value \( t_i \), which represents the earliest time step at which the output \( x_i \) is not yet scheduled to receive a cell. If input \( i \) has an “unscheduled” cell to send to output \( x_i \) and has no other cell scheduled to be sent at time \( t_i \), it schedules the waiting cell for transmission at time \( t_i \), and then increments \( t_i \) before passing its value to input \( i-1 \). The input keeps track of when each waiting cell is scheduled for transmission, and sends it at the appropriate time.

The matrix shown below represents a set of cells waiting to be scheduled through a crossbar switch using this generalized time-slotted arbitration ring. The entry in row \( i \), column \( j \) is the number of cells at input \( i \) going to output \( j \). For each input, list the output that sends it a cell to on each successive time step, until all the cells are sent.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Assume that initially, all the \( t_i \) values are zero and that \( x_i = i \).
The figure below shows what happens during steps 2 through 6 of the scheduling algorithm (nothing happens in the first step, since there are zeros in all the diagonal entries. The matrix at the top left highlights which VOQs are under consideration during step 2. The value of \( t_i \) after the step 2 is shown next, along with the partial schedule constructed at that point. The center matrix in the top row is shows the number of unscheduled packets in each of the VOQs and highlights the VOQs that are under consideration in the next step. In the partial schedules, the underscore character (_) is used to indicate time steps during which a particular input is not scheduled to send a cell.

\[ \begin{array}{c|c} t_i & schedule \\ \hline 1 & 2 \ 3 \ 1 \ 0 \ 3 \\ 2 & 0 \ 3 \ 1 \ 1 \ 1 \\ 3 & 0 \ 0 \ 0 \ 2 \ 3 \\ 1 & 4 \ 0 \ 0 \ 0 \ 0 \\
 \end{array} \]

The figure below shows what happens during steps 2 through 6 of the scheduling algorithm (nothing happens in the first step, since there are zeros in all the diagonal entries. The matrix at the top left highlights which VOQs are under consideration during step 2. The value of \( t_i \) after the step 2 is shown next, along with the partial schedule constructed at that point. The center matrix in the top row is shows the number of unscheduled packets in each of the VOQs and highlights the VOQs that are under consideration in the next step. In the partial schedules, the underscore character (_) is used to indicate time steps during which a particular input is not scheduled to send a cell.

\[ \begin{array}{c|c} t_i & schedule \\ \hline 0 & 1 \ 2 \ 0 \ 0 \ 3 \\ 2 & 0 \ 2 \ 0 \ 0 \ 1 \\ 3 & 0 \ 0 \ 0 \ 1 \ 2 \\ 0 & 1 \ 1 \ 1 \ 3 \ 0 \\
 \end{array} \]

The figure below shows what happens during steps 2 through 6 of the scheduling algorithm (nothing happens in the first step, since there are zeros in all the diagonal entries. The matrix at the top left highlights which VOQs are under consideration during step 2. The value of \( t_i \) after the step 2 is shown next, along with the partial schedule constructed at that point. The center matrix in the top row is shows the number of unscheduled packets in each of the VOQs and highlights the VOQs that are under consideration in the next step. In the partial schedules, the underscore character (_) is used to indicate time steps during which a particular input is not scheduled to send a cell.

\[ \begin{array}{c|c} t_i & schedule \\ \hline 0 & 1 \ 2 \ 0 \ 0 \ 2 \\ 1 \ 0 \ 2 \ 0 \ 0 \ 0 \\ 2 \ 0 \ 0 \ 0 \ 1 \ 2 \\ 0 \ 1 \ 1 \ 0 \ 2 \ 0 \\
 \end{array} \]
This figure shows steps 13-18.

The final schedule and the final values $t_i$ are shown below.

<table>
<thead>
<tr>
<th>$t_i$</th>
<th>schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1,2,3,<em>,5,1,2,5,</em>,5,2</td>
</tr>
<tr>
<td>11</td>
<td>2,3,4,5,0,2,<em>,</em>,0,2</td>
</tr>
<tr>
<td>11</td>
<td>4,5,0,<em>,</em>,4,5,0,5,0</td>
</tr>
<tr>
<td>11</td>
<td><em>,0,</em>,1,<em>,</em>,1,1,1,1</td>
</tr>
<tr>
<td>10</td>
<td>5,_,1,2,3,3,3,1,2,3,3</td>
</tr>
<tr>
<td>10</td>
<td>0,1,2,3,4,_,1,2,4,4</td>
</tr>
</tbody>
</table>