E-GENTING PROGRAMMING COMPETITION

2002

LECTURE NOTES

Sixth draft
Jonathan Searcy
27 April 2003

Table of Contents

1 INTRODUCTION

I am most pleased to be able to present this Public Lecture to so many interested people.

The purpose of this lecture is to disseminate the know-how needed to solve the problems in the E-Genting Programming Competition held last November.

After reviewing the responses to the programming competition, it is clear that more effort needs to be invested in learning fundamental programming techniques even if this means reducing the variety of programming tools to which students are exposed.

What I hope to do in this lecture is to consider the problems posed in the programming competition and review the fundamental techniques needed to answer each question. We will see that all the techniques are quite easy to master.

2 TRAFFIC LIGHTS

2.1 Preamble

Let's start with the Traffic Lights problem. If you recall, it involved programming a traffic light system. There were three essential elements:

1. The traffic lights had to execute a basic cycle, giving right-of-way to each of the North, East, South and West branches one after another.
2. If, during execution of the basic cycle, there were no vehicles waiting at the next branch, that branch should be skipped and right-of-way should be passed to the next branch with vehicles waiting.
3. If there were no vehicles waiting at any branch, the sequence should revert to the basic cycle, as if there were vehicles waiting at all the branches.

2.2 Operating System Interface

The traffic light problem was made slightly more difficult by an application/operating system interface that prevented the program being written top-to-bottom in the sequence of the traffic light cycle.

```
interface TrafficSystem {
    static final int     RED = 0;
    static final int     AMBER = 1;
    static final int     GREEN = 2;
                        // Traffic light colour codes
    void display (int northColour, int eastColour,
        int southColour, int westColour);
                        // Set the traffic light colours
}
abstract class TrafficApp {
    abstract void initApp (TrafficSystem);
                        // Initiate application
    abstract void tick ();
                        // Process clock tick
    abstract void vehicleWaiting (int northWait,
        int eastWait, int southWait, int westWait);
                        // Change in vehicle waiting inputs
}
```

The `TrafficSystem` interface represented the traffic light operating system.

The `display` method in the `TrafficSystem` interface could be called by the traffic light application to set the colours displayed by the physical traffic lights. The parameters were the traffic light colours to be displayed in the various directions. The parameter values may be `RED`, `AMBER` or `GREEN`.

The `TrafficApp` class was the super-class of the traffic light application. The traffic light application had to override the `initApp`, `tick` and `vehicleWaiting` methods to control the physical traffic lights.

The `initApp` method was called by the traffic light operating system to initialise any application data. The `initApp` method was required to initialise any application data and then return immediately. The application could not hold control in the `initApp` method. The `initApp` method had to save the reference to the `TrafficSystem` instance so that the application could call the `display` method later on. It also had to call `display` to establish the initial traffic light settings.

The `tick` method was called by the traffic light operating system once every second. The application (which the contestants had to design) had to use the calls it received through the `tick` method to measure the elapse of time.

The `vehicleWaiting` method was called by the traffic light operating system whenever it sensed a change in the state of the vehicle waiting sensors. Each parameter was zero if no vehicles were waiting on the branch and non-zero if vehicles were waiting.

Despite event-delivering nature of the operating system interface, it is still useful to express the traffic light program in top-to-bottom pseudo-code in order to develop a strong technique for solving similar problems.

2.3 Top-to-Bottom Pseudo-Code

In the Traffic Lights problem, the top-to-bottom pseudo-code would read something like this:

1. branch = NORTH;
2. Loop:
    a. Display green light combination (branch);
    b. For green time (branch):
        i. Wait for a clock tick [‘GREEN’ state];
    c. Display amber light combination (branch);
    d. For amber time (branch):
        i. Wait for a clock tick [‘AMBER’ state];
    e. Read vehicle waiting information;
    f. branch = Next (branch, vehicle waiting information);
3. End loop.

The traffic light operating system required the application to be written as an event-driven finite state machine. Most graphical operating systems, such as Windows or the Java Applet interface impose this application structure. It was for this reason that the traffic light application-programming interface was structured in the way it was.

Overcoming this inconvenience is not difficult. For every top-to-bottom program that must wait on external events, there is an equivalent event-driven finite state machine.

2.4 Event-Driven Finite State Machine

The top-to-bottom representation of the Traffic Light sequence is easily converted into an event-driven finite state machine. To do this, allocate one state for the uninitiated machine, and then one more for each ‘wait’ operation in the pseudo code. For Traffic Lights, the states are:

1. UNINITIATED – the state before the operating system calls initApp.
2. GREEN – the application is waiting for a clock tick while a green light combination is displayed.
3. AMBER – the application is waiting for a clock tick while an amber light combination is displayed.

The Traffic Lights pseudo-code can then be translated into a state transition table.

State	Event	Procedure
UNINITIATED	Initiate	1. branch = NORTH; 2. Display green light combination (branch); 3. state = GREEN.
GREEN	Clock tick	1. If green time(branch) has elapsed: a. Display amber light combination (branch); b. state = AMBER; 2. End if.

AMBER	Clock tick	1. If amber time(branch) has elapsed:
		a. Read vehicle waiting information;
		b. branch = Next (branch, vehicle waiting information);
		c. Display green light combination (branch);
		d. state = GREEN;
		2. End if.

The top-to-bottom and finite state machine modes are equivalent to client-mode and server-mode respectively.

When a client-mode process needs the services of a server, it sends a request to the server and suspends execution until the server responds with an answer. This is equivalent to the top-to-bottom representation of the Traffic Lights problem.

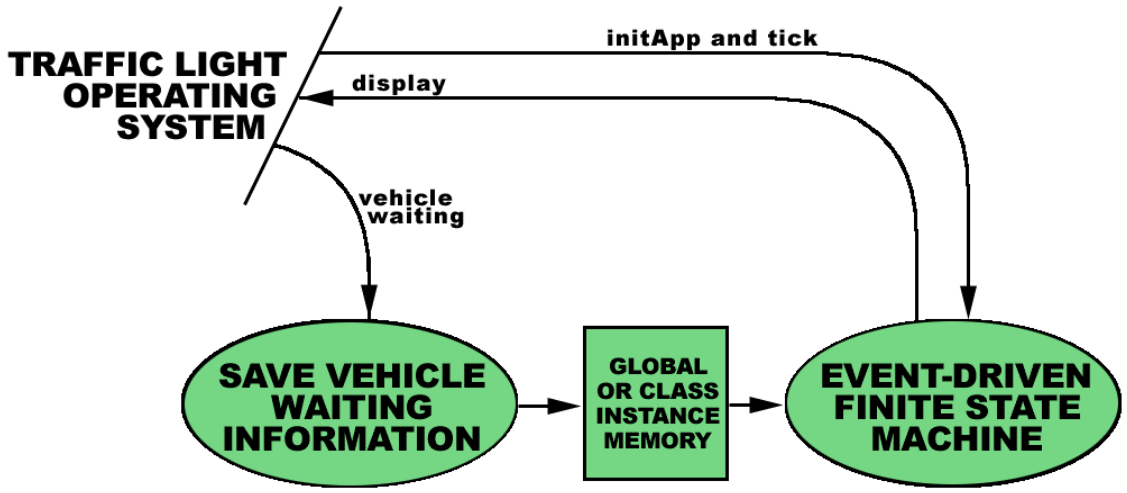
A server-mode process waits in a stable state until a request is received from a client. It then processes the request, perhaps with some internal state changes, and sends a response back to the client. This is equivalent to the finite state machine representation of the Traffic Lights Problem.

Conversion from client-mode to server-mode and vice versa is a common problem in computer programming. Frequently a programmer is stuck with two cooperating processes, both of which would be easier to program in client mode, but because they cooperate, one must be converted into server mode. I will discuss this issue further, later on.

2.5 Synchronising the Vehicle-Waiting Information

Two problems now remain before the Traffic Light application can be programmed. One is to transfer the asynchronous vehicle-waiting information passed to the application via the `vehicleWaiting` method to the ‘Read waiting-data statement in the pseudo-code. The other is the programming of the next branch function.

You can find a solution to the information-transfer problem by sketching a data flow diagram.



The `vehicleWaiting` function can be programmed to save the vehicle waiting information in global or class instance memory. The event-driven finite state machine can then read the vehicle waiting information at its convenience.

2.6 The Next Branch Function

Programming the next branch function should have been easy. Nevertheless, it confused all but a few of the contestants.

Although I did not program it this way in the published solution, I will describe a solution that is perhaps easier to understand.

First symbolic constants can be defined to represent each of the branches. For example:

```
static final int    NORTH     = 0;
static final int    EAST      = 1;
static final int    SOUTH     = 2;
static final int    WEST      = 3;
```

A further symbolic constant can be defined to represent the number of branches:

```
static final int    NBRANCHES = 4;
```

The basic cycle can now be represented as an increment operation in a modular domain of size NBRANCHES. It can be programmed in a multitude of ways. For example:

```
if (branch == NBRANCHES-1)
    branch = 0
else
    branch ++ ;
```

```
branch = (branch + 1) % NBRANCHES;
```

```
branch ++ ;
if (branch >= NBRANCHES) branch = 0;
```

```
if (++branch >= NBRANCHES) branch = 0;
```

```
branch ++ ;
if (branch >= NBRANCHES) branch -= NBRANCHES;
```

```
if (++branch >= NBRANCHES) branch -= NBRANCHES;
```

```
branch = (branch + 1) & (NBRANCHES - 1);
```

Choosing between the alternatives is not a matter of tossing a coin. The first alternative, although occupying 4 lines, operates in a way that must be obvious to even the intellectually compromised.

The second alternative, which makes use of the remainder operator, may execute slowly on some embedded microprocessors. Typically multiply and divide are slow operations. Nevertheless, the source is compact, and that makes it attractive.

The third, fourth and fifth alternatives are probably the best in most situations. They will give reasonable performance on most machines while being quite easy to understand.

The last alternative takes advantage of the coincidence that NBRANCHES is a power of 2. It will execute quickly on just about all machines, but may result in a program failure if a naive maintenance programmer changes the value of NBRANCHES.

The reason I have listed all these alternatives is that, with so many good ways to program the basic cycle function that work, why choose a bad one that doesn't. Yet many contestants did exactly that.

Next, let's program a method, `basicCycle` that increments the branch number in its modular domain.

```
int
basicCycle (
     int       branch)        // Branch number
{
     branch ++ ;
     if (branch >= NBRANCHES) branch = 0;
```

```
        return branch;
    }
```

Now, all that remains is to deal with the question of the existence or otherwise of waiting vehicles. Let's assume that the `vehicleWaiting` method was programmed in this way:

```
    boolean [] waitFlag;     // Vehicle waiting flags
                             // allocated in constructor
    void
    vehicleWaiting (
        int       northWait,
        int       eastWait,
        int       soutWait,
        int       westWait)
    {
        waitFlag[NORTH] = northWait != 0;
        waitFlag[EAST] = eastWait != 0;
        waitFlag[SOUTH] = southWait != 0;
        waitFlag[WEST] = westWait != 0;
    }
```

Incidentally, a common mistake in the programming competition was to use the greater-than operator to test the values of the integer arguments instead of the not-equal operator. A common non-zero value would be all bits set, or –1. In this circumstance, the programs submitted by the participants would have failed.

The next branch function can now be programmed.

```
    int
    nextBranch (
        int       branch,        // Branch number
        int []    waitFlag)      // Wait flags
    {
        if (
            waitFlag[NORTH] ||
            waitFlag[EAST] ||
            waitFlag[SOUTH] ||
            waitFlag[WEST]
        ) {
            do
                branch = basicCycle(branch);
            while ( ! waitFlag[branch] );
        } else {
            branch = basicCycle(branch);
        }
        return branch;
    }
```

Alternatively,it could be programmed in this way:

```
int
nextBranch (
     int       branch,         // Branch number
     int []    waitFlag)       // Wait flags
{
     int       oldBranch;      // Old branch number

     oldBranch = branch;
     do {
          branch = basicCycle(branch)
     } while (!waitFlag[branch] && branch!=oldBranch);
     if ( ! waitFlag[branch])
          branch = basicCycle(oldBranch);
     return branch;
}
```

Or even this way.

```
int
nextBranch (
     int       branch,         // Branch number
     int []    waitFlag)       // Wait flags
{
     int       i;              // General purpose index

     i = 0;
     do {
          branch = basicCycle(branch);
          i ++ ;
     } while (i <= NBRANCHES && !waitFlag[branch]);
     return branch;
}
```

In practice, there is not much between the three alternatives. The main problems are to avoid entering into an infinite loop in the `nextBranch` function and to avoid returning the same branch number originally passed to the function, especially when there are no vehicles waiting on any branch.

2.7 Making Use of Tables

Very few contestants made use of tables in the competition. Yet tables make the problem a lot easier. For example, the pseudo-code statement 'Display green light combination' can be easily, conveniently and efficiently programmed using a simple table.

```
static final int RED     = TrafficSystem.RED;
static final int AMBER   = TrafficSystem.AMBER;
static final int GREEN   = TrafficSystem.GREEN;

static final int greenComb[][] = {
     {GREEN, RED, RED, RED},
     {RED, GREEN, RED, RED},
     {RED, RED, GREEN, RED},
     {RED, RED, RED, GREEN}
};

/* Display green light combination */

lightSys.display (
     greenComb[branch][NORTH],
     greenComb[branch][EAST],
     greenComb[branch][SOUTH],
     greenComb[branch][WEST]
);
```

Similar tables can be used to store the green and amber times for each branch.

2.8 The initApp and tick Methods

Putting it all together, the `initApp` and `tick` methods can be programmed in the following way:

```
class MyTrafficApp
extends TrafficApp
{
     static final int    greenTime[] = {30, 60, 40, 50};
     static final int    amberTime[] = {5, 5, 5, 5};
     int                 ticksRemaining;
     static final int    GREEN_STATE = 0;
     static final int    AMBER_STATE = 1;
     int                 state;         // State variable
     TrafficSystem       lightSys;      // Reference to OS

     //-----------------------------------------------------
```

```
// INITIATE APPLICATION

void
initApp (
     TrafficSystem  ts)       // Reference to OS
{
     lightSys = ts;
     branch = NORTH;
     lightSys.display (
          greenComb[branch][NORTH],
          greenComb[branch][EAST],
          greenComb[branch][SOUTH],
          greenComb[branch][WEST]
     );
     state = GREEN_STATE;
     ticksRemaining = greenTime[branch];
}

//-----------------------------------------------------

// PROCESS CLOCK TICK

void
tick ()
{
     switch (state) {
     case GREEN_STATE:
          ticksRemaining -- ;
          if (ticksRemaining <= 0) {
               lightSys.display (
                    amberComb[branch][NORTH],
                    amberComb[branch][EAST],
                    amberComb[branch][SOUTH],
                    amberComb[branch][WEST]
               );
               state = AMBER_STATE;
               ticksRemaining = amberTime[branch];
          }
          break;
```

```
            case AMBER_STATE:
                 ticksRemaining -- ;
                 if (ticksRemaining <= 0) {
                      // waitFlag loaded by vehicleWaiting
                      branch = nextBranch (branch, waitFlag);
                      lightSys.display (
                           greenComb[branch][NORTH],
                           greenComb[branch][EAST],
                           greenComb[branch][SOUTH],
                           greenComb[branch][WEST]
                      );
                      state = GREEN_STATE;
                      ticksRemaining = greenTime[branch];
                 }
                 break;
            }
       }
}
```

2.9 Summary

The Traffic Lights problem, though simple, exercised the following techniques:

1. Conversion of a top-to-bottom sequence into an event-driven finite state machine.
2. Transferring data from one process to another using global or class-instance memory.
3. Programming a simple search in a modular domain.
4. The use of static data tables to drive an application program.

3 DATA PROJECTION AND SORTING

3.1 Preamble

The Data Projection and Sorting problem (colloquially, but incorrectly, known as the 'Sel-Sort' problem) involved reading a flat text file containing one customer data record on each line, sorting the file in a user-specified sequence and displaying the sort-keys in key priority order.

Example customer file:

```
custId name                     typeCode region address
"001" "Humphrey Bear"            "Host"  "AUS" "Colin St."
"002" "Donald Duck"              "Comic" "US"  "Hollywood"
"003" "Michael \"Micky\" Mouse" "Comic" "US"  "Disney World"
"004" "Goofy"                    "Comic" "US"  "Downtown LA"
```

Example field name string:

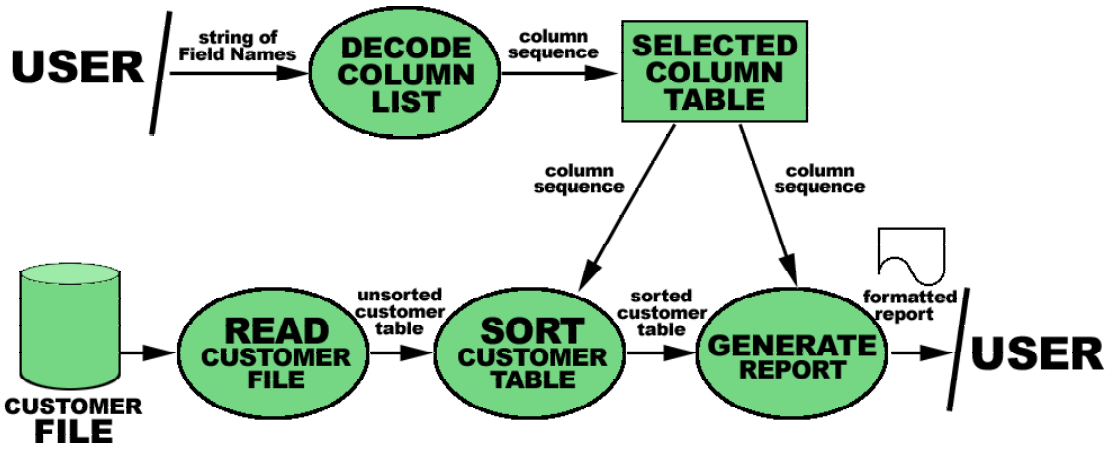
```
region, name, typeCode
```

Example output:

```
AUS  Humphrey Bear          Host
US   Donald Duck            Comic
US   Goofy                  Comic
US   Michael "Micky" Mouse  Comic
```

3.2 Data Flow Diagram

Here we have a data flow diagram of the Sel-Sort problem:



The problem reduces to converting the column list string into a selected columns table, reading the customer file, sorting the customer file and then listing the selected columns.

3.3 Selected Columns Table

The Selected Columns Table is one of the keys to a simple solution to the Sel-Sort problem.

The Selected Columns Table maps the order of the columns in the Customer File to the order of the columns in the sort key and the formatted report. It is best programmed as an array indexed by report column number that contains the Customer File column numbers corresponding to each of the report columns.

For example, if the string of field names was:

region, name, typeCode

Then the Selected Columns Table should contain:

```
int []    selColArr;     // Selected column array
int       selColCnt;     // Selected column count

selColArr = new int [5];
selColArr[0] = 3;        // region
selColArr[1] = 1;        // name
selColArr[2] = 2;        // typeCode
selColCnt = 3;           // number of columns
```

Of course, we cannot hard-code the loading of the table in this way. The program must automatically translate the field name string

3.4 Field Name Translation

Converting the string of field names into the Selected Columns Table is a straightforward problem in language translation, i.e. compiler writing. The input language is the string of field names and the output language is an array of column numbers.

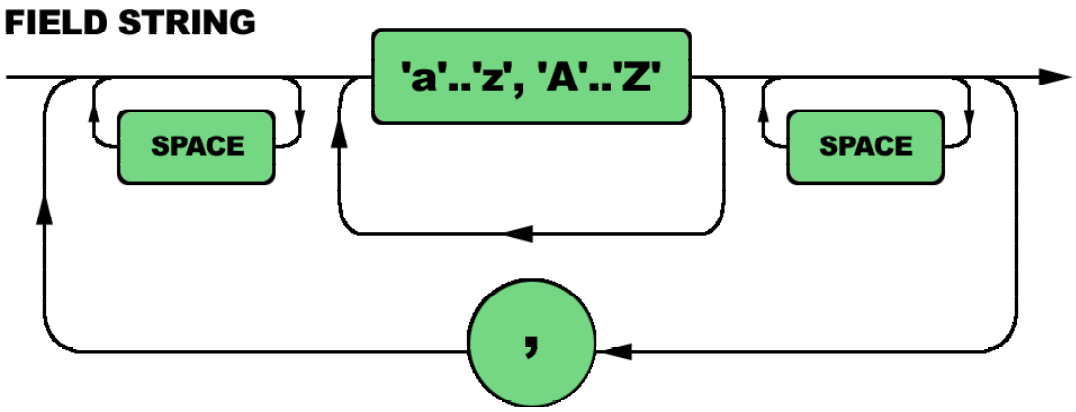
Without going deeply into the theory, the input language belongs to a class of languages that can be analysed with a single look-ahead character. To get started a single look-ahead character is loaded from the input string. The logic of the translator then follows the character sequence of the input string.

In order to formally describe the character sequence, it is useful to express it in Backus Naur Form or BNF. In this case the (slightly stylised) BNF is:

```
fieldString    : paddedName
               | fieldString ',' paddedName
               ;

paddedName     : spaces fieldName spaces
               ;

spaces         : /* empty */
```

```
                | spaces ' '
                ;

fieldName       : fieldChar
                | fieldName fieldChar
                ;

fieldChar       : ['a'..'z', 'A'..'Z']
                ;
```

BNF can be tricky to understand for those not used to it. As an alternative we can use railway diagrams. This is the railway diagram for the earlier BNF.



The string translator is easily derived from either the BNF or the railway diagram:

```
sr = new StringReader (fieldString);
ch = sr.read();
for (;;) {
    while (ch == ' ') ch = sr.read();
    if ( ! fieldChar(ch))
        reject ("invalid field identifier");
    sw = new StringWriter ();
    while (fieldChar(ch)) {
        sw.write (ch);
        ch = sr.read();
    }
    while (ch == ' ') ch = sr.read();
    // append column sw.toString() to table
    if (ch != ',') break;
    ch = sr.read();
}
if (ch != -1) reject ("unexpected character");
```

The function `fieldChar` returns true if the character is alphabetic, otherwise false.

As you can see, the code is simple and straightforward and the method is very flexible. It can analyse almost all the input formats you will ever come across.

The next problem is to take the column name stored in the StringWriter instance and convert it into a column number. This can be done with a simple table search.

```
static final String [] colNames = {
    "custId", "name", "typeCode",
    "region", "address"
};
int         i;          // General purpose index

for (
    i = 0;
    i < colNames.length &&
        ! colNames[i].equals(sw.toString());
    i ++
);
if (i >= colNames.length)
    reject ("unrecognised field name");
selColArr[selColCnt++] = i;
```

It is just too easy!

3.5 Reading the Customer File

The same processes that were used to translate the string of field names into the Selected Column Table can be used to translate the contents of the Customer File into a table such as this.

```
static final int     MAXCUSTROWS = 100000;
static final int     NUMCOLS = 5;
String [] []         custArr;       // Customer array
int                  custCnt;       // Customer count

custArr = new String [MAXCUSTROWS] [];
custArr[0] = new String [NUMCOLS];
// and so on as required
```

I will leave it for you to figure out the fine details.

3.6 Sorting the Customer Table

Both Java and C provide a mechanism for sorting arrays. The way to sort the Customer Table in C has been available on our Web site since the date of the competition. This is how it can be done in Java.

The Java library contains a static method that can sort the customer table.

```
Arrays.sort (
    custArr,        // reference to array
    0,              // beginning of sort range
    custCnt,        // end of sort range
    new CustComp (selColArr, selColCnt)
                    // reference to an object that
                    // compares the array rows
);
```

The first three arguments to the `sort` method are quite straightforward. They are a reference to the array and the array indices of the beginning and end of the sort range.

The last argument is more interesting. It is a reference to an object that implements the `Comparator` interface and compares the array rows.

Let's now look at how the object class that compares the array rows could be programmed.

```
class CustComp
implements Comparator
{
    int []      selColArr;   // Selected columns array
    int         selColCnt;   // Selected columns count

    CustComp (
        int []      sca,          // Selected columns array
        int         scc)          // Selected columns count
    {
        selColArr = sca;
        selColCnt = scc;
    }
```

The sorting process has two parameters. These are a reference to the Selected Columns Table array and a count of the number of columns in the Selected Columns Table. The constructor must save these parameters so that they can be used later on in the `compare` method.

This is how the `compare` method could be programmed.

```
public int
compare (
    Object      row1,       // First customer row
    Object      row2)       // Second customer row
{
    int         i;          // General purpose index
    int         c;          // Result of comparison

    for (i = 0; i < selColCnt; i++) {
        c = ((String[])row1)[selColArr[i]].compareTo (
                ((String[])row2)[selColArr[i]]
        );
        if (c != 0) return c;
    }
    return 0;
}
```

The sort-phase of the Sel-Sort problem is an excellent example of applied polymorphism. If I had to teach the principles of polymorphism, this is the example I would use.

The sort method can potentially sort an array of any kind of row. However it needs access to a sub-process to determine whether one row is above or below another in the sorting order. The naive solution is to simply pass a function name to the sort method as in the C library `qsort` function. However the function by itself is not enough. In the Sel-Sort problem, the function needs access to the Selected Columns Table in order to correctly sort the rows.

The traditional solution was to put the configuration parameters of the comparison function in global memory. However this makes the sorting process non-re-entrant. Re-entrance is the property of a computer program that allows more than one thread to execute the same code at the same time. In the traditional solution, only one sort thread could execute at a time because there was only one global instance of the configuration parameters.

The next answer was to pass the address of a data block to the sort function along with the address of the comparison functions. The data block could be passed to the comparison function along with the addresses of the rows to be compared. This architecture was the hard-coded predecessor of the modern object class.

An object class allows data and methods to be passed as a bundle to a process that manipulates broad classes of data, thereby facilitating the modularisation of processes, such as the `sort` method, that are sandwiched both above and below by code tailored to a particular application. This is the essential utility of polymorphism.

3.7 Generating the Report

Generating the report is simply a matter of using the Selected Column Table to write the Customer Table columns in the required order in much the same way as it was used to

compare the customer table rows in the `compare` method. Again, I will leave this part to you.

3.8 Summary

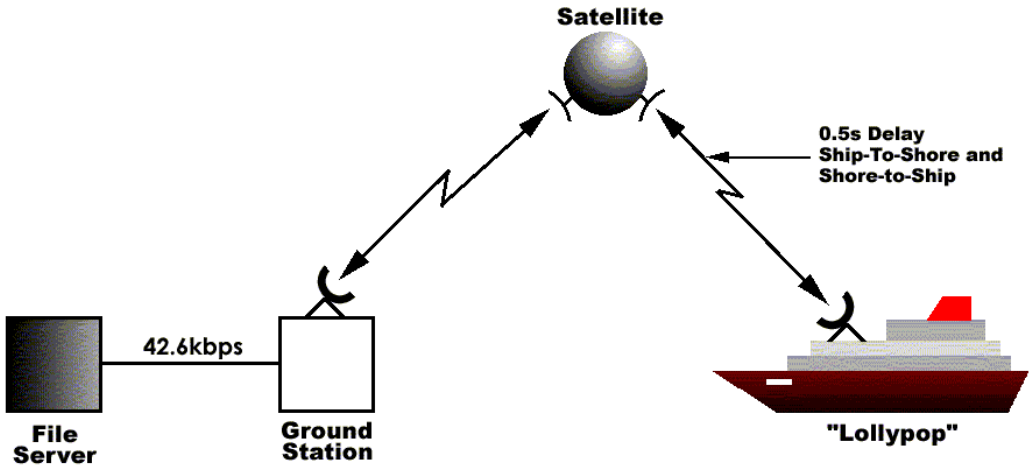
The Projection and Sort problem exercised the following techniques:

1. Data flow analysis of a straightforward application program.
2. Use of a table to map columns in the Customer File to the sort order and the columns to be displayed in the final report.
3. Single look-ahead character data stream translation.
4. Using a table to convert an identifier to a column number.
5. Making use of polymorphism to implement a standard interface to support the sort system call.

4 LOLLYPOP CRUISE LINES

4.1 Preamble

Now let us move on to some more interesting problems. *The Good Ship Lollypop* was a children's nursery rhyme. I use the name '*Lollypop*' to identify the archetypal cruise ship.



In the problem in the programming competition, a file server was connected to a satellite ground station via a 42.6 kbps modem. The ground station relayed data from the file server via a satellite, to the good ship Lollypop. The satellite connection had, for all intents and purposes, infinite bandwidth. It could relay gigabits per second. However, it introduced a delay of 0.5 seconds as a consequence of the orbit height of the satellite and the amount of time it takes for the microwave signal to reach the satellite and return to earth.

4.2 Description of the Problem

The crew of the Lollypop had complained about the amount of time it was taking to download a 1Mb daily data file. According to their reports, the download was taking between 20 minutes and half an hour. They needed to reduce this time to around 5 minutes in order to complete their daily routine on time.

The ISP that operated the satellite ground station suggested upgrading the connection between the file server and the ground station to a 256 kbps leased line. This upgrade would have cost around RM120,000.

A technician on Lollypop, who understood the nature of the protocol between the file server and Lollypop, was doubtful that increasing the speed of the connection between the file server and the ground station would have any effect.

Apparently the protocol between Lollypop and the file server was very simple. The Lollypop sent a 64-byte message to the ground station to request a 1kb block from the file server. The file server then responded with a 1kb block in a message 1088 bytes long. When the Lollypop received the 1kb block it repeated the sequence by sending another 64-byte message to request the next block.

The Lollypop only waited 5s for a response from the file server. If either the 64-byte message or the 1kb block were lost or corrupted in transit, the 5s period would expire, and the Lollypop would retransmit the original 64-byte message to restart the sequence. The technician on board the Lollypop, in an effort to find out more about the cause of the long download time, had kept records of data losses. He found that on the average, the likelihood of either a ship-to-shore or shore-to-ship satellite message being lost was 1/200. No message losses were recorded on the file server to ground station link.

The technician believed that the best way to reduce the download time was to change the communications protocol to transfer 8 blocks in parallel. He suggested that the protocol should be split into 8 parallel processes. The first process should take responsibility for transferring the first block in the file. The second process, the second block and so forth. When a process finished transferring a block, it should move on to transfer the next block waiting to be transferred until all the block transfers were completed.

4.3 The Programming Task

The programming task was to resolve the dispute between the ISP and the technician by writing a computer simulation program that simulated the transfer of, say, 1000 1Mb download files and estimated the transfer time under the following conditions:

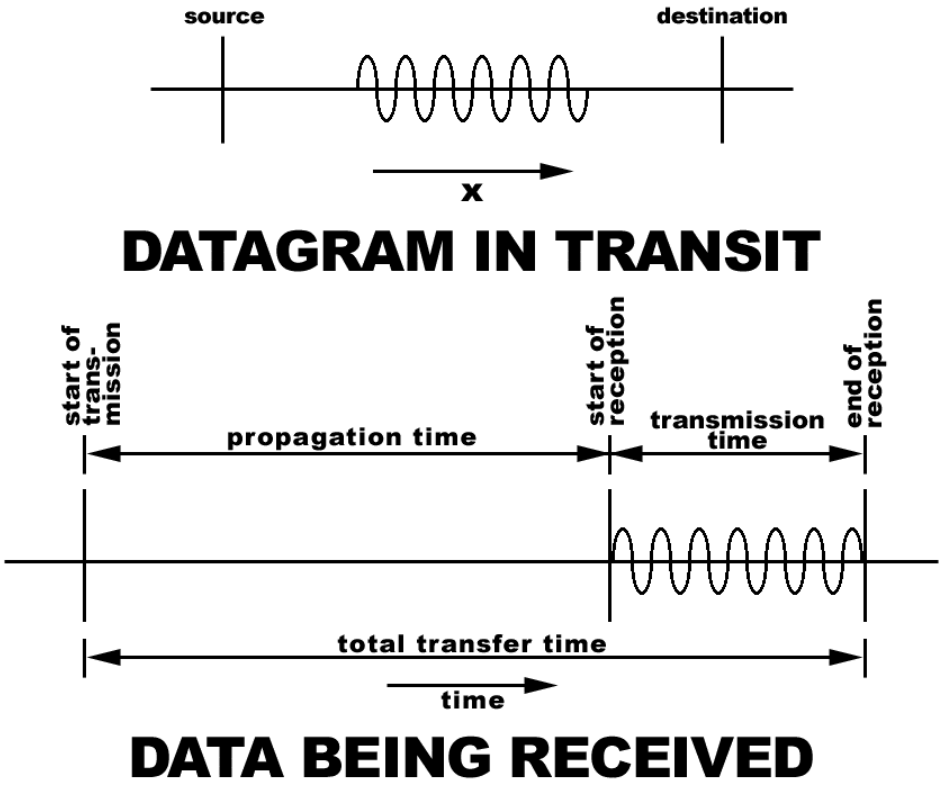
1. the existing single-threaded protocol with the 42.6 kbps modem speed;
2. the existing single-threaded protocol with the 256 kbps leased line speed proposed by the ISP;
3. the 8-threaded protocol proposed by the technician with the existing 42.6 kbps modem speed.

For the purposes of the model, the contestants could assume that any delays introduced by the file server, ground station and satellite were negligible. The ground station and the file server both queued messages to be transmitted across the line between them in the order that messages were put in the respective transmit queues. The connection between the file server and the ground station was serial, full duplex, asynchronous, 8 data bits, one stop bit and no parity.

4.4 Bandwidth and Propagation Time

Before we delve into the intricacies of modelling the communications interface, first we need to review some communications theory. The communications interface between the file server and Lollypop had two legs. The link from Lollypop to the satellite ground station had high bandwidth, but a long propagation time (0.5s). The link from the ground station to the file server had restricted bandwidth, but a negligible propagation time.

This diagram shows the difference between propagation time and the delay introduced by restricted bandwidth that I will call ‘transmission time’.



It is not hard to imagine a datagram flying through space as a train of photons. The datagram will have physical length and will take a finite amount of time to go past any given point.

If we now look at the signal at the destination in the time domain, we will see that from the time when the first photon is transmitted until it reaches the receiver, the destination signal is quiet. This period is the propagation time. It then takes a finite period for the individual bits in the datagram to be received and decoded. This period is the transmission time. The two together make up the total transfer time for the datagram.

We need to consider both propagation time and transmission time to solve the Lollypop problem.

The propagation time from the ground station to the Lollypop via the satellite was clearly stated in the question paper. It was 0.5s. The question paper stated that for all intents and purposes the satellite bandwidth was infinite. This meant that we could disregard the transmission time component of the satellite leg.

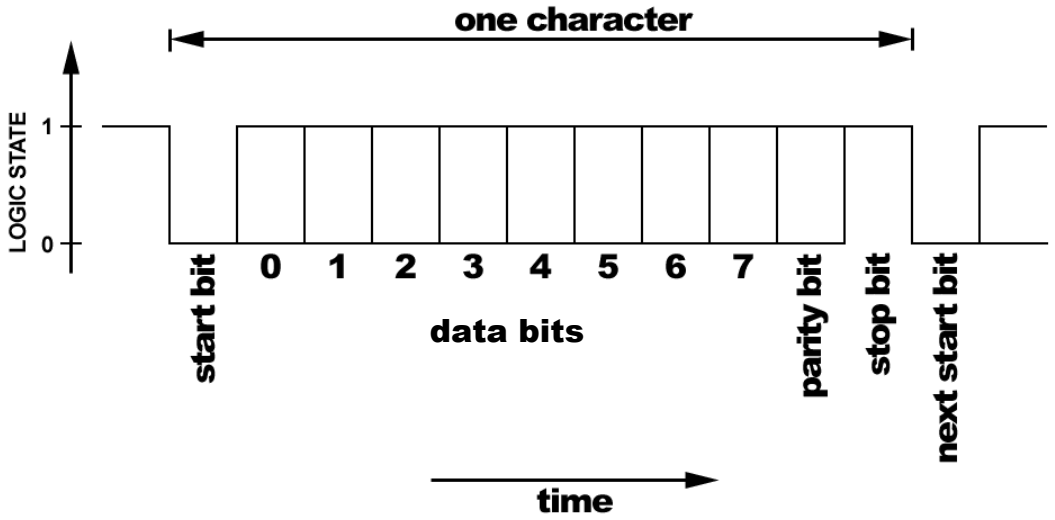
4.5 Asynchronous Communications

The delay introduced by the restricted bandwidth of the connection between the ground station and the file server is not so easily assessed. We know that the bandwidth is either

42.6k or 256k bits per second (note that it is a lower-case ‘k’ for 1000, not capital ‘K’ for 1024). The problem is: how many bits are there per byte.

The connection between the file server and the ground station was described as ‘serial, full duplex, asynchronous, 8 data bits, one stop bit and no parity’. This information fully describes the format of each character frame.

This diagram represents the transmission of a single asynchronous character.



ASYNCHRONOUS CHARACTER FRAME

The frame starts with a single start bit, which is one bit-period long. The data bits and an optional parity bit follow the start bit. One or more stop bits finish the frame. The transition from the previous stop bit to the next start bit establishes the time-domain reference for decoding the data bits.

We can now calculate the total number of bits required to transfer a byte of data:

Data	**Bits**
Start bit	1
Data bits	8
Parity bit	0
Stop bit	1
Total	10

And the transmission time:

$$t = 10 * N / f$$

Where: t = transmission time;
N = number of bytes in the datagram;
f = bandwidth.

4.6 Generalisation of the Problem

We can now move on to an analysis of the protocol described in the question paper. The paper asks for three analyses. Two are for single-threaded protocols, the last is for the multi-threaded protocol proposed by the technician.

It takes little more than high-school maths to calculate the transfer time for the 1Mb download file for the first two alternatives, though the possibility of one or more lost datagrams adds a twist that will slow down the over-confident.

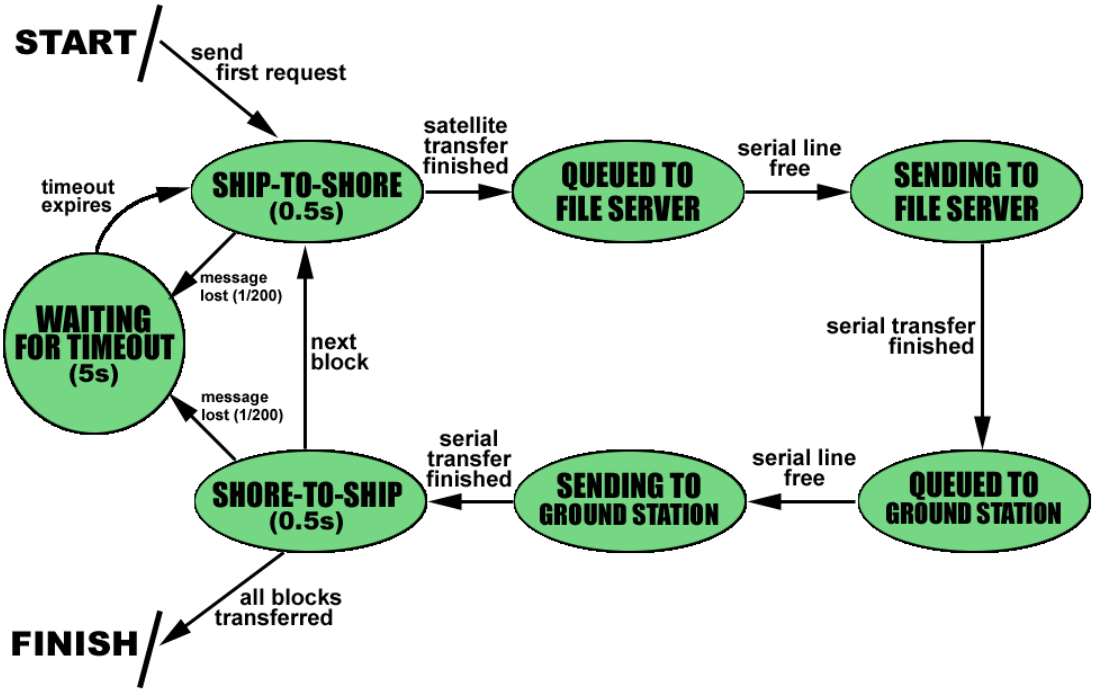
The third alternative, however, is resistant to a simple mathematical solution. Datagrams from the multiple threads will become queued waiting for transmission over the serial link. Queuing times, and especially the times associated with the initial loading of the queues and their final emptying, are difficult to evaluate with a pen and paper, especially when the possibility of a lost datagram or two is thrown in. The problem is that the binomial tree of possible outcomes becomes overwhelmingly complicated.

Given that we must resort to simulation to evaluate the third alternative, and that the question paper asks us to resolve the dispute by simulation anyway, we may as well consider the two single-threaded alternatives to be one-threaded instances of the more general multi-threaded problem and proceed immediately to modelling the multi-threaded alternative.

4.7 Analysis of the Protocol

We can now move on to an analysis of the multi-threaded protocol described in the question paper.

With some thought we can derive this state transition diagram for a single thread of the multi-threaded protocol.



STATE TRANSITION DIAGRAM

The protocol starts with the transmission of a 64-byte request ship-to-shore. The high bandwidth of the satellite connection means that we can disregard the possibility of the request being queued behind other datagrams waiting for satellite transmission. The transfer from ship to shore will take 0.5 seconds.

Next, the request must be transmitted over the serial link. However, because of the finite bandwidth of the serial link, there is a possibility that another thread might be transmitting on the link when the satellite transfer completes. If this situation arises, the datagram will be queued waiting for its turn to be transmitted.

When the datagram reaches the head of the queue and the serial link becomes free, it will be transmitted across the serial link from the ground station to the file server. The transmission time of this transfer can be calculated using the formula derived earlier.

When the file server receives the request, it will be processed and a data block will be queued for transmission back across the serial link to the ground station. The question paper tells us that the processing time is negligible, so for the purposes of analysis, the protocol state changes directly from a request being received by the file server to a data block being queued to the ground station.

When the response reaches the head of the queue to the ground station and the serial link becomes free, the data block is transmitted across the serial link to the ground station. The elapsed time can once again be calculated using the transmission time formula.

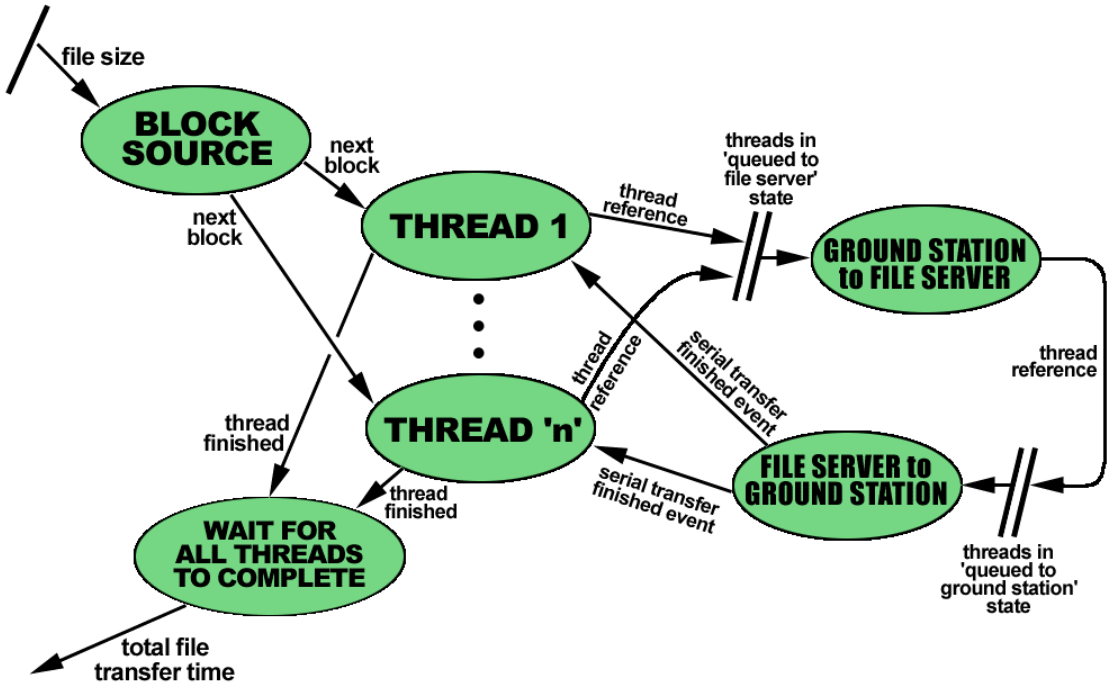
When the serial transfer completes, the data block is sent across the satellite link back to the ship. This process takes another 0.5 seconds.

When the ship receives the data block, if other data blocks remain to be requested, a new request is sent over the satellite link. If all data blocks have been requested, the thread terminates.

Last, there is the 1/200 possibility that a datagram may be lost on either of the two satellite legs. If this occurs, the protocol waits for the 5-second timeout and then proceeds to retransmit the request.

4.8 Model Data Flows

The next problem is to consider what happens when several of the single threads are simulated in parallel in the simulation program. It is not a huge leap to go from the preceding state transition diagram to this data flow diagram.



SIMULATOR DATA FLOW DIAGRAM

In a live transfer the Block Source would be the data file on the server. In our simulator it need be nothing more than a counter to determine when all the blocks have been transferred.

Each thread is a state transition machine similar to the one I just described, but with the serial transfer components consolidated into a single wait state for the queues and the serial transfers. In effect, the serialised states of individual threads have been taken out and transferred into two queues and two transfer processes shared by all threads.

When a thread enters the 'queued to file server' state, it queues a reference to itself in the 'Queued to File Server' queue.

The Ground Station to File Server process can monitor the Queued to File Server queue. When a request appears in the queue, it can wait, in virtual time, for the transfer across the serial link to complete and then but the thread reference in the 'Queued to Ground Station' queue.

In turn, the File Server to Ground Station process can monitor the Queued to Ground Station queue and simulates the transfer time from the file server to the ground station. When the simulated transfer completes, it notifies the thread of a serial transfer finished event.

One further process monitors the states of the threads. When all the threads finish, the current virtual time is the total file transfer time.

4.9 Thread State Transitions

Now let us consider the state transitions of the processes that simulate the communications threads in the data flow diagram.

State	Event	Processing
Uninitiated	Initiate	If the file size is non-zero: Decrement the count of blocks remaining; Block start time = current virtual time; In 1/200 cases: Next time = block start time + 5s; State = message lost; Otherwise: Next time = current virtual time + 0.5s; State = ship-to-shore; End; Else: Emit 'thread finished'; End if.
Ship-to-shore	Next time reached	State = queued; Append thread reference to 'Queued to File Server' queue.
Queued	Serial transfer finished	In 1/200 cases: Next time = block start time + 5s; State = message lost; Otherwise: Next time = current virtual time + 0.5s; State = shore-to-ship; End.

Shore-to-ship	Next time reached	If blocks remain to be transferred: Decrement the count of blocks remaining; Block start time = current virtual time; In 1/200 cases: Next time = block start time + 5s; State = message lost; Otherwise: Next time = current virtual time + 0.5s; State = ship-to-shore; End; Else: Emit 'thread finished'; End if.
Message lost	Next time reached	Block start time = current virtual time; In 1/200 cases: Next time = block start time + 5s; State = message lost; Otherwise: Next time = current virtual time + 0.5s; State = ship-to-shore; End.

4.10 Serial Transfer State Transitions

The two serial transfer processes and their input queues are really multiple instances of essentially the same process. The only differences are the different transfer times of the request and data block datagrams and the different treatment of completed transfers. These differences can be addressed with parameters passed to the constructor of a class that represents a transfer process.

We can now consider the state transitions of the serial transfer processes.

State	**Event**	**Processing**
Uninitiated	Initiate	State = inactive.
Inactive	Transfer request	State = active; Current thread = requesting thread; Next time = current virtual time + transfer time.
Active	Transfer request	Append thread reference to the transfer queue.

Active	Next time reached	If this is 'Ground Station to File Server': Send transfer request for the current thread to the File Server to Ground Station serial transfer process. Else if this is 'File Server to Ground Station' Send a serial transfer finished event to the current thread; End if; If the transfer queue is empty: State = inactive; Else: Remove the thread reference from the head of the transfer queue and make it the current thread; Next time = current virtual time + transfer time; End if.

4.11 Simulator Executive

We have now defined the processing, in terms of state transitions, of each process in the data flow diagram. All that remains is to consider how these processes will be linked together. This is quite straightforward. Consider the following main line:

```
Current virtual time = 0;
Initiate threads;
While threads are not finished:
        Search the threads and the serial transfer processes that are currently
        waiting for time to elapse and select the process 'p' with the earliest value
        of 'next time';
        Current virtual time = next time;
        Trigger 'next time reached' in p;
End while.
```

All that is required is an executive that polls each of the processes and selects the one that needs to be executed next. Each process that is waiting for time to elapse has a 'next time' value. The executive need only search for the process with the lowest, active 'next time' value, advance the current virtual time to that 'next time' value and then trigger the 'next time reached' event of the process. When all the threads have entered the 'finished' state, the current virtual time is the file transfer time.

An important thing to understand is that a process in a data flow diagram is rarely the same thing as a process in a multi-tasking or time-sharing operating system. A process in a multi-tasking or time-sharing context is an independently executing program with its own program counter, data segment, stack and so forth.

A process in a data flow diagram consists of state information, and methods that receive and transmit information and alter the state of the process. Indeed a class instance in object-oriented programming is frequently the same concept as a process in a data flow diagram. However, a class instance is not always the best or most convenient way to implement a process in a data flow diagram. Frequently the overheads of encapsulation

exceed the benefits of abstraction. As has been learned time and time again in a multitude of disciplines, rarely is there a panacea for all ailments. Usually the best cure is found by investigating a wide variety of potential solutions.

Frequently the best way to implement a process in a data flow diagram is with nothing other than its state information stored in some convenient way. Indeed this is the implementation of the thread processes in the published solution to the Lollypop problem.

4.12 Summary

I will not take the Lollypop solution any further. The connection between the state transition tables, the executive and the solution published on the Web is quite straightforward.

Summing up, the Lollypop problem exercised the following:

1. An understanding of the components of datagram transfer times over long and short distances and at varying bandwidths.
2. An understanding of the format of asynchronous character transmission frames and how to calculate asynchronous data transfer times.
3. Conversion of a description of a protocol into a state transition diagram.
4. Conversion of the protocol state transition diagram into a data flow diagram of the simulator.
5. Representing of the processes on the data flow diagram as finite state machines.
6. Designing an executive to schedule the execution of the processes on the data flow diagram.

You won't knock over the Lollypop problem by sitting down on a keyboard and tapping away. It required serious analysis, but if the analysis was done well, the Lollypop problem did not generate a large number of source statements.

5 OLD TAPE DATA RECOVERY

5.1 Preamble

The most difficult problem posed in the competition involved recovering data from an old magnetic tape. Apparently, twenty years ago an eminent scientist with considerable foresight undertook extensive climatic data collection across peninsula Malaysia. At the time of the competition there was considerable debate as to whether two decades of progressively increasing vehicle emissions and industrial pollution had altered the climate of the peninsula. A team of scientists had been appointed to study the matter. They wanted to make use of the twenty-year old data but were unable to read the old backup tape format.

5.2 Tape Format

The old tapes were an image copy of the original disk file system. The disk file system consisted of 1024 byte blocks arranged in this way:

Block number	Contents
0	Bootstrap loader for the operating system
1 to 31	Directory blocks
32 onwards	Pointer and data blocks

5.3 Directory Block Format

Each directory block consisted of an array of directory entries, each with this structure:

Byte number	Contents
0 to 15	File name
16 to 19	File length in bytes.
20 to 23	Block number of the first data block in the file
24 to 27	Block number of a pointer block that contained the block numbers of the 2nd to the 257th data blocks in the file
28 to 31	Block number of a pointer block that contained the block numbers of a second level of pointer blocks that contained the block numbers of the 258th to the 65,793rd data blocks in the file.

Unused data block numbers were zero. For example, if the length of the file were 1024 bytes, only the first block number in the directory entry would be used. If the length of the file was 2048 bytes, the first and second block numbers in the directory entry would be active, but only the first block number in the first pointer block would be used.

5.4 Objective

The objective was to write a computer program to accept a file name as an input parameter, extract the file from the tape and write it to the local disk drive.

5.5 Complications

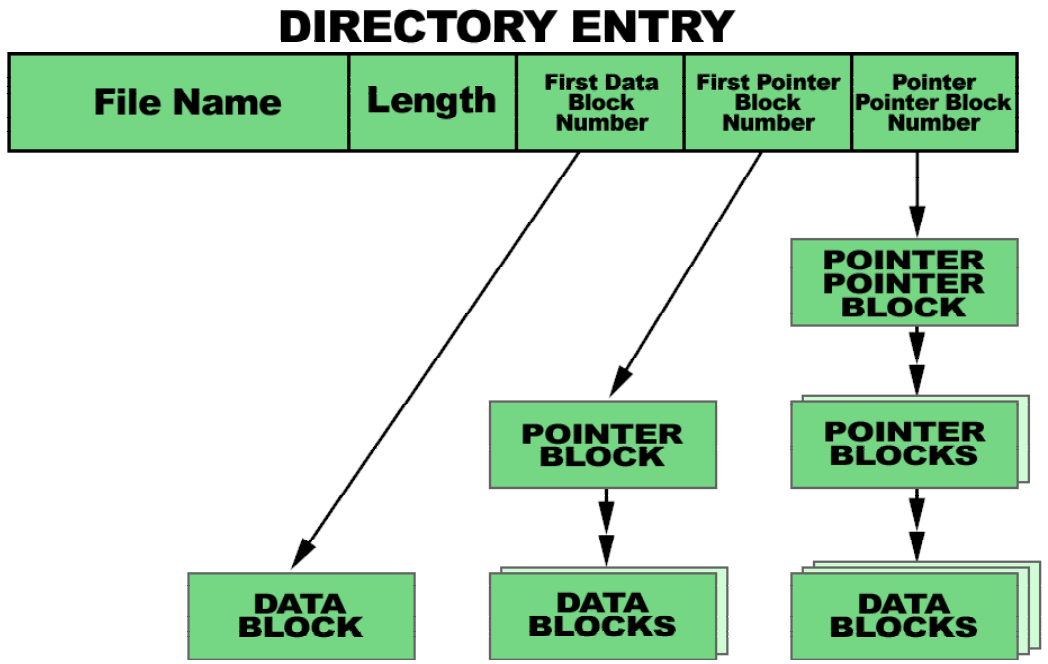
File names were between 1 and 16 characters long. If the file name was less than 16 characters, it was null terminated. If it was 16 characters long, the null terminator was omitted. Unused directory entries had a null character in the first character of the file name.

The byte order of the block numbers was big-endian (i.e. the most significant byte came first). The file extraction program had to run on an Intel microprocessor, a small-endian computer.

Because the backup media was tape, it could only be sequentially read and rewound. The file extraction program had to endeavour to extract the requested file from the tape in the minimum number of passes. The file extraction program could not use more than 500K of data RAM. There was not enough free disk space to unload the entire tape onto disk and then randomly access the copy of the tape on disk.

5.6 Entity Relationship Diagram

The first step in solving the Old Tape problem is to get a good definition of the structure of the data on the tape.



ENTITY RELATIONSHIP DIAGRAM

Blocks 1 to 31 each contained an array of directory entries. Each directory entry was 32 bytes long. This meant that there must also be 32 directory entries per tape block. The file system had a capacity of 31 x 32 or 992 files.

Each directory entry contained a 16-byte file name, followed by four 4-byte numbers.

The first number was the file size in bytes.

The second number was the block number of the first data block in the file.

The third number was the block number of a pointer block that contained an array of block numbers of additional data blocks.

The fourth number was the block number of a pointer block that contained an array of block numbers of further pointer blocks, which, in turn contained the block numbers of additional data blocks.

As I mentioned earlier, two peculiarities complicated the directory format. First the file name was only null-terminated if it was less than 16 characters long. If it was 16 characters long it was not null terminated.

Second, the numbers were all big-endian and the computer to be used for file extraction was little-endian.

Dealing with the conditional absence of the null-terminator is straightforward and at this late stage not worthy of further discussion. We should, however, spend a few moments on the byte order conversion problem.

5.7 Big-Endian to Little-Endian Conversion

Apparently the terms big-endian and little-endian are derived from Jonathan Swift's novel *Gulliver's Travels*. In *Gulliver's Travels* the Lilliputians fight a civil war over whether boiled eggs should be opened at the big end or the little end.

So it is with the question of whether the first byte in a word should be the most significant byte or the least significant byte. There are advantages and disadvantages with both architectures.

Nevertheless, as the engineers and mathematicians debate the pros and cons of byte order, programmers like us must deal with machines built both ways.

The byte order problem can be dealt with by simply swapping the first and fourth, and second and third bytes in each word, but this is not portable. If the program is compiled on a machine with the same byte order as the data, the program will fail. A better way of dealing with the byte order problem is this:

```
unsigned long
BigEndToLong (
     unsigned char  b[4])     // Big endian number
{
     unsigned long  l;        // Long value

     l = b[0];
     l = (l << 8) | b[1];
     l = (l << 8) | b[2];
     l = (l << 8) | b[3];
     return l;
}
```

Notice the similarity between this algorithm and the algorithm for string-to-number conversion. It is simply a string-to-number conversion of four radix-256 digits.

The algorithm is also efficient. The shift and or operations are all fast register-based instructions.

5.8 Hunters and Gatherers

Next we must decide whether to create a solution in the style of a hunter or a gatherer.

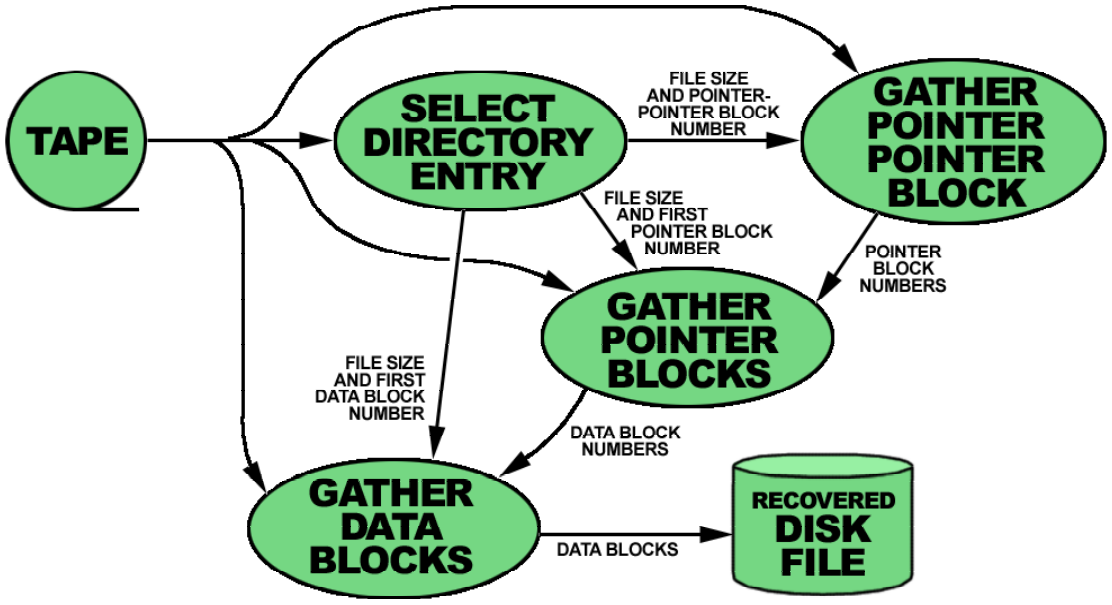
HUNTERS AND GATHERERS

The hunters prefer to track down and catch their prey. Their strategy for the Old Tape problem is to recursively descend the pointer block tree to get the data blocks. But in this case their strategy is frustrated by the inability of the tape drive to randomly read specific blocks. Some obsessive hunters have suggested that the tape drive could be rewound and spooled forward to seek each required block. Imagine several thousand rewinds and re-reads. I'm not that patient. The hunters are going to go hungry.

The gatherers, on the other hand, scan the tape with a manifesto of the information that they need. As they scan the tape they recognise and pick up pieces of relevant information and improve their manifesto until they have collected the complete data file. The gatherers can solve the problem.

5.9 The Basic Strategy of the Gatherers

The basic strategy of the gatherers is this.



DATA FLOW DIAGRAM

First, a Select Directory Entry process can skip the bootstrap block and scan the directory blocks, selecting the directory entry corresponding to the required file.

The Select Directory Entry process can then pass the file size and the first data, pointer and pointer-pointer block numbers to other processes for gathering the data and pointer blocks and the pointer-pointer block.

These processes, in turn, can gather the block numbers of the other pointer and data blocks and ultimately deliver the data blocks to the recovered disk file.

5.10 Gathering Data Blocks

The process for gathering data blocks can be thought of as an array of sub-processes, one for each data block.

5.11 How Many Data Blocks?

The number of data blocks can be calculated using the following formula:

$$Nd = (Fs + Bs - 1) / Bs$$

Where: Nd = the number of data blocks;
Fs = the file size

Bs = the block size

The calculation is simply the file size divided by the block size, rounded up. To get integer division to round up instead of down add the denominator minus one to the numerator.

5.12 Gathering Individual Data Blocks

This pseudo-code describes the sub-process for gathering each data block:

1. Receive the data block number from either the Select Directory Entry or Gather Pointer Blocks process;
2. Repeat:
    a. Receive a tape block;
3. Until the tape block number matches the data block number received earlier;
4. Write the data block to its correct position in the Recovered Disk File.

5.13 Gathering Pointer Blocks

The process for gathering pointer blocks is also an array of sub-processes, one for each pointer block.

The number of pointer blocks can be found with a formula similar to the one for the number of data blocks.

This pseudo-code for the sub-processes to gather each pointer block is this:

1. Receive the pointer block number from either the Select Directory Entry or Gather Pointer-Pointer Block process;
2. Repeat:
    a. Receive a tape block
3. Until the tape block number matches the pointer block number received earlier;
4. For each data block number in the pointer block:
    a. Send the data block number to the appropriate Gather Data Blocks sub-process.

The data block numbers passed to the Gather Data Blocks process are actually tuples containing a disk block number and a tape block number. The disk block number represents the position of the block in the disk file and the tape block number represents the position of the block on the tape.

The disk block number identifies the gather data block sub-process that should receive the tape block number. It can be found from the position of the gather pointer block sub-process in the array of sub-processes and the position of the data block number within the received pointer block.

The tape block number is the data block number stored in the pointer block.

5.14 Gathering the Pointer-Pointer Block

The process that gathers the pointer-pointer block is virtually identical to the process that gathers the pointer blocks with the exception that pointer block numbers are passed to the Gather Pointer Block sub-processes instead of data block numbers.

5.15 Main Line

Now that we've identified the structure of each of the processes in the data flow diagram, and understand how many passes are required, we are in a position to create a main line to read the tape and pass the tape block stream to each of the processes.

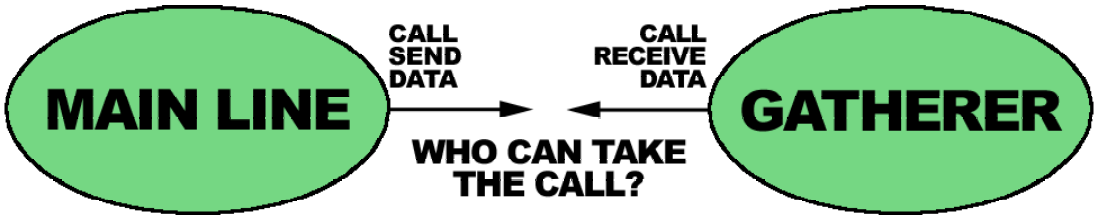
Consider this.

1. Open the tape device;
2. Read blocks 0 to 31 and pass them to the Select Directory Entry process;
3. Repeat:
    a. For each remaining tape block:
        i. Read the block from the tape;
        ii. Broadcast the block to the block-gathering sub-processes
    b. Rewind the tape;
4. Until all the data blocks have been written to the disk file;
5. Close the tape device.

However, we have a problem, the main line is in client mode and so are the gathering sub-processes.

5.16 Resolving the Client-Client Standoff

I introduced the concept of client-mode and server-mode processes in my discussion of the Traffic Lights problem. We will now consider the subject a little further.



CLIENT-CLIENT STANDOFF

The executable part of most application programs is written in client mode. When the program needs to communicate with an external entity, for example a terminal, the program executes a read or write system call to the operating system. It then waits until the information is received or its transmission is assured.

The operating system, on the other hand is usually in server mode. It waits in an idle state until it is called by an application. It then performs the requested task, returning control to the application and reverting to its idle state when the task is finished.

A good example of a simple server-mode process is a stand-alone function that does not reference any global, static or class instance variables. It waits in an inactive state until it is called; it then does whatever it is that it does, returning control back to its caller at the end of the sequence.

A more advanced example of a server-mode process might be a class instance with a number of methods. For example, an instance of a class that reads a file. The class would be instantiated on an as-required basis by a client-mode application. It would then wait in the open, but idle, state until the application needs to read data from the file. Its read method could then be called to fetch the required data.

It is usually easiest to visualise processes in client-mode. But it is difficult with conventional programming languages to get two client-mode processes to communicate with each other.

Take the example of the main line and the Data Block Gatherer in the Old Tape problem. The main line would like to read a block from the tape and then send the block to the Data Block Gatherer, preferably by calling a method. On the other side, the Data Block Gatherer would like to receive tape blocks by calling a function similar to the `read` system call. Both processes would like to call a method in the other, but neither process in a state where a suitable method can be called. It's a client-client standoff.

There are simple and complicated ways of resolving the standoff. Depending on the situation, we might have to use either. The simple way, which we will use here, is simply to convert one of the processes to server-mode by transforming the top-to-bottom representation of the algorithm into an event-driven finite state machine.

If both processes are complex, with many nested subroutines, or if one process is already written in client-mode and the other is complex, for example if the processes are Winsock and an HTTP parser, then it may be necessary to link the two processes via a co-routine linkage. Co-routine linkage is a big topic all by itself. I will have to leave it for another day, though I would be please to come back and talk about it.

Many multi-tasking operating systems have features similar to the Unix pipe that allow concurrent client-mode processes to communicate with each other. Such arrangements typically turn out to be co-routine linkages hidden in the operating system. Nevertheless, sometimes they can resolve client-client standoffs without introducing too much entropy into the application.

But fortunately, in most cases it is easy to convert one of the client-mode processes to server mode. Indeed this is the preferred alternative because it avoids the inherent complexity of co-routines, multiple threads or processes, inter-process communication, shared memory, semaphores and so-forth.

5.17 Gathering Blocks in Server Mode

Each sub-process for gathering data blocks is easily converted into server mode. Consider this state transition table.

State	Event	Processing
Uninitiated	Initiate	State = block number unknown.
Block number unknown	Incoming data block number	Store the data block number; State = block number known.
Block number known	Incoming tape block.	If the block number of the incoming tape block is the same as the stored data block number: Write the block to the Recovered Disk File; State = data block loaded; End if.
Data block loaded	Anything	Do nothing.

A similar transformation can be applied to the sub-processes for gathering the pointer and pointer-pointer blocks.

From here to a working program is only a matter of translating the pseudo-code into your favourite high-level language.

5.18 Programming Practicalities

The data gathering sub-processes could easily have been programmed as object-oriented classes with methods for handling each of the events. However this would have created a problem on some machines.

We were restricted to 500 kilobytes of data. Each data gatherer needs a state variable that can store three different states as well as a 4-byte block number. The state variable will fit in a single byte and the block number takes 4 bytes. A little arithmetic determines that a single file can use a maximum of 66,051 data and pointer blocks. 5 x 66,051 = 330,255. Technically it will all fit.

However, there's a problem. E-Genting's Unix machine, and many others, inserts three filler bytes in a structure containing one char and one long variable to maximise bus bandwidth utilisation and avoid odd address traps. 8 x 66,051 = 528,408. Oops!

The solution is to store the process states and the block numbers in separate arrays and thereby avoid the word alignment problem.

Given that data of the gathering sub-processes are stored in separate arrays there is little point in programming a class definition to access it. In practice the sub-process methods can simply be programmed in-line in what constitutes the main line.

I should also mention that the last blocks, at data, pointer and pointer-pointer level, might all be partially full. A practical program needs to make appropriate arrangements for dealing with the block fragments.

5.19 How Many Passes?

So how many passes were actually required?

1. The first pass can gather the directory entry and any pointer-pointer block, thus obtaining all the pointer block numbers.
2. A second pass can gather all the pointer blocks, thus obtaining all the data block numbers.
3. A third pass can gather all the data blocks.

At most three passes are required. If the file is small or if the layout of the file on the tape is such that the pointer blocks are positioned before the data blocks, fewer passes may be needed.

5.20 Summary

The Old Tape problem was made considerably more difficult by the temptation to go hunting when the conditions demanded that one go gathering. Yet this is a very common situation in practical computer programming. For example, should one attempt to gather all the information one needs in a single pass of a large database table or should one go hunting for the information via indices? The answer is not always obvious.

Hunt or gather, client-mode or server-mode, pseudo-code or state transitions? Such is the life of a programmer.

The importance of making the right decisions should not be under-estimated. An unrecognised mistake usually leads to a deadlock in the design process and unexpected and costly delays.

So what did the Old Tape problem teach us?

1. Analysis of a simple predefined data structure (i.e. the organisation of blocks on the tape);
2. Byte order conversion;
3. Deciding between hunting and gathering;
4. Division of a large problem into manageable processes with a data flow diagram;
5. Rounded-up integer arithmetic;
6. Representation of the processes on the data flow diagram in pseudo-code;
7. Resolution of a client-client standoff;
8. Conversion of pseudo-coded client-mode processes into server-mode finite state machines;
9. Dealing with an assortment of minor practicalities.

The first time I gave this lecture, we were at Genting Highlands and had a dinner organised for the contestants. At that time my punch line was that having worked on the problems for so long we should avoid the whole question of hunting and gathering by being waited on at the Genting Palace Restaurant.

Right now, the SARS crisis is hitting us rather badly and we would very much appreciate your patronage.

Thank you, and have a good evening.