

The projects described in this section were designed as laboratory work for students studying from various parts of the book. The projects have all been tested in our own teaching at Queen Mary and Westfield College, but laboratory work inevitably has local dependencies. We have done a quick pass through the projects to remove specific details that are obviously local, but we cannot guarantee that all local dependencies have been eliminated. Teachers should scan the projects very carefully for problems that may arise when students attempt them in their local environment. It might be wise to regard these descriptions as drafts for project sheets, to be adjusted before use to suit the local environment and students' knowledge.

The projects are outlined below:

Projects 1 and 2 are designed to exemplify and explore some of the issues raised in Chapter 6. Ideally, they should be undertaken before the study of Chapter 6. They require an understanding of the basics of operating systems and interprocess communication. We allow two afternoons of laboratory working time for each of these projects.

Project 3 consolidates the basic knowledge about RPC implementation in Chapter 4 and 5 and gives students experience in constructing programs that utilize UNIX sockets (with UDP). We allow 3–4 afternoons for it.

Project 4 should be undertaken by students who have studied the description of the use of process groups for replication in Chapter 11. It depends upon the use of ISIS for group communication. It is a useful introduction to the ISIS system. We allow students to work in groups, and they require several afternoons to complete it.

Projects 5 and 6 are concerned with the use of idle workstations to perform computations in parallel. Project 5 is loosely based on the Linda model for parallel computation and consolidates the knowledge learned in Chapter 12 about synchronization in servers. As described it requires the ANSA Testbench implementation of RPC (described in Section 5.4). It would be difficult to port the project to Sun RPC or some other simple RPC package. It requires several afternoons. Project 6 uses a computation model based on a master and workers. It is based on UNIX sockets for interprocess communication (with UDP).

Projects 7–10 consolidate knowledge about clients and services from several chapters. They develop and refine an associative database service, progressively adding features such as access control and recoverability. They provide extensive experience in defining and implementing modular services and their interfaces. We describe them in terms of Sun RPC, but any RPC package should provide an adequate software environment for these projects. We allow 1–2 weeks for each of these four projects.

Several C source code files are available in directory *projects/source* on our FTP server (see book page xiv for details on how to access the server). These files correspond to the code included in the appendices to projects 3, 4 and 5.

- C What accounts for variations in the figures you obtain for *getpid*?
- D How did you obtain a single timing figure from the varied measurements?

3 Measuring Process Creation Time

- There is only one way to create a new process under UNIX: the *fork* system call. Do *man fork* to find out about it. Do *man exit* to find out how a process can terminate itself. Do *man wait* to find out how a parent can await its children's termination.
- Write a program which uses *fork* to create as many processes as the system will allow and then awaits their termination. Make sure that each new process immediately calls *exit*. Run your program a suitable number of times and record (here) the values obtained.

A How many processes were created in each experiment?

B How long, in milliseconds, does each *fork + wait* take? Careful: design a different experiment to the one you just produced. What is the accuracy of your answer?

- Do *man exec*, to find out how to run a specified program in the calling process. You will find that various forms of *exec* exist. A simple form is *execv*, which requires a full path name for the program binary, and an *argv* pointer. Note that *argv* cannot be NULL (0).
- Create a very simple program which immediately exits. Call the resultant binary *fred*, say.
- Produce a modified version of the *fork+wait* program, so that each child process *execs* *fred*. **Attach this program.** Run this program several times and record the values obtained.

C How do you account for the time difference between this case and the last experiment?

D You probably found that the first measurement in the *exec* experiment was larger than subsequent ones. Why should that be?

4 Exploring the Address Space

- Do *man size* to find out about this command. Do *size fred* for some executable *fred*. Note that the program text starts at address 0x148, whilst the data and bss sections (respectively, initialised and uninitialised data) reside contiguously at a higher address. The other range of addresses we shall consider is that covered by the program's stack.

- Do *pagesize* to print the size of a page – the smallest unit of mapped memory.

- Input the following simple program, compile it and do *size* on the binary.

```
static char*aStaticString = "fred";
static int aStaticDataItem;
main(int argc, char *argv[])
{
    unsigned char* cp;
    unsigned char aCharacter;

    cp = &aCharacter;
    printf("address of character on stack = 0x%x\n", cp);
    printf("address of initialised string = 0x%x\n", aStaticString );
    printf("address of uninitialised int = 0x%x\n", &aStaticDataItem);
    printf("address of procedure main = 0x%x\n", main);
}
```

- A** Apart from the address of *aCharacter*, account for the address values printed by this program, compared with those printed by *size*.

- B** The workstations use a Motorola 68030/68040 processor, with a 32-bit address range. What do you think is the top of the stack's address range?

- Add the following code to the program above, to find out the limits on the stack:

```
for(cp = &aCharacter; ; cp -= PAGESIZE) { /* you need to #define PAGESIZE */
    aCharacter = *cp;
    printf("made it to 0x%x\n", cp);
}
```

- C** You should find that this program gives a 'segmentation violation' after a small number of loop iterations. What has happened?

- D** Unix can grow stacks automatically. Why didn't it in this case?

- E** Write a program to walk the whole address space of a process, from address 0. The program should report address ranges in which read accesses to memory are possible. **Attach a listing**, and briefly explain here what you found by running the program. Hints: An attempt to access an address outside the address space causes a segmentation violation exception (SIGSEGV: do *man signal*). Your program may take a long time; consider how crude you can make the search.

Record your answers on this paper

NB Some of the following experiments require two dedicated machines for meaningful results to be gathered. Since the number of machines available is limited, it is suggested that a machine is kept free by mutual agreement, and is reserved by individuals for 10-20 minute time slots whilst they perform their experiments. In the early debugging stages, when correctness is being tested rather than measurements taken, such mutual exclusion will not, of course, be necessary.

1 In-process invocation

- Describe the experiments you performed to reach results for the following two questions: the number of repeats of the operation being measured, the number of the experiments, how you selected the final result, and its accuracy.

A How long does a procedure call with no arguments and no body take?.....

B How long does a procedure call with 10 arguments and no body take?.....

2 Inter-process, intra-machine invocation

- Create a client program and a server program, described in the following. Use some sensible port number for the server, unique to yourself.

Write a program which, when given "client" as an argument does the following repeatedly: calls *sendto* with a dummy data argument of size *datasize* (given as a second program argument), followed by *recvfrom* with a 4-byte data argument.

When given "server" as an argument it is to repeatedly call *recvfrom* with a data argument of size *datasize* (given as the second program argument), then make a reply using *sendto* with a 4-byte data argument. Note that this server does not perform any useful processing upon the arriving 'requests'. Nor does it do a lookup to identify a (dummy) procedure or 'method' to call.

- Run a client and server together AT THE SAME MACHINE, and test that they work together.

- A** How long does a single request-reply interaction take when *datasize* is a) 4 bytes?.....
b) 1024 bytes?

State the accuracy of your answers, and how you arrived at them from the measurements.

- B** How do you think the elapsed time for a local request-reply interaction is spent? Give a qualitative answer.

3 Inter-machine invocation

- Run the server program at a dedicated workstation and the client program at your own workstation

- A** How long does a single request-reply interaction take when *datasize* is a) 4 bytes?
b) 1024 bytes?

State the accuracy of your answers, and how you arrived at them from the measurements.

- B** *Sendto* uses UDP. This places an 8-byte header onto an IP packet, which in turn uses a 20-byte header. An Ethernet packet carries 18 bytes of its own header and checksum. The minimum Ethernet packet size is 64 bytes. The Ethernet bandwidth is 10Mbits/sec.

What is the total hardware transmission time (time "on the wire") for the packets in a single remote request-reply interaction? a) 4 bytes b) 1024 bytes.....?

How do you think the rest of the elapsed time for a remote request-reply interaction is spent? Give a qualitative description, in terms of differences to your answer to 2B.

4 Inter-machine asynchronous invocation

- Adapt your previous program, to produce the following. As a client, the program repeatedly calls *sendto* – but not *recvfrom*. As a server, it repeatedly calls *recvfrom* – and does not reply. Make the server report if any messages are lost.
- Run a client and server at separate, dedicated machines. **NB** You may find that *sendto* fails for lack of buffer space after a certain number of sends. Try to accommodate this by reducing the number of attempted sends each time.

A Do messages get lost?

If so, how? Under what conditions did you find losses?

How did you detect losses?

B How long does each client request take if *datasize* is a) 4 bytes..... b) 1024 bytes.....?

5 TCP and Connected Sockets

- Do **man netstat**, to find out how to show per-protocol packet statistics.
- Adapt your previous program so that the client and server use a single pair of connected sockets for **request-reply** communication.

NB It is a good idea to use the call:

```
{int option = 1;
 setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &option, sizeof(option));
}
```

before calling *bind* on the socket you will use to accept a connection. This avoids a problem of your program not working because the bound address otherwise stays “in use” for a considerable time after the program has terminated.

A How long does a single request-reply interaction take when *datasize* is a) 4 bytes.....? b) 1024 bytes.....?

B Use *netstat* to count the number of packets involved in a request-reply exchange. How do you think the elapsed time for a remote request-reply interaction is spent? Give your answer in terms of whether/how it differs from your answer to 3B, and in particular refer to the fact that TCP is a reliable protocol.

6 TCP for Request/reply Interactions?

- A** If you were designing an RPC system under Unix, would you use connected or unconnected sockets? Why? What are the merits of the other choice?
- B** If the server takes a long time to reply, then does TCP send additional "are you still there?" or acknowledgement messages? Describe an experiment performed to check whether/how the TCP protocol actions vary with the amount of processing performed by the server. Comment on your results. **Attach your program and any related information.**

Introduction

This sheet contains a set of exercises that are intended to lead you through the steps necessary to master the use of UNIX datagram sockets to build a form of remote procedure call. You should use ANSI standard C function prototypes.

Preamble

Before working on the coursework itself, you should perform the following exercise, which is not examined. It involves compiling and running an existing sockets program. Study it carefully. Like the subsequent programs you are asked to write, it uses UDP (not TCP) sockets. You may borrow any code you feel necessary from this program, such as *printSA*, *MakeDestSA*, *MakeLocalSA*, *MakeReceiverSA*, *anythingThere* and use them as utilities in the following exercises. Also note the 'include' files needed and the function prototype for *gethostbyname*.

Warm-up exercise.

Take a copy of the file *UDPsock.c* from the Appendix. This file comprises a C program. It can be run either as a "sender" - a process that calls the *sender* procedure or as a "receiver" - a process that calls the *receiver* procedure. You should compile it and then run it as follows:

Use "rlogin" to log into another computer and run the program as a "receiver" by giving it the argument "r". (First get permission from the user of the other computer!)

Run the program as a sender by giving it the four arguments: "s", the name of the machine where you are running the receiver and two messages. The program prints out the socket address used and the messages received.

Definitions

In these exercises you are to use the following type definitions:

```
#define SIZE 1000
typedef struct {
    unsigned int length;
    unsigned char data[SIZE];
} Message;
typedef enum {
    OK,          /* operation successful */
    BAD,         /* unrecoverable error */
    WRONGLENGTH /* bad message length supplied */
} Status;
typedef struct sockaddr_in SocketAddress ;
```

Exercise 1: A server that echoes client input

You are required to produce client and server programs based on the procedures *DoOperation*, *GetRequest* and *SendReply* (see Chapter 4). The client and server behave as follows:

Client: this takes the name of the server computer as an argument. It repeatedly requests a string to be entered by the user (use *gets*), and uses *DoOperation* to send the string to the server, awaiting a reply.

Server: repeatedly receives a string using *GetRequest*, prints it on the screen and replies with *SendReply*. The server exits when the string consists of the single character 'q'.

You are to implement *DoOperation*, *GetRequest* and *SendReply*. Their prototypes (to which you must adhere) are as follows:

```
Status DoOperation (Message *message, Message *reply, int s, SocketAddress serverSA);
Status GetRequest (Message *callMessage, int s, SocketAddress *clientSA);
Status SendReply (Message *replyMessage, int s, SocketAddress clientSA);
```

DoOperation (&callMessage, &replyMessage, s, serverSA) sends a message *callMessage* to socket address *serverSA* via a socket with descriptor *s* and blocks until it returns with *replyMessage*. The *Status* value returned will reflect the values returned by *UDPsend* and *UDPreceive* (see below).

GetRequest (&callMessage, s, &clientSA) receives a message *callMessage* via a socket with descriptor *s*; the caller's socket address is received via the variable *clientSA*. The *Status* value returned will reflect the value returned by *UDPreceive* (see below).

SendReply(&replyMessage, s, clientSA) sends a message *replyMessage* via a socket with descriptor *s* to the client's socket address *clientSA*.

UDPsend and UDPreceive

The procedures *DoOperation*, *GetRequest* and *SendReply* must utilise two procedures *UDPsend* and *UDPreceive* to be written by you, which respectively send and receive a message over/from a socket. You are to implement these functions using the system calls *sendto* and *recvfrom* (and any others you think necessary). Their prototypes are:

```
Status UDPsend(int s, Message *m, SocketAddress destination);
Status UDPreceive(int s, Message *m, SocketAddress *origin);
```

UDPsend(s, &m, destination) sends the message *m* through the socket with descriptor *s* to socket address *destination*.

UDPreceive(s, &m, &origin) receives a message through the socket with descriptor *s* into buffer *m* and puts the socket address of the sender into *origin*.

Each procedure returns a value of type *Status* which reports on the success of its execution. For example, if the *sendto* or *recvfrom* system calls return negative values, your procedures should return a *Status* value of *BAD*.

Choosing a server port

You will want to run *server* processes that can coexist with other people's processes in the same computer. You need to select an agreed port number for the server to receive messages from clients. Two servers on the same computer cannot use the same local port number. You will therefore want to choose a port number that is sure to be different from other people's port numbers. If everybody takes the first unreserved port number and adds their *uid*, there should be no such clashes - i.e. :

```
aPort = IPPORT_RESERVED + getuid();
```

What to demonstrate

Run your two programs on different computers and make them work correctly. The server should still work correctly if you have two clients on different computers.

You should also report whether you managed to make the server drop any messages, and what experiment you performed to test this. Include a short write-up as a comment in the code. [This counts for 25% of the marks for this exercise].

Exercise 2: An arithmetic server using RPC

In this exercise you will create an adaptation of your last program(s), so that the strings that users type to the clients are arithmetic expressions; and the server evaluates each arithmetic expression and returns the results. Communication is to be by RPC. Client and server are as follows:

Client: Each line typed at the client is to be interpreted as a simple arithmetic operation (e.g. 34+67, 89*54, etc.). This requires the expression to be separated (e.g. using *scanf*) into an operation (+, -, *, /) and two non-negative integer arguments.

Server: This must implement the four procedures *add*, *subtract*, *multiply* and *divide* with identical prototypes, e.g.

```

Status divide(int x, int y, int *z)
{
    if y == 0 return DivZero;
    else {
        *z = x/y;
        return OK
    }
}

```

You should use the following definition of an RPC message:

```

typedef struct {
    enum {Request, Reply} messageType;    /* same size as an unsigned int */
    unsigned int RPCId;                  /* unique identifier */
    unsigned int procedureId;            /* e.g.(1,2,3,4) for (+, -, *, /) */
    int arg1;                            /* argument/ return parameter */
    int arg2;                            /* argument/ return parameter */
} RPCMessage;                          /* each int (and unsigned int) is 32 bits = 4 bytes */

```

The fields *arg1* and *arg2* can be used for operation arguments or returned value and status. Note that no “sourceport” field is necessary, because the return address is handled by *UDPreceive*.

This exercise requires you to write *marshalling* and *unmarshalling* procedures:

```

void marshal(RPCmessage *rm, Message *message);
void unMarshal(RPCmessage *rm, Message *message);

```

marshal(&RPCm, &m) flattens *RPCm* into *m.data* and puts its length into *m.length*. Marshalling is very simple because each element of the structure is an integer – and will occupy 4 bytes of *m.data*. Note that you should take network ordering into account, and therefore should apply the function *htonl* to each integer (or unsigned integer) before flattening it.

unMarshal(&RPCm, &m) unflattens *m.data*, taking each successive 4 bytes as an integer belonging to the struct *RPCm*. To take network ordering into account, you should apply the function *ntohl* to each integer (or unsigned integer) after unflattening it.

Two other matters should be addressed:

- both client and server should make use of the *messageType* field of the *RPCmessage* structure after unmarshalling to test that Requests and Replies are not confused;
- the client should generate a new *RPCId* for each call; and the server should copy the *requestId* from the Request message to the reply message so that the client can test that the replies correspond to the requests.

What to demonstrate

The result should be an arithmetic server that performs arithmetic operations for several clients, and which behaves sensibly when dealing with exceptional conditions.

Exercise 3: Adding a timeout

Add a timeout in your client program. This should have the effect that if there is no response from the server for several seconds after sending the call message, the client resends the request for up to a small, fixed number of times. This is in case a message was dropped, or the server has crashed. The client should report on its behaviour in these circumstances.

Timing-out can be done by using the *select* system call to test whether there is any outstanding input on the socket before calling *UDPreceive*. The procedure *anythingThere* in the example program shows how to do this. You can test your time-out by running the client when the server is not running.

Appendix

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>

struct hostent *gethostbyname();
void printSA(struct sockaddr_in sa);
void makeDestSA(struct sockaddr_in *sa, char *hostname, int port);
void makeLocalSA(struct sockaddr_in *sa);
void receiver(int port);
void sender(char *message1, char *message2, char *machine, int port);

#define RECIPIENT_PORT 1234
#define SIZE 1000

/* main for sender and receiver - to send give s machine message1 and message2
   - to receive give r
*/
void main(int argc, char **argv)
{
    int port = RECIPIENT_PORT;

    if(argc <= 1){
        printf("Usage:s(end) ...or r(eceive) ??\n");
        exit();
    }
    if(*argv[1] == 's'){
        if(argc <=2) {
            printf("Usage: s machine message1 message2\n");
            exit();
        }
        sender(argv[3], argv[4], argv[2], port);
    } else if(*argv[1] == 'r') receiver(port);
    else printf("send machine or receive ??\n");
}

/*print a socket address */
void printSA(struct sockaddr_in sa)
{
    printf("sa = %d, %s, %d\n", sa.sin_family,
        inet_ntoa(sa.sin_addr), ntohs(sa.sin_port));
}

/* make a socket address for a destination whose machine and port
   are given as arguments */
void makeDestSA(struct sockaddr_in *sa, char *hostname, int port)
{
    struct hostent *host;

    sa->sin_family = AF_INET;
    if((host = gethostbyname(hostname)) == NULL){
        printf("Unknown host name\n");
        exit(-1);
    }
    sa->sin_addr = *(struct in_addr *) (host->h_addr);
    sa->sin_port = htons(port);
}
```

```

/* make a socket address using any of the addresses of this computer
   for a local socket on any port */
void makeLocalSA(struct sockaddr_in *sa)
{
    sa->sin_family = AF_INET;
    sa->sin_port = htons(0);
    sa-> sin_addr.s_addr = htonl(INADDR_ANY);
}

/* make a socket address using any of the addresses of this computer
   for a local socket on given port */
void makeReceiverSA(struct sockaddr_in *sa, int port)
{
    sa->sin_family = AF_INET;
    sa->sin_port = htons(port);
    sa-> sin_addr.s_addr = htonl(INADDR_ANY);
}

/*receive two messages via s new socket,
   print out the messages received and close the socket
   bind to any of the addresses of this computer
   using port given as argument */
void receiver(int port)
{
    char message1[SIZE], message2[SIZE];
    struct sockaddr_in mySocketAddress, aSocketAddress;
    int s,aLength, n;

    if((s = socket(AF_INET, SOCK_DGRAM, 0))<0) {
        perror("socket failed");
        return;
    }
    makeReceiverSA(&mySocketAddress, port);
    if( bind(s, &mySocketAddress, sizeof(struct sockaddr_in))!= 0){
        perror("Bind failed\n");
        close(s);
        return;
    }
    printSA(mySocketAddress);
    aLength = sizeof(aSocketAddress);
    aSocketAddress.sin_family = AF_INET;
    if((n = recvfrom(s, message1, SIZE, 0, &aSocketAddress, &aLength))<0)
        perror("Receive 1");
    else{
        printf("Received Message:(%s)length = %d \n",
            message1,n);
    }
    if((n = recvfrom(s, message2, SIZE, 0, &aSocketAddress, &aLength))<0)
        perror("Receive 2");
    else {
        printf("Received Message:(%s)length = %d \n",
            message2,n);
    }
    close(s);
}

```

```

/*do send after receive ready, open socket
   bind socket to local internet port (use any of the local computer's addresses)
   send two messages with given lengths to machine and port
   close socket
*/
void sender(char *message1, char *message2, char *machine, int port)
{
    int s, n;
    char message[SIZE];
    struct sockaddr_in mySocketAddress, yourSocketAddress;

    if(( s = socket(AF_INET, SOCK_DGRAM, 0))<0) {
        perror("socket failed");
        return;
    }
    /*
    if((x = setsockopt(s, SOL_SOCKET, SO_BROADCAST, &arg, sizeof(arg))<0)
        perror("setsockopt SO_BROADCAST---");
        exit(-1);
    */
    makeLocalSA(&mySocketAddress);
    if( bind(s, &mySocketAddress, sizeof(struct sockaddr_in))!= 0){
        perror("Bind failed\n");
        close (s);
        return;
    }
    printSA(mySocketAddress);
    makeDestSA(&yourSocketAddress,machine, port);
    printSA(yourSocketAddress);
    strcpy(message,message1);
    if( (n = sendto(s, message, strlen(message), 0, &yourSocketAddress,
        sizeof(struct sockaddr_in))) < 0)
        perror("Send failed\n");
    if(n != strlen(message)) printf("sent %d\n",n);
    strcpy(message,message2);
    if( (n = sendto(s, message, strlen(message), 0, &yourSocketAddress,
        sizeof(struct sockaddr_in))) < 0)
        perror("Send failed\n");
    if(n != strlen(message)) printf("sent %d\n",n);
    close(s);
}

#include <sys/time.h>
/* use select to test whether there is any input on descriptor s*/
int anythingThere(int s)
{
    unsigned long read_mask;
    struct timeval timeout;
    int n;

    timeout.tv_sec = 10; /*seconds wait*/
    timeout.tv_usec = 0; /* micro seconds*/
    read_mask = (1<<s);
    if((n = select(32, &read_mask, 0, 0, &timeout))<0)
        perror("Select fail:\n");
    else printf("n = %d\n",n);
    return n;
}

```

Overview

Your task is to create a program that can act as both a bulletin board (BB) server and a client 'reader' program, as informally specified below. Users can view bulletin board items, post items and reply to items. The BB service reliably stores the last N messages *in volatile memory*.

Bulletin board items

There is only one bulletin board. We shall only concern ourselves with item identifiers, sender names, dates and subjects – you can assume that the text (i.e. body) of the item is kept somewhere, but you don't need to implement it. BB items are therefore of the form:

<id: integer, sender: string, date: integer, subject: string>.

If a reply is posted, its subject should be "re:....."

A message which is "re:" another message should always appear after its referent.

Persistence

You should keep up to the N (100, say) most recent BB items in memory, and you can throw others away. However, the most recent items should continue to be available even though there are no BB readers at certain times, by using background servers.

Availability

The service should be able to stand the loss of any one machine. If a state is reached in which only one machine holds the data, then this should be reported. A system administrator can then start up another server.

Clients

Users are, by default, shown the last *m* messages, but they can scroll back to see other message still stored by the service. All newly arrived messages are immediately displayed.

Notes

You will need the *Notes on Programming in ISIS* (Appendix A) and Chapter 4 of the ISIS User's Guide and Reference Manual. See also Chapter 11.

An example ISIS program is given in Appendix B.

Suggested user interface: Simple, using *curses* library.

Notes on Programming in ISIS

These notes introduce programming in ISIS. They cover only the most basic programming primitives. They assume that the reader is familiar with the concept of process group, and of multicasting to a process group. The reader is also referred to the ISIS User's Guide [1]. **Warning:** the User Guide gives incorrect definitions for certain ISIS functions. If in doubt, look at the prototypes in the include file `isis.h`. The differences consist of the types of the arguments, and the addition of an extra argument to some functions.

Process Groups and Multicasts

A running ISIS program consists of one or more groups of processes which can communicate with one another by multicasting messages. An ISIS group is given a character string name, supplied when the group is created. Once it has been created, processes can request to join or leave the group. Processes wishing to multicast messages to the group can look up the group by providing its name. The run-time system returns an address which can be used with a multicast primitive.

All three multicast types `ABCAST`, `CBCAST` and `GBCAST` are available.

A multicast message consists of a) an *entry number* and b) a list of typed data. The entry number serves to distinguish between different types of message arriving at a given process. In response to a multicast message, recipient processes are able to send a reply. A reply message just consists of a list of typed data.

Tasks and Entries

An ISIS process can be conceived of as being event-driven. The programmer writes a main program whose main purpose is to declare the *tasks* that the process is to run. A task is a thread executing a C function supplied by the programmer in a declaration. Each task is associated with a type of event which causes the task to be run. Tasks are run repeatedly, whenever an event of the corresponding type occurs. There are also event handlers called *watches* which handle only a single event, but we shall not discuss these here.

The most common type of event is the arrival of a multicast message. When a multicast message is delivered to a process, the ISIS run-time system dispatches a task to handle it. A task which is run when a multicast message is received is also called an *entry*. The system chooses a C function for the task to run according to the arriving message's entry number. The programmer associates the function with a given entry number through the call `isis_entry` ([1] p. 98):

```
isis_entry(int entry_no, void (*func)(message *), char *name);
```

Where *name* is a character string name corresponding to the function *func*. When ISIS runs an entry, it supplies a pointer to the message whose arrival caused it to be run as the single argument of *func*.

Tasks that are not entries also have to be declared in the main program in a separate call `isis_task` ([1], p. 131):

```
isis_task(void (*func)(void *arg), char *name);
```

Again, *name* is chosen to correspond to *func*. *func* takes a single argument, specified in a separate call. The programmer specifies the first task to be run to initialise the process by calling `isis_mainloop` ([1], p. 131):

```
isis_mainloop(void (*first_task)(void *arg), void *arg0);
```

The argument *arg0* is given to *first_task*. After *first_task* returns (or, after it calls `isis_start_done()`), ISIS automatically chooses tasks to run. ISIS does not run tasks pre-emptively. It can only run a different task when the current task yields. A task normally yields by returning from its main function body, but there are other ways of yielding, such as making a blocking multicast call or calling `sleep()`.

Tasks run together in a single Unix process, whose address space they share. They each have their own stack, which is of limited size (32 K by default) and cannot be dynamically extended. If a task runs over the end of its stack, other areas of memory will be trashed, but this may not be detected by ISIS.

Joining and Leaving a Group

Another major type of event recognised by ISIS is that of group membership changes. Processes can join or leave a group.

A process can simultaneously join a group and register a task to be run when group membership changes. The call *pg_join* ([1], p. 105):

```
address *pg_join("fred", PG_MONITOR, group_change, void *arg, 0);
```

is used to join the calling process to the group called "fred". It returns the group's multicast address. The group is created by default if it did not already exist. The option *PG_MONITOR* causes a task rooted with the void-returning function *group_change* to be run whenever a process joins or leaves the group. It is called with the argument *arg*. Note that *group_change* will be called for the event of the calling process itself joining the group.

ISIS ensures that group membership changes are seen in the same order by all processes. Moreover, no multicast sent to a group will be seen before a membership change by some group member and after it by another.

To find out the new group membership after a change, the call *pg_getview* ([1], p. 112):

```
groupview *pg_getview(address gaddr_p);
```

returns a pointer to a structure which contains a list of the members' addresses. The boolean function:

```
addr_ismine(address gaddr_p);
```

can be used to identify the caller in this list.

Multicast Primitives

There are distinct calls *abcast*, *cbcast* and *gbcast* corresponding to the multicast types *ABCAST*, *CBCAST* and *GBCAST*. The general form of a multicast call is ([1], p. 100):

```
a/c/g/bcast(address *addr_p, u_char entry, char *fmt1, arg1, arg2, ...,
            nwanted,
            char *fmt2, rep1, rep2, ....);
```

addr_p is the address of either an individual process or of a process group. The address of a process group can be obtained without joining the group through a call ([1], p. 111):

```
address *pg_lookup(char *group_name);
```

In the *bcast* call:

entry is the entry number of the multicast message.

fmt1 and *fmt2* are used to marshal data into the outgoing message, and to unmarshal data from the reply messages, respectively. They are like the format strings used in *printf()* and *scanf()*. *fmt1* is used to specify the types of the data items to be placed in the multicast message; the data values are those of the corresponding arguments *arg1*, *arg2*,... *fmt2* specifies the types of the data items expected in each reply message. The received values are to be stored in the arrays *rep1*, *rep2*, etc.

nwanted is the number of replies required. Each member of the group can reply. The arrays *rep1*, *rep2*, .. must be able to hold at least *nwanted* items. The data received from the first reply will be {*arg1*[0], *arg2*[0], ..}; that of the second will be {*arg1*[1], *arg2*[1], ..}. If *nwanted* is 0, *fmt2*, *rep1* etc. may be omitted.

The return value of a multicast call is the number of replies actually received.

As an example,

```
cbcast(....., "%d %d", i, j, 2, "%c", reply_array);
```

specifies an outgoing message of two 32-bit integers with values taken from the variables *i* and *j*. Two replies are expected. Each reply is to be a single character; the reply data is to be stored in the array *reply_array*.

To reply to a message, there is the call *reply* ([1], p. 100):

```
reply(message *in_msg, char *fmt1, arg1, arg2, ..);
```

in_msg is the multicast message being replied to (obtained as the argument to the entry function).

fmt1, *arg1* etc. are used as in the multicast primitives to specify the data of the reply message. The string *fmt1* of this call should match the string *fmt2* of the multicast call which will receive the reply.

Each multicast call has a long form: *abcast_l*, *cbcast_l* and *gbcast_l*. These take an additional option argument as their first argument. One useful option is for the calling process not to receive a multicast, even though it itself belongs to the multicast group. This can be used only for multicasts which are asynchronous – i.e. which do not receive replies.

The calls are then:

```
a/c/gbcast_l("x", addr_p, ...);
```

Messages

The following are some of the main data types which can be specified in a format string ([1], p. 92):

```
%s    null-terminated character string.
%c    single byte.
%d    32-bit integer.
%h    16-bit integer.
```

There is a problem with the (format, arg1, arg2, ...) technique of specifying the data of a message: what if the number and possibly type of data items to be sent can only be determined at run-time? The format string is normally a constant string. Although it could be constructed dynamically, this would be awkward.

For this and other reasons of convenience, ISIS allows messages to be explicitly created and manipulated. Having created a message and inserted its dynamically determined data, we enclose its data in another message using the format string "%m". The following is an example of this:

```
cbcast(gaddr_p, ENTRY1, "%m", msg, ....);
```

If a reply is to be made with data from a message separately prepared, then the long form of the reply primitive must be made:

```
reply_l("m", message *in_msg, message *reply_msg);
```

Creating and Manipulating Messages

Messages are created using *msg_newmsg* ([1], p. 893):

```
message *msg_newmsg(void);
```

They must be freed explicitly using *msg_delete* ([1], p. 90):

```
msg_delete(message *msg);
```

Data can be inserted into a message using *msg_put* ([1], p. 89):

```
msg_put(message *msg, char *format, arg1, arg2..);
```

– which appends the data items specified using (format, arg1, ..) after any already stored in *msg*.

Data can be read from a message using *msg_get* ([1], p. 90):

```
msg_get(message *msg, char *format, arg1, arg2, ..);
```

– which reads data from *msg* and stores it in areas pointed to by *arg1* etc. Multiple calls to *msg_get* can be attempted using the same message: the next data items, if any, are read each time.

Input/Output

If a task calls a function which blocks awaiting input from the terminal, like *getchar()* or *scanf()*, then the entire process will be blocked. No incoming messages will be processed.

ISIS therefore allows the programmer to declare an input handler task. This is a task which is run whenever data is available for input on a given UNIX file descriptor. The call is *isis_input* ([1], p. 131):

```
isis_input(int file_desc, void (*function)(), void *arg);
```

arg is supplied as an argument to *function* whenever the task runs. *function* can take the value *NULLROUTINE* to cancel the effect of a previous call to *isis_input* with a non-NULL function.

Nuts and Bolts

This section concerns the nuts and bolts you need to know in order to create and run ISIS programs.

Creating an ISIS program

Your ISIS program will be divided into one or more process groups, representing a functional division of your application. For each process group, you will create a separate executable file. This will be an ordinary Unix executable, but will have the ISIS libraries linked in. The ISIS libraries are:

```
.../isis/lib/{libisis1.a,libisis2.a,libisism.a}.
```

Also, don't forget to include the header file `"/official/isis/include/isis.h"` in each of your source files.

Running an ISIS program

The ISIS execution environment must run at every host machine where you want to create a process. This is achieved by logging in to each host concerned and typing:

```
.../isis/bin/istart
```

This command should create a number of background processes. To ensure that this is successful, type:

```
.../isis/bin/cmd
```

and select the 'sites' option to see if your host has been included as an ISIS site. Try experimenting with the *cmd* command (type '?' or 'help' to list the options).

Once ISIS has been started at each host, you are ready to run your ISIS program. Simply run the appropriate application executable at each host.

References

- [1] ISIS User's Guide and Reference Manual, Chapter 4.

Appendix B

```
#include <stdio.h>
#include "/official/isis/include/isis.h"

#define TEXT_ENTRY      1
#define MAX_TEXT       40
#define MAX_GROUPSIZE  10

void maintask(void *nullarg);
void handle_text(message *);
void handle_input(void *);
void monitor_group(groupview *, int);

static char      *group_name; /* string name of group*/
static int       myrank;      /* my rank in group view list */
static int       n_in_group;  /* # of processes in group*/
static address   *group;      /* address of group*/

void main(int argc, char *argv[])
{
    /*
     * This program allows the user to broadcast messages to the others in their group.
     * Anyone can type a message at any time.
     */

    /*
     * They first have to agree on the (string) name of their broadcast group.
     * The program argument should be the group name
     */
    if(argc != 2){
        printf("syntax: %s group-name\n", argv[0]);
        exit(0);
    } else
        group_name = argv[1];
    isis_init(0);
    /*
     * Declare tasks
     */
    isis_task(maintask, "maintask");
    isis_task(monitor_group, "monitor_group");

    /*
     * Declare entries
     */
    isis_entry(TEXT_ENTRY, handle_text, "handle_text");

    /*
     * Declare input handler
     */
    isis_input(0, handle_input, NULLARG);

    /*
     * Start with main task
     */
    isis_mainloop(maintask, NULLARG);

} /* end main */
```

```

void maintask(void *nullarg)
{
    message    *mp;
    groupview  *gview_p;
    /*
     * Create group if necessary; otherwise join existing group.
     * Declare monitoring task at same time.
     */
    group = pg_join(group_name, PG_MONITOR, monitor_group, 0, 0);
    if(addr_isnull(group)){
        isis_perror("join failed");
        printf("Couldn't join/create session called %s\n", group_name);
        exit(0);
    }
    /*
     * Enable new tasks to be started.
     * (Unnecessary here since we are returning)
     */
    isis_start_done();
} /* end maintask */

void monitor_group(groupview *gview_p, int arg)
{
    /*
     * Set the static variables myrank and n_in_group.
     */
    for(myrank = 0; !addr_ismine(&gview_p->gv_members[myrank]); ++myrank)
        continue;
    n_in_group = gview_p->gv_nmemb;
    /*
     * It could be I who joined
     */
    if(addr_isequal(&gview_p->gv_joined, &gview_p->gv_members[myrank]))
        printf("I joined the group!\n");
    else
        printf("someone joined or left my group!\n");
} /* end monitor_group */

void handle_text(message *msg_p)
{
    char        text[MAX_TEXT];
    int         who;
    message     *embedded_msg;
    /*
     * Handle message received from another member of my group.
     * NB in "%+m" the "+" causes ISIS to create space for the new message.
     */
    if(msg_get(msg_p, "embedded msg=%+m", &embedded_msg) <= 0 ||
       msg_get(embedded_msg, "who=%d their text=%s", &who, text) <= 0)
        printf("error receiving text\n");
    else /* Throw away message if it is from me */
        if(who != myrank) printf("received message '%s' from %d\n", text, who);
    /* Free the embedded message */
    msg_delete(embedded_msg);
    /* Reply with my rank */
    reply(msg_p, "myrank=%d", myrank);
}

```

```

void handle_input(void*nullarg)
{
    message    *msg;
    int        ranks[MAX_GROUPSIZ];
    char       text[MAX_TEXT];
    int        nreplies;

    /*
     * Handle event that keyboard input is available.
     * Begin by getting line of text from stdin.
     */
    gets(text);

    /*
     * Prepare the message.
     * Use of an embedded message is for the sake of an example - there's
     * no real need for it in this case.
     */
    msg = msg_newmsg();
    msg_put(msg, "myrank=%d mytext=%s", myrank, text);

    /*
     * Do the broadcast.
     * Recipients should reply with their ranks (which we throw away).
     */
    nreplies = cbcast(group, TEXT_ENTRY, "embedded message=%m", msg,
                     n_in_group-1,
                     "ranks of repliers=%d", ranks);

    /*
     * Free the message we used.
     */
    msg_delete(msg);
} /* end handle_input */

```

The objective of this project is to build a simple toolkit that enables processes running in several workstations to carry out parts of a computation in parallel. The general idea is that a master process places sub-tasks of a computation in a 'Task Bag' and worker processes select tasks from the Task Bag and carry them out, returning the results to the Task Bag. The master then collects the results and combines them to produce the final result.

The idea of the 'Task Bag' comes from the Linda system which was designed by Carriero and Gelertner at Yale University, [1] and [2]. In Linda the task bag is implemented as distributed shared memory. In this project we are asking you to implement the Task Bag as a server and to use it as a basis for performing a parallel computation on several workstations.

Three types of process are involved: the Task Bag service, the master process and the worker processes. The master and worker processes are clients of the Task Bag service.

The Task Bag service

The Task Bag is a service whose functionality is to provide a repository for *Pairs*. Each *Pair* may be regarded as a **task description**. A *Pair* consists of two parts - a *Key* and a *Value*. The *Value* contains the actual description of a task and the *Key* is anything that can be used to reference the *Pair*. A typical *Key* might be a task name or number. A task description may be used by the master to describe tasks and by workers to describe results. Clients may add task descriptions to the Task Bag, remove or retrieve them from the Task Bag. To access a task description for removal or retrieval, the client must specify a *Key*.

The Task Bag service will offer the operations *Out*, *In* and *Read* in its interface. They are defined as follows:

<i>Out(Key, Value)</i>	causes a <i>Pair (Key, Value)</i> to be added to the Task Bag. The client process continues immediately;
<i>In(Key) → Value</i>	causes some <i>Pair</i> in the Task Bag that matches <i>Key</i> to be withdrawn from the Task Bag; the <i>Value</i> part of the <i>Pair</i> is returned and the client process continues. If no matching <i>Pair</i> is available, the client waits until one is and then proceeds as before. If several matching <i>Pairs</i> are available, one is chosen arbitrarily;
<i>Read(Key) → Value</i>	is the same as <i>In(Key)</i> except that the <i>Pair</i> remains in the Task Bag.

The application

You should choose one application that requires fairly intensive computation to carry it out and which is easily divided into a number of identical subtasks. You could consider tasks such as: (i) parallel compilation of the modules of a program (parallel make); (ii) searching for files containing a particular text string; (iii) finding prime numbers (iv) matrix multiplication or (v) fractal images.

Parallel Programming with the Task Bag

We consider the sort of parallel program that involves a transformation or series of transformations to be applied to all the elements of some set in parallel. This type of parallelism is suitable for modelling with the master-worker paradigm.

A master process provides a set of tasks to be done by a collection of identical worker processes. Each worker is capable of performing any one of the steps in a particular computation. In the simplest cases, there is only one step.

A worker repeatedly gets a task, carries it out and then puts the result in the Task Bag. The results are collected by the master. The program executes in the same way whether there are 1, 10 or 1000 workers.

We refer to two examples throughout this explanation. In the first example, the joint task is to generate all the prime numbers less than some limit, MAX. We use one master process together with one or more worker processes. The master process sets up the first task and then waits to collect the prime numbers calculated by the workers. Each worker process repeatedly gets a range of numbers within which to search for prime numbers. Each worker places the sets of primes it finds in the Task Bag, from whence the master may collect them.

For the second example, we consider a program that multiplies two matrices A and B. In this program one master process sets up the multiplication tasks and collects the results, generated by one or more workers. Each worker repeatedly gets an element to calculate and puts the result in the Task Bag (for later collection by the master).

How the workers know which task to do next

In many computations, there is a collection of tasks, numbered *First* to *Last*. Each worker repeatedly carries out one (or a group) of the tasks. Before a worker starts it needs to know which task to do next. A *Pair* with the key *Next Task* can be used for this purpose. The master puts in the first task:

```
Out("Next Task", First);
```

and each worker in turn takes the Pair out, increments its value and puts it back. e.g.

```
In("Next Task") → nextElement;  
Out("Next Task", nextElement + GRANULARITY);
```

The number of tasks done together is a constant (GRANULARITY). When there are no more tasks to be done, the worker does not replace *Next Task* in the Task Bag. When other workers attempt to remove it, they will block. No more work will be done until the master supplies another collection of tasks to calculate.

In the prime numbers example, the worker calculates primes within the range *nextElement* to *nextElement + GRANULARITY-1*.

In the matrix multiplication example, GRANULARITY = 1 and the worker works out the row and column of the element to calculate from the value retrieved. e.g. the elements may be numbered in order across the rows.

The workers' results

It is important to note that many workers perform similar tasks and generally return values with identical keys to the Task Bag. The Task Bag must be implemented so that many *Pairs* with the same key may be held at the same time.

In the prime numbers example, all the results calculated by the workers may bear the same key: *Result*. A worker can put a collection of prime numbers in the Task Bag as follows:

```
Out("Primes", <a collection of primes>);
```

The master just collects all the *Pairs* with the key *Primes* e.g., by:

```
In("Primes") → <a collection of primes>;
```

In some application, each worker needs to apply a different key to the results of its work. In the matrix multiplication example, each worker task consists of calculating one element of the result: the key of the result needs to indicate the row and column numbers of the element calculated. For example, if a worker has calculated an element (row, column), it will specify the number of the row and column in the result:

```
sprintf(Key, "Element %d %d", row, column);  
Out(Key, <the calculated result>);
```

Data for the workers

In some computations, the workers need data in order to perform their task. For example in the matrix multiplication task, a worker needs row *i* of matrix A and column *j* of matrix B in order to multiply them together. This data is put in the Task Bag by the master and may be accessed by workers that need it. The master can put in the rows of matrix A and the columns of matrix B as follows:

```

Out("A1", <A's first row>);
Out("A2", <A's second row>);
•
Out("B1", <B's first column>);
Out("B2", <B's second column>);
•

```

In this example, many workers will require the same rows and columns, they therefore use the *Read* operation rather than the *In* operation. A worker may for example access a particular row of A and column of B as follows:

```

sprintf(Key, "A%d",row);
Read(Key) → aRow;
sprintf(Key, "B%d",column);
Read(Key) → bRow;

```

In the calculation of prime numbers, a worker calculates whether a number, n is prime by dividing it by all the prime numbers up to \sqrt{n} . Therefore the worker needs to know the previously calculated primes up to \sqrt{n} . As the master collects the primes calculated by the workers it can put them in order and then place copies of sets of them in the Task Bag for use by the workers.

Monitoring

The above arrangement is not fault-tolerant. If a worker fails before completing a task, the master will block when it attempts to read the corresponding result. In our example, if a worker fails between removing the value of *Next* and replacing the next value, all the workers will block.

The user who starts the parallel computation should be able to monitor its progress. The monitor should report on the state of the computation and provide the ability to recover from incomplete computations.

Concurrency in the Task Bag Service

The question is: if *In* or *Read* blocks, should the blocking take place in the server or the client? If blocking in the server is chosen, the Task Bag service will need to have several threads of execution to enable other clients to use it when it is blocked in an *In* or a *Read*.

A server with concurrent threads will require synchronisation between the threads to prevent interference between operations being performed concurrently on behalf of different clients. (e.g. protect each operation with a *Mutex*).

In addition, when task descriptions are added, the service must *Signal* threads that are *Waiting* for them (i.e. blocked in *In* or *Read*).

The Appendix shows how to implement a *Mutex* and the operations *Signal* and *Wait* in terms of event counts and sequencers.

Giving in the work

Please supply us with print outs of your master and worker programs and of the Task Bag service interface. We will also require you to demonstrate the programs to us in the lab.

References

- [1] *Linda and Friends*, Ahuja, S., Carriero, N. and Gelertner. D., IEEE Computer, August 1986. pp 26-34.
- [2] *How to write parallel programs: A guide to the perplexed*, Carriero, N. and Gelertner. D., Computing Surveys, Vol. 21, No 3, Sept. 1989.

Appendix

An example of an ANSA server with threads and synchronisation

We consider a "Bounded Buffer" server as an example of a multi-threaded service. It keeps a bounded buffer of "Things". Client processes can be producers or consumers. The producers request the server to deposit the "Things" it produces in its buffer and the consumers request the server to fetch "Things" from its buffer and return them.

In this service, the producers must wait if there is no more room in the buffer. Consumers must wait if the buffer is empty. The server is implemented with multiple threads of control. When a producer or consumer cannot be served immediately, the server thread blocks. Meanwhile other clients can be serviced on other non-blocked threads. When a producer adds a "Thing" to an empty buffer, a thread blocked on behalf of a consumer will be released and can return the "Thing". When a consumer fetches a "Thing" from a full buffer, a thread blocked on behalf of a producer will be released. The interface definition of the "Bounded Buffer" service is:

```
BB: INTERFACE =
-- File : BB.idl
-- Specification for the BB (Bounded Buffer) service.
BEGIN
Thing : TYPE = STRING;
  Produce : OPERATION[x : Thing] RETURNS [];
  Consume : OPERATION[] RETURNS [Thing];
END.
```

Concurrency in a service

To request that a server have several threads, put the following in the server program:

```
#define CONCURRENCY 2 /* CONCURRENCY defines the initial number of threads */
GLOBAL ansa_Cardinal Ansa_InitialTasks = (ansa_Cardinal)CONCURRENCY;
```

This allows up to *CONCURRENCY* client calls to be serviced concurrently. The concurrency can be increased by calling the *nucleus_tasks* procedure

```
nucleus_tasks((ansa_Cardinal)1, (ansa_Cardinal) 0);
```

The first argument specifies the number of extra threads. The second argument requests a default stack size.

Synchronisation between threads

The mechanism for synchronisation between threads is based on event counts and sequencers.

An **event count** is used to record the number of times an event has occurred. It is represented by an integer initialised to zero and is manipulated by the following primitives

```
advance(ec):    Increases ec by 1.
read(ec):      Returns ec;
wait(ec, value):  Waits until ec >= value.
```

A **sequencer** dispenses unique values. It is represented by an integer initialised to zero and is manipulated by the atomic function:

```
ticket(seq)    Increases seq by 1 and returns the old value of seq.
```

The synchronisation operations *Signal* and *Wait* can be implemented in terms of event counts and sequencers. We define a type *Semaphore* that consists of an event count and a sequencer.

We define a header file "Sem.h":

```
typedef struct {
  ansa_EventCount Ec;
  ansa_Sequencer Sq;
} Semaphore;
Semaphore * Create();
void Signal();
void Wait();
```

The following defines the synchronisation primitives *Signal* and *Wait* in terms of event counts and sequencers:

```
#include "ansa.h"
#include "capsule.h"
#include "ecs.h"
#include "Sem.h"
Semaphore *Create(int initEc, int initSq)
{
    Semaphore *newSem;
    newSem = (Semaphore *) malloc( sizeof(Semaphore));
    newSem->Ec = ecs_makeEventCount(initEc);
    newSem->Sq = ecs_makeSequencer(initSq);
    return newSem;
}
void Wait(Semaphore * aSem)
{
    ecs_await(aSem->Ec,ecs_ticket(aSem->Sq));
}
void Signal(Semaphore * aSem)
{
    ecs_advance(aSem->Ec);
}
}
```

BB (Bounded Buffer) server.

We now discuss the use of the synchronisation primitives *Signal* and *Wait* in the procedures *Produce* and *Consume*. There are three types of synchronisation:

1. Mutual exclusion. This prevents two concurrent threads from interfering with one another's effects in accessing the buffer. The variable *Mutex* is used for this purpose. Both producer and consumer execute *Wait(Mutex)* before accessing the buffer and *Signal(Mutex)* after the access is complete. The initial value of the *Mutex* semaphore has both event counter and sequencer set to zero.
2. Producer blocking when the buffer is full: *EmptySem* is used for this purpose. It is initialised so that its event count is in advance of its sequencer by one less than the size of the buffer. Each time *Wait(EmptySem)* is called, the sequencer advances by 1. When the consumer takes an item from the buffer it calls *Signal(EmptySem)*.
3. Consumer blocking when the buffer is empty: *FullSem* is used for this purpose. It is initialised so that its event count is one less than its sequencer.

```

/* File : Server.dpl Server capsule for the BB service. */
!USE BB
!DECLARE { myRef } : BB SERVER
#include "ansa.h"
#include "tBB.h"
#include "machine.h"
#include "opsys.h"
#include "Sem.h"
#include "buffer.h"
extern char **environ;
#define CONCURRENCY 2
GLOBAL ansa_Cardinal Ansa_InitialTasks = (ansa_Cardinal)CONCURRENCY;
Semaphore *EmptySem, *FullSem, *Mutex;

void body(int argc, char *argv[])
{
    ansa_InterfaceRef myRef;
    char propbuf[1024];
    sprintf(propbuf, "Jean BB");

! {myRef}<-traderRef$Export("BB", "/ansa/testservices", propbuf, 8)
    EmptySem = Create(MAX, 1);
    FullSem = Create(0, 1);
    Mutex = Create(0, 0);
    initBuffer();
}

/* producer - calls Deposit to add a Thing to buffer
 * waits on EmptySem - (no of empty slots)
 * waits on Mutex and claims it - prevent 2 concurrent buffer operations
 * releases Mutex and signals a Full Slot has been added
 */
int BB_Produce( ansa_InterfaceAttr *_attr, Thing x)
{
    Wait(EmptySem);
    Wait(Mutex);
    Deposit(x);
    Signal(Mutex);
    Signal(FullSem);
    return SuccessfulInvocation;
}

/* producer - calls Fetch to get a Thing from buffer
 * waits on FullSem - (no of full slots)
 * waits on Mutex and claims it - prevent 2 concurrent buffer operations
 * releases Mutex and signals an Empty Slot has been added
 */
int BB_Consume( ansa_InterfaceAttr *_attr, Thing *_R1)
{
    Wait(FullSem);
    Wait(Mutex);
    *_R1 = Fetch();
    Signal(Mutex);
    Signal(EmptySem);
    return SuccessfulInvocation;
}

```

USING UNIX IPC TO IMPLEMENT A SIMPLE MASTER-WORKER MODEL

This project is based on the idea of using several workstations to carry out parts of a computation in parallel. See the Appendix to Project 3 for information about Unix datagram sockets.

Master-worker model

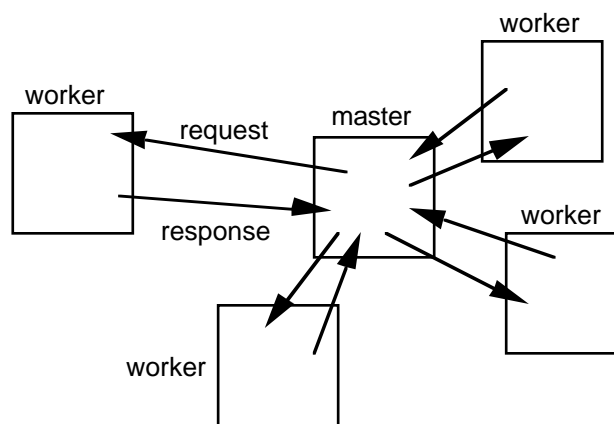
In a master-worker parallel program, a number of worker processes are available and a master process establishes a collection of identical worker processes. Each worker is capable of performing any one of the steps in a particular computation. In the simplest cases, there is only one step. A worker repeatedly gets a task from its master, carries it out and sends back the result.

The master keeps a record of the subtasks of a task (or computation) it is designed to perform. As each subtask is completed by one of the workers, the master records the result. Finally when all the subtasks have been completed, the master produces the complete result.

The program works in the same way irrespective of the number of workers available - the master just gives out a new task to any worker who has completed the previous one.

For example, suppose that we want to discover something about the usage of bulletin boards. Suppose that we want to know the proportion of the messages that are replies to previous messages and whether this varies from one bulletin board to another. The overall task consists of inspecting all the messages in each bulletin board and producing statistics for overall use of all bulletin boards and similar statistics for each bulletin board separately. The subtask for a worker consists of supplying the number of messages and number of replies for one single bulletin board. The master has to keep a list of bulletin boards; supply the name of a bulletin board to a worker when it is ready and record the result each time a worker completes a subtask. When all the subtasks have been completed, the master has to report the final result.

The messages exchanged between master and workers



Master-worker model

The master and workers communicate by sending and receiving UDP datagrams. Each datagram contains either a *request message* from the master to a worker or a *response message* from a worker to the master. The figure shows a master and 4 workers. The master issues a request message to a worker to do a subtask and then receives a response message when the worker has completed the subtask. The request message contains a description of the subtask - in our example the request message will contain the name of a bulletin board. The response message contains the result of the subtask - in our example, it will give the bulletin board name and the statistics collected. The response message should normally contain the specification of the subtask as well as the result so that the master does not need to keep track of which worker is doing a particular task.

Note that when a process uses *recvfrom* to receive a datagram it can determine the Internet address and port number of the sender. This allows both master and workers to check the origin of any message they receive.

Ports

You will want to run worker processes that can coexist with other people's processes in the same computer. You need to select an agreed port number for the workers to receive request messages from the master and another for the master to receive response messages. The same number can be used for both purposes. Two processes on the same computer cannot use the same local port number. You will therefore want to choose a port number that is sure to be different from other people's port numbers. If everybody takes the first unreserved port number and adds their *uid*, there should be no such clashes - i.e. :

```
aPort = IPPORT_RESERVED + getuid();
```

Task names

You may consider making it possible to have more than one type of parallel computation, say one program to do the bulletin board statistics and another to generate prime numbers. If you do, it may help to provide a task name that can be recognised by both master and worker. This will be useful if, for example, you have some workers doing the bulletin board tasks and subsequently start up a prime number task. If any of the workers from the first task are still about, they may pick up messages about the prime number task and get very confused. When the master establishes the group before doing the task, it will ask the worker to check whether it has the same task name.

Starting up the workers and establishing the group

To start a master-worker program first start several worker processes - each one running on a different workstation. Each worker is now available for work, but does not yet know about its master.

Now start up the master process. Its first task is to establish a group of workers.

A simple way is to supply as arguments to the master process the names of the computers where the workers are running. The master then takes the supplied list of computer names as the definition of its group of workers. Its first task is to get the Internet address of each worker and record it in the information about the group.

Worker process

The first thing the worker does is to create a UDP socket and bind it to the agreed local port. The worker process repeatedly waits for request messages from its master and each time one arrives it performs an action (generally a sub-task of the master's main task) and sends back a response message containing the results.

A worker process should be able to recognise and respond to the following different sorts of request messages:

start up - learns who is its master, checks task name - if correct, records its master's computer Internet address and responds with ok;

sub task - gets arguments of subtask in request message (e.g name of bb), does subtask and sends response message containing arguments and result;

close down - the worker closes its socket, responds ok to master and exits.

When the worker receives either the *sub task* request message or the *close down* request message, it should check whether it came from its master before obeying.

The worker process will also contain the procedures for carrying out a subtask. These *application procedures* should be kept as separate as possible from the rest of the worker program. The aim would be to make it easy to substitute a different set of application procedures.

Master process

The master process has a list of subtasks and their results (if any). This information is dependent on the application chosen. It should be separated as far as possible from the rest of the master program so that it would be easy to reconfigure the master for a different application.

The master also keeps a record of its list of workers as described above. It will eventually record for each worker whether it has yet responded to the most recent request message - you should work out the details as to what information you need here about start up and close down etc.

The master creates a UDP socket and binds it to the agreed local port. The lifetime of the master consists of *repeatedly*:

either sending a request message to a worker;

or receiving a response message from a worker;

or receive input from the terminal requesting it to print out monitor information

until all subtasks are completed, whereupon it instructs the workers to close down; reports its results and exits.

The master decides on the basis of the information in its list of workers which sort of request message to send out and which worker to send it to. For example, (when there is still outstanding work) and when it has heard that a particular worker is either ready to start or has just completed its previous task it will send out a message requesting it to do another subtask. The difficult question is: "how do you make the master co-ordinate all these things?". Hint: uses the system call *select*.

The master should be able to **send** the following different sorts of **request messages**:

start up - to check that a particular worker is running and has the same task name;

sub task - message specifies one of the outstanding subtasks;

close down - requests worker to close down.

The master should be able to **receive** and act on the following **response messages**:

start up - message specifies whether a worker is available, master notes the availability of the worker in its worker list;

sub task - message specifies one of the outstanding subtasks and its result, master records result and notes worker has responded;

close down - message confirms that worker has closed down; master notes the fact.

Monitoring

The user who starts the master should be allowed to monitor its progress. The master program should be able to recognise that the user has typed something. The monitor output will report information about the following:

the state of the workers (from the worker list);

the progress of the task - fraction of sub-tasks done;

if possible some more information from the list of results, e.g. printed by a procedure supplied in the application.

The application

You should choose one application that requires fairly intensive computation to carry it out and which is easily divided into a number of identical subtasks. You could consider tasks such as: (i) parallel compilation of the modules of a program (parallel make); (ii) searching for files containing a particular text string; (iii) finding prime numbers.

Giving in the work

Please supply us with print outs of your master and worker programs. On both of them, mark in colour the part that is specific to the application and supply us with a description of what we would have to do to reconfigure them for a different application. We will also require you to demonstrate the programs to us in the lab.

This is the first of a series of projects (Projects 7-10) designed to give you some experience in the design of clients and servers and the use of Remote Procedure Calling. In this project, you first implement a single (non-distributed) program and then divide it into two separate parts. One part will play the role of an Association Server. The other part will be a client program. The two parts will communicate by means of Remote Procedure Calling.

This project suggests the use of a hashing algorithm. However, the associations may be stored in any appropriate data structure, e.g. a linked list or an array of structs.

Sets of Associations

The program maintains sets of associations. An association is simply a mapping from a *key* to a *value*. Each association includes an *identifier* as well as the *key* and *value* - this means that several different sets of associations can be stored. Some examples of such sets are:

mappings of names to telephone numbers, e.g.

QMW → 081-980 4811

UCL → 071-387 7050

Imperial → 071-589 5111

mapping from French words to English words

lundi → monday

mardi → tuesday

mercredi → wednesday

Each set of mappings will be given an identifier, e.g. the first set could have $id = 1$ and the second set could have $id = 2$. The key part of an association is a string, but the value part should be capable of holding any information that can be expressed as a sequence of bits (e.g. pictures etc.).

Storing and Retrieving Associations

This program will be structured so that one module provides users with a means for looking up values from their keys, e.g. supplying a name as a key and receiving the corresponding telephone number. The same module will also provide a facility for adding/deleting/replacing associations within a particular set. We will refer to this module as the **command interpreter module**. (This module will eventually become the client program).

Another module of the program will be concerned with storing the associations in such a way that, given an identifier and a key, the corresponding value can be retrieved very quickly. In the discussion of hashing we refer to a single key, but this can be regarded as a combination of our identifier and key.

Hashing

The basic idea of hashing is to divide all the potential data items into a number, N of separate sets, so that when you need to search for a particular data item, you can go directly to the set to which it has been assigned. The number of data items in each set should be much smaller than the total number of data items, so the search for a particular item is relatively quick.

The technique of hashing requires a hash function which takes a key as argument and returns an index whose value is between 0 and N and refers to one of the sets of data items. Hash functions are designed so that the potential data items will be distributed evenly between the sets. With an even distribution of the data, the lookup time should be $1/N$ of the lookup time within a single sequence of data, assuming that the hash function is simple to calculate.

The hash function

The hashing algorithm is designed to map variable-length text strings onto small integers. It is described in [1]. The algorithm (in C notation) to get an 8-bit hash from a string *s* is:

```
h = 0;
while(*s) h = T[h ^ *s++];
return h;
```

where ^ is a bit-wise exclusive or operation and T is an array of N elements containing a permutation of the integers 0-(N-1). The table T can be constructed as follows:

```
unsigned char T[N];
makeT(nPerms, seed)
    int nPerms, seed;
{
    int i, a, b, tmp;

    for(i=0; i<N; i++) T[i] = i;
    srand((unsigned) seed); /* seed for random numbers */
    for(i=0; i<nPerms; i++) {
        a = rand() % N; /* rand returns 0 to 32767*/
        b = rand() % N; /* the % symbol is the mod operation */
        tmp = T[a];
        T[a] = T[b];
        T[b] = tmp;
    }
}
```

where *N* is 256. The argument *nPerms* specifies the number of permutations desired and should be greater than 256.

The Program Modules

In the next project, you will be asked to divide your program into two separate parts. The Associations module and the modules on which it depends (Hash, Hash Sets and Sets modules) will become the Association Server. The command interpreter module will be adapted so that it can be used by clients of the Association Server.

Associations Module

The purpose of this module is to call the functions that use hashing to store and retrieve associations. An association is defined by a tuple of values (id, key, value, size), whose types are defined in the following table:

<i>name</i>	<i>type name</i>	<i>description</i>
<i>id</i>	<i>ID</i>	a long integer (and may be bigger in subsequent versions)
<i>key</i>	<i>Key</i>	(pointer to) a null terminated string
<i>value</i>	<i>Value</i>	(pointer to) an array of bytes that may include null characters (it is not null terminated)
<i>size</i>	<i>Size</i>	the number of bytes in the <i>Value</i> component (necessary because the value is not null terminated)
<i>s</i>	<i>Status</i>	indicates the success or otherwise of the operation, e.g. <i>OK</i> , <i>Replacement</i> , <i>NotFound</i> , etc.

This module will provide the following functions, each of which returns a *Status* value:

<i>name</i>	<i>description</i>
<i>PutAssociation: ID X Key X Value X Size</i> → <i>Status</i>	takes as arguments the identifier of a set and a (key, value) pair to be added to that set. It causes the information to be stored for future retrieval. If <i>PutAssociation</i> is called a second time with the same key, the new value should replace the old one. The status value will distinguish between the two cases.
<i>GetAssociation: ID X Key</i> → <i>Value X Size X Status</i>	takes as arguments the identifier of a set and a key, it causes the stored set to be searched for the given key. If it is found, the value is returned. The <i>Status</i> value will distinguish between success and failure.
<i>DeleteAssociation: ID X Key</i> → <i>Status</i>	takes as arguments the identifier of a set and a key. It causes the association to be removed if it is present in the stored set. The <i>Status</i> value will indicate whether the association was found.
<i>Enumerate(ID)</i> → <i>set of Key X Status</i>	returns all the keys associated with a give ID.

Note that when the function *Enumerate* becomes a remote procedure, the keys can be concatenated into a single output argument. Therefore it may be a good idea to implement it like this now. You will need to consider how to recognise the end of each key and of the sequence of keys.

Command interpreter module

This module consists of a function whose body contains a loop. In each iteration the user is prompted to give a command, the user's command is read, executed and any result is printed. The commands include:

Specify Id of set (prompts user for id)

Put an association with the current id (calls *PutAssociation* after prompting user for key and value)

Get the value from an association with the current id (calls *GetAssociation* after prompting user for key, if a value is returned, prints it)

Delete an association with the current id (calls *DeleteAssociation* after prompting user for key)

Explain - print out a message explaining the meaning of the most recent status value.

Quit

At the end of each iteration, the user will be shown whether the status value indicated success. If not, the user can request an explanation.

Hash module

This module implements the hash function described earlier. It will include the table, *T* with 256 entries, together with the function *makeT* described above and the function *hash*:

<i>name</i>	<i>description</i>
<i>hash: char * → unsigned char</i>	takes a null terminated string of any length as argument and returns a positive integer in the range 0-255.

This module should include another hash function that is designed to take our *id, key* pair as argument. It combines them to produce a string and uses it as an argument to the *hash* function. (e.g. use *sprintf* to generate the string).

<i>name</i>	<i>description</i>
<i>ourHash: ID X char * → unsigned char</i>	takes an id and a null terminated string of any length as argument, calls <i>hash</i> and returns a positive integer in the range 0-255.

Hash Sets module

This module implements the *N* separate sets into which the associations are divided. We take *N=256* to correspond to the range of values supplied by the hash function. We use the type name *Tuple* to refer to the type of a tuple of values (id, key, value, size) whose types are defined as before.

The module provides the following functions, each of which should return an appropriate status value:

<i>name</i>	<i>description</i>
<i>InitialiseSets: → Status</i>	give each of the 256 sets an initial value representing "empty set".
<i>Insert: Tuple → Status</i>	use <i>id</i> and <i>key</i> from the <i>Tuple</i> , call <i>ourHash</i> to get an integer between 0-255, add the tuple to the set indexed by that integer.
<i>Find: Tuple → Tuple X Status</i>	use <i>id</i> and <i>key</i> from the <i>Tuple</i> , call <i>ourHash</i> to get an integer between 0-255, look for the tuple in the set indexed by that integer. If found, return it.
<i>Delete: Tuple → Status</i>	use <i>id</i> and <i>key</i> from the <i>Tuple</i> , call <i>ourHash</i> to get an integer between 0-255, look for the tuple in the set indexed by that integer. If found, remove it.

The N sets can be represented by an array whose slots are indexed from 0-255. Each slot will represent a set of tuples.

The empty hash table is initialised as an array of 256 empty lists. Each time a new tuple is inserted, it is added to the front of the list in the appropriate slot.

This module assumes the existence of a module that implements operations for sets of tuples.

Sets module

This module provides the operations for sets of tuples. Each function will return a status value indicating its success or otherwise. These functions will include

<i>name</i>	<i>description</i>
<i>EmptySet: → Set X Status</i>	return a value representing an empty set
<i>AddMember: Tuple X Set → Set X Status</i>	add the given tuple to the given set and return a value representing the new set
<i>Find: Tuple X Set → Tuple X Status</i>	look for a tuple matching the tuple argument in the given set, if one is found, return it.
<i>RemoveMember: Tuple X Set → Status</i>	remove the given tuple from the given set and return a value representing the new set

Testing the non-distributed program.

The program requires a main module that calls the functions that initialise the table T used by the hash function, and the N empty sets and then call the command interpreter function. The program should be tested thoroughly before you attempt to divide it into client and server parts. You should keep the non-distributed version of the program for future use in the development of later projects. The service will require a service interface for the Association service. You should make use of Sun RPC, which is described in Section 5.4. For further information on Sun RPC see [Sun 1990].

The service interface

The server will provide the four functions in the Associations module:

PutAssociation: ID X Key X Value X Size → Status
GetAssociation: ID X Key → Value X Size X Status
DeleteAssociation: ID X Key → Status
Enumerate(ID) → set of Key

For each function in the service interface, it is necessary to decide which are the *input* and which are the *output* parameters (see Section 5.1). For example, the unique identifier and the key are input parameters in all the functions.

The service interface should be defined in Sun XDR as shown in Figure 5.4 and put in a file with a suitable name (e.g *Association.x*). This requires that the types of each parameter and result are specified using *typedefs*. For

example, the type *ID* of the unique identifier could be a long (but should not be less than 32 bits - it will be made into a capability in a later project). The *Key* can be of type *string* which corresponds to the type *char ** in the C client and server programs. The *Value* can be passed as an array of characters whose size is specified in *Size* when the procedure is called (see the *Data* example in Figure 5.4).

All of these functions return a *Status* value that indicates whether the procedure was executed successfully. In the case of a successful execution, the value *OK* is returned. In the case of an unsuccessful execution, an error value is returned, for example, *GetAssociation* would not succeed if the (*id*, *key*) pair did not exist and the return value could be *NotFound*. The set of possible values for *Status* can be defined as an *enum* in C notation in the interface specification. In addition a *struct* has to be defined for the input and output arguments of each of the service functions. Consider *GetAssociation* whose input arguments are the *ID* and *Key* and output arguments are the *Value* and *Size*. These require a constant and pair of structs e.g.:

```
const MAX = 7500; /* maximum size of value (see note on client program) */
typedef int ID;
typedef string Key<>; /* XDR provides type string which is compiled to char ** */
enum Status { OK = 0, NotFound = 1, Replaced = 2};
struct getargs {
    ID id;
    Key key;
};
struct Value {
    int size;
    char value[MAX];
    Status status;
};
```

Finally, the service interface must specify the signatures of all the procedures inside a *program* and *version* declaration (see Figure 5.4). For example, you could specify the signature of *GetAssociation* as:

```
Value GETASSOCIATION(getargs) = 2;
```

Server Program

The program for the association server requires a Service Interface module containing the four remote procedures, the Association module and the modules on which it depends. Figure 5.6 shows the header files required. The *typedefs* generated from the XDR interface are included from a file with the same name (but *.h* extension) e.g. *Association.h*. The name of a service function must be the lower case equivalent of the name in the interface definition followed by an underscore and version number. In addition, each procedure has exactly one argument. The simplest method is to define some new service functions that call the functions in the Association module. For example, the service function for *GetAssociation* might be:

```
Value * getassociation_1(getargs *a) /* version 1 */
{
    static Value value;
    Status status;

    /* use the components of the argument a and call the existing function e.g.
       value.status = GetAssociation(a->u, a->key, &value.value, &value.size)
    */
    return &value;
}
```

Note that the composite value to be returned must be held in a static local variable and that the parts returned (*Value*, *Size* and *Status*) must be assigned to this variable before they are returned. In addition, it is the address of the variable that is returned.

Client program

The client program is based on the command interpreter module from the first project. But each call to a remote procedure must be denoted as shown in Figure 5.5. For example, the call to the service procedure corresponding to *GetAssociation* might be:

```
gettargs a;
Value * value;
...
value = getassociation_1(&a, clientHandle);/* where the version number is 1 */
```

where *clientHandle* is obtained as the result of *clnt_create*.

Note that if there is an error due to failure of an actual RPC, the result will be NULL. Therefore the client program should test each result. For example, after the above call to *getassociation_1* test it as follows:

```
if (value == NULL) {
    clnt_perror(clientHandle, serverName);    /* prints an error message */
    exit(1);
}
```

If the result is not NULL, the *Status* value returned by the server should be tested. If it is anything other than OK, the user should be told there is an error. The user may then request a call to the *Explain* procedure to get an explanation as before.

Note that if you use Sun RPC with UDP, the value in the call to *PutAssociation* should not be so big that the RPC message exceeds a maximum of 8 kilobytes. Note the declaration of the constant MAX in the interface definition. This should be checked in the client program before the RPC call.

Note also that the *Enumerate* function in the server may possibly need to return more keys than can fit into the return argument of a single RPC call. If you have time, you should try to address the problem (See Section 12.2 and Exercise 12.5).

Compiling and testing the program

Use the interface compiler *rpcgen* to compile the interface definition in *Association.x*. This produces the header file *Association.h* and client and server skeletons in *Association_clnt.c* and *Association_svc.c* and marshalling procedures in *Association_xdr.c*. These may be compiled with the other components of the client and server programs. Run the server on one computer and test it with several clients running on different computers and make sure that it behaves correctly in the presence of multiple clients.

Reference

[1] *Fast Hashing of Variable-Length Text Strings*, by Peter Pearson, Communications of the ACM, Vol. 33, No. 6, pp 677-680 (June 1990).

STORING THE SETS OF ASSOCIATIONS IN UNIX FILES

This is the second project on the Association service. It consists of enhancing the server so that it provides *recoverable* associations (see Section 12.3). That is, associations that are automatically saved in disk storage and can be recovered after the server crashes. This enhancement should be tested first with the non-distributed version of the program.

In the first project it was assumed that all the associations handled by your server reside in volatile memory. Therefore you lose them all each time the program exits. The general idea of this project is that you should provide functions for storing associations in files on disk and for reading them back and incorporating them in the volatile memory representation.

Sets of permanent Associations

For convenience in describing our suggested scheme, we use the term “permanent Association” for an association that has been saved in a file and can be read from the file and installed in the set of Associations in volatile memory. We now describe a simple scheme in which all permanent Associations are saved in the same file during a particular run of the program. The last part of the project extends the scheme to provide a recoverable service.

When the program is restarted, all the Associations in that file are read in and installed in volatile memory. Each Association is read from the file and treated in the same way as an Association supplied by the user via *PutAssociation* - that is, it is made into tuple of values (*id, key, value, size*) and the *Insert* function is used to add the tuple to the set indexed by the integer its (*id, key*) pair hashes onto.

The operations *GetAssociation*, *PutAssociation*, *DeleteAssociation* work exactly as before - they apply to the sets of Associations stored in volatile memory. The current task involves the design of two further functions

<i>name</i>	<i>description</i>
<i>RetrieveAssociation: FileId</i> → <i>Status</i>	reads the next Association from the file referenced by <i>FileId</i> , builds it into a tuple (<i>id, key, value, size</i>) and adds it to the appropriate set of associations. The <i>Status</i> value will distinguish between success and failure - the latter should indicate the sort of failure (e.g. <i>EOF, badFileId</i>).
<i>StoreAssociation: FileId X Tuple</i> → <i>Status</i>	takes as arguments a <i>FileId</i> and a <i>Tuple</i> containing (<i>id, key, value, size</i>) to be appended to that file. It causes the information to be appended to the file for future retrieval. If <i>StoreAssociation</i> is called a second time with a tuple containing the same <i>id</i> and <i>key</i> , the new value will be later in the file.

The type names are defined as:

<i>Status</i>	as in the first project indicates the success or otherwise of the operation, e.g. OK, Replacement, NotFound, etc.
<i>FileId</i>	The <i>FileId</i> will be a file descriptor if you use the <i>Read</i> and <i>Write</i> system calls. It will be a <i>FILE *</i> structure if you use the library functions <i>FRead</i> and <i>Fwrite</i>
<i>Tuple</i>	refers to the type of a tuple of values (<i>id, key, value, size</i>) whose types are defined as in the first project.

Note that if the entire file of permanent Associations is read in, then replacements come later in the file and are inserted later in the sets - producing the effect of replacement.

Using the permanent Associations

You can add two commands in your command interpreter e.g.:

Save all the associations in a file (the file is made empty and then all the Associations currently stored in volatile memory are enumerated and *StoreAssociation* is called for each. Finally the file is closed)

Restore all the associations from a file (opens the file at the beginning and then repeatedly calls *RestoreAssociation* until the end of file is reached)

There is an asymmetry here. The Restore function repeatedly calls *RestoreAssociation* to read associations from the file and calls *Insert* and is quite similar to repeated calls to *PutAssociation*. Note that if Save always uses the same file, you risk losing the permanent associations written by the previous Save operation. You should arrange to retain a copy of the previous file until the current Save is successful.

The Save function depends on the Hash Sets module to enumerate each of the N separate sets into which the associations are divide.

<i>name</i>	<i>description</i>
<i>EnumerateSets: Something</i> → <i>Status</i>	take each of the 256 sets and calls enumerate in the Sets module to do something with each of its members - in this case <i>Something</i> is the function <i>StoreAssociation</i> that is passed on to each member of the set.

Thus the Save function also relies indirectly on the Sets module to provide an operation to enumerate the members of a set and do something else with it - in this case, flatten it into a buffer and then write it to file.

<i>name</i>	<i>description</i>
<i>Enumerate: Set X SomethingElse</i> <i>X Status</i>	take each member of the set argument and does something else with it. <i>SomethingElse</i> is a function that can be applied to a tuple - in this case, <i>StoreAssociation</i> .

Flattening and unflattening tuples

The contents of any Unix file is a sequence of bytes. Therefore any information you write to a file must be converted to a sequence of bytes - when the original information is structured with pointers, this conversion process is sometimes called flattening. In this example we have a tuple that contains (id, key, value and size). The key and the value are pointers to other information stored on the heap. You have to design a protocol for storing an association as a sequence of bytes. The main point of the protocol is that you must be able to reconstruct the Association from the sequence.

We suggest that when you have designed your protocol, you define a pair of functions:

<i>name</i>	<i>description</i>
<i>Status flatten: TupleX Max</i> → <i>Buffer X Size</i>	given a tuple, flattens it out into an array of bytes in a buffer of maximum size; returns total size via last argument.
<i>Status unFlatten: BufferX Max</i> → <i>Tuple X Size</i>	given an array containing a flattened tuple, construct a tuple; returns total size via last argument

The type names are defined as follows:

<i>Buffer</i>	(pointer to) an array of bytes that may include null characters (it is not null terminated)
<i>Max</i>	an integer specifying the maximum number of bytes that may be put in Buffer
<i>Size</i>	returns the number of bytes in the Buffer component (necessary because the value is not null terminated)

Recoverable Associations

The plan is to make the program into a recoverable association service with multiple clients. Recovery should be transparent to the clients, which means that the server should decide when to save its associations and will not take instructions from clients about recovery.

To make the associations truly recoverable in the case that the server crashes, each new association must be saved in the file at the time it is created (by *PutAssociation*), instead of just calling *Save* from time to time.

In this case, you will have to allow for the effects of *DeleteAssociation*. What additional information must be written to the file with each tuple? How will this alter the recovery procedure?

What measures can you take to prevent the file becoming too large?

Modify the program to provide truly recoverable associations. Implement a *Recovery* procedure that can be called each time the server is restarted, Note that the *Save* operation can be used to save all the current associations immediately after recovery.

Recoverable Association Service

The interface definition and client program from the previous project can be used without modification. The server program can use the same Service Interface module as the previous project but with one addition to the *main* of the server program to call the *Recovery* procedure when the server is restarted.

The server program must of course make use of the new Association module which includes the new recoverable version of *PutAssociation* and the definitions of *RetrieveAssociation* and *StoreAssociation*. It also requires the flattening and unflattening procedures, together with the corresponding enhanced versions of the Hash Sets and Sets modules.

PROTECTING SETS OF ASSOCIATIONS WITH CAPABILITIES

This is another project on the Association service. It may be done after Project 7. It requires you to treat the association identifiers as **capabilities**. Before you do it you should answer some questions about the design.

Planning the service interface

The service interface will offer the procedures in the Association module, but with capabilities instead of IDs:

PutAssociation: Capa X Key X Value X Size → Status

GetAssociation: Capa X Key → Value X Size X Status

DeleteAssociation: Capa X Key → Status

Enumerate: Capa → set of Key X Status

Together with two new service functions to deal with capabilities.

<i>name</i>	<i>description</i>
<i>NewCapa</i>	server generates a new capability.
<i>RestrictCapa</i>	server restricts capability.

In the earlier projects, users just made up IDs for their associations. From now on, we will refer to IDs as capabilities. All capabilities are issued by the Association server and the latter stores only those associations that contain capabilities that it has issued previously.

Questions about Capabilities in the Association Service

The project requires you to design the IDs in your "Association Service" as **capabilities**. See Section 7.5.

1. Describe the components of a capability for your association service - it should include fields for the server number as well as access rights for each of the 4 operations provided in this service. Describe the purpose of each component and how many bits you plan to use for it.
2. Describe which parts of a capability are revealed to clients. Will clients be able to tell what rights they have?
3. Describe the components of a capability as stored in the server. Where will capabilities be stored?
4. Explain how the clients will be prevented from forging their access rights.
5. Define the procedure *NewCapa* in the Association service that can be called whenever a client wants a new capability. Say what its arguments and results will be and (briefly) what its action will be - e.g. what access rights will be in the capability generated and how the server will remember it has issued this capability.
6. Define another procedure *RestrictCapa* that enables a client that possesses a capability to ask the server to return a restricted form for giving to another client. Can a restricted capability be further restricted? In your design must this procedure be part of the server or can it be carried out by the client without loss of security?
7. When the client calls the functions *PutAssociation*, *GetAssociation*, *DeleteAssociation* and *Enumerate*, the call will include a capability. How will the server recognise the access rights in the capability?
8. Describe the changes to the functions *PutAssociation*, *GetAssociation*, *DeleteAssociation* and *Enumerate* that are necessary to make what were previously unique identifiers play the role of capabilities.
9. Give a complete list of the error values required in the program when capabilities are included

Encryption

Your description will probably have mentioned the encryption of components of capabilities. You can use a very simple algorithm, which could be replaced by a more sophisticated one in a serious implementation.

Program testing

We will want to run the program as before, but we will not be entering the values of the IDs or capabilities. We will expect the command interpreter to give us additional commands for the following:

- getting a new capability;

- producing a restricted capability;

We will want you to convince us that when we ask for a new capability it gives us full access rights. We also want to see that when we ask your program to produce a restricted capability, the latter will do only those operations for which it is designed. We will also want to see that the server refuses to use a forged capability.

PROJECT 10 A DIRECTORY SERVICE

This project may be done after Project 7 on the Basic Association service. It requires you to build a “Directory Service” as a client of your “Association Service” and to extend the client program to use both. You should keep the original versions of the programs for testing and do this project as a new version.

Named Sets of Associations

The Association service can be used to map the textual names of sets of associations onto their IDs. For example, you might store the mappings:

“Phone Numbers” -> ID1

“French Words” -> ID2

then all the phone numbers will be stored with ID1 as their identifier and the french words with ID2. We will refer to the text name to ID mappings as *directories*. The first part of the project is to implement a set of directory operations as another program that is a client of the Association service. This program will be made into a *directory server* and will have clients of its own.

The directory service interface

Each of the directory operations will take the ID of a directory as its first argument. The directory operations are:

LookUp(*DirID*, *name*, *ID*) - for looking up a text name in a directory and returning the ID.

AddName(*DirID*, *name*, *ID*) - for adding a text name -> ID mapping to a directory.

UnName(*DirID*, *name*) - for removing a name from a directory

ReName(*DirID*, *oldName*, *newName*) - alter the name in a mapping in a directory

GetNames(*DirID*, *Names*) - returns the set of names in a directory

Each of these functions should return a *Status* value that indicates whether the procedure was executed successfully. The directory service should also provide an *Explain* function similar to the one in the Association service. In the RPC interface definition, *DirID* and *name* arguments should have the same types as *ID* and *Key* in the Association service. See page 208-210 for further ideas about a directory service.

Implementing the directory server

The operations *AddName*, *LookUP*, *UnName* and *GetNames* in the directory service will be implemented by making RPC calls to *PutAssociation*, *GetAssociation*, *DeleteAssociation* and *Enumerate*. In *AddName* and *LookUP* you will have to pass the ID of the set of associations as the *value* in a mapping. You can also include information about the ID (e.g. a type indicating whether it is referring to another directory or to a set of mappings). It may be convenient to convert the ID and its type to a string of characters in *AddName* before calling *PutAssociation* and converting it back in *LookUP*.

The *ReName* function can be implemented in terms of *GetAssociation*, *DeleteAssociation* and *PutAssociation*. Do you foresee any potential problems here?

The client program for testing the directory server (client v2)

The client program should be designed with an interactive interface to enable users to call each of the procedures supplied by the directory server repeatedly. This time you can start with a fixed ID to represent the directory. The *GetNames* procedure will require a client side procedure for printing out the names.

The client program for using both servers (client v3)

The final client program will hide the details of the IDs from its users by providing a “User Package” (see page 130) with an interface that first selects a directory by name and then performs the operations on that directory until another one is selected. For example, supposing the selected directory already has two entries in it:

```
ListNames() -> Phone Numbers, French Words
```

```
ChangeDirectory(“Phone Numbers”); Put(“QMC”,”01-980-4811”); Get(“QMC”) -> “01-980-4811”
```

```
ListNames() -> QMC ...
```

```
GoToParentDirectory();
```

```
ListNames() -> Phone Numbers, French Words
```

```
ChangeDirectory(“French Words”); Put(“Pain”,”Bread”); Get(“Pain”) -> “Bread”
```

Initially a default directory *ID* is set and all directory operations refer to that directory. For example, *ListNames* will call the *GetNames* with that *ID* as argument. The *ChangeDirectory* operation will use the *LookUp* operation to find a *ID*; after that, the current directory changes to the one selected. The *Put* and *Get* operations will call *PutAssociation* and *GetAssociation* with this *ID*. You should provide corresponding functions for deleting and enumerating associations. Note that the program must be able to go back to the starting directory. If you plan to implement a hierarchy of directories, then your *GoToParentDirectory* will need to be more sophisticated.

This client program (v3) should also be designed with an interactive interface to enable users to call each of the procedures *ListNames*, *Select*, *Put*, *Get* etc.

Consider the design of the function that removes names from a directory - what should it do if there are associations in the directory to be removed?