

Final project report Ticket purchasing system

Objectives

The objective of this project is to simulate a ticket purchase system. The system would have one main server and several clients connected to it. The concept of this project is to simulate a ticket selling system. It would have a main server to keep tracking the number of tickets available for sale, and several registers to sell the tickets. The main server has a database server running to keep the number of tickets. The server also receives purchasing order from the registers and authorizes the transactions if the tickets are available for sale. The registers are the place that accepts purchasing orders from customer. In the real situation, the system would have many registers set up in many locations, and all registers could request for tickets at the same time. To make the communication between the clients and the server, they would be all connect to the same network so that they can send and receive data with each other. In this project, we also try to imitate this system by having one register as a selling station and connect those registers together to the main server through the same network. The main problem in this project is that the server has to be able to handle with multiple purchasing orders from several registers at the same time. It has to control the number of selling tickets and give authorize to the registers to sell tickets if they are available. At the end of the day, the tickets sold at the registers and the server have to match so the server has to make sure that it authorizes the order only if the tickets are available and reply the authority to sell to the right register.

At the end of this project, we expect to see the result that the server have a database run, and all registers can send and receive messages successfully. Also, the server has to be able to handle with multiple orders from multiple registers and make sure that both the clients and the server can authorize tickets to sell correctly.

Implementation

To implement this system, we use one TS-7250 board as the main server and several TS-7250 boards as a client. The main server acts as the main machine to control the whole system. All clients have to connect with the server. Then the main server has to receive purchasing orders and reply back with a permission to sell tickets to the clients. It also run a database server to keep tracking the number of tickets that are available to sell. We can divide the implementation details to 2 main parts, which are server part and client part.

For the implementation of server part, we implement the server to run on a TS-7250 board. The system has to have one server only, and the server has Faircom database server to keep the record of the tickets. To implement the system with Faircom database, we follow the same way as in the tutorial from Faircom. In other words, we edit the tutorial that Faircom gave to us and add other functions into the file on top of it. First we put our ticket table into the database and add socket into the program to make this be able to communicate with other clients connected in the same network. The server receives an order from a client and then processes the order to check if it can sell the tickets. Finally,

it replies back to the client to let it know whether or not it can sell the request to a customer. To setup the Faircom database server, first we have to run the database server on the main server board. Then, in our main program, we initialize and control the database from here. First the main program connects to the database server, and creates a table to store our ticket data. Then, we add our ticket data to the table and use this table to track the number of tickets left. When our main server receives an order from a client, it comes look at the table to see if it can sell that the tickets as requested or not, and then it update the number in the table if it can sell and send a result back to the client. The flowchart of our main server is shown in figure 1.

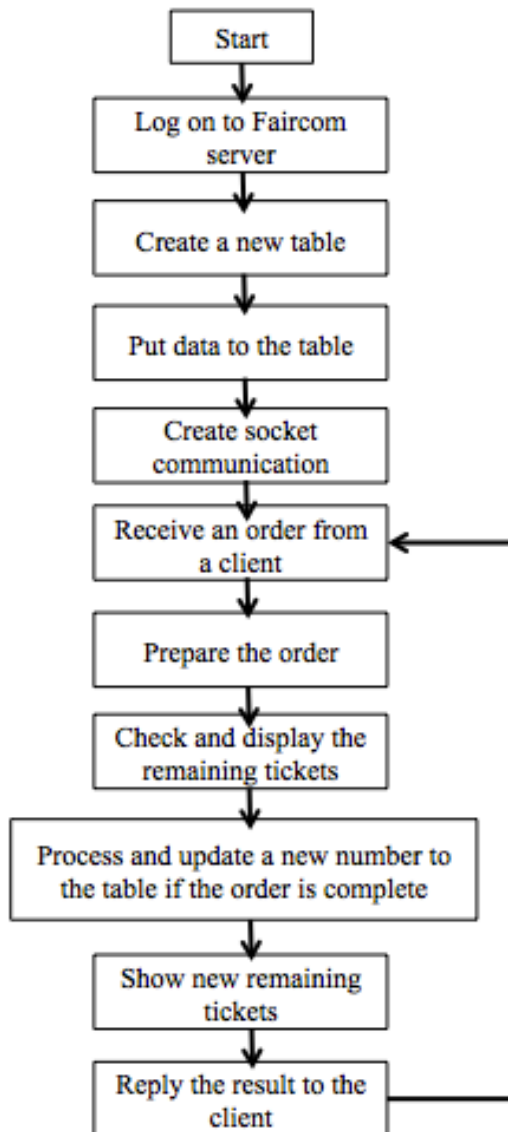


Figure 1 Flowchart of main server

The clients are created on the concept that the system can have selling register in many locations, and in one location it can have more than 1 register. To implement under this concept, we use one TS-7250 board to represent one location, and in that board we create 3 threads in the main program. These threads are used to represent all registers that are at the same location. Thus, each thread would receive orders from customer and send the orders to the main server. In order to feed multiple orders to the main server, we could not manually type the orders on the keyboard because this would not give fast enough to see the behavior on the system if it can handle multiple orders at the same time. To fix this issue, we would install a kernel module in the board to feed orders to the threads. In this implementation of our client program, it can be mainly divided into 2 parts, which are user space program and kernel program.

In the user space program, first we basically create socket communication to the board so it could talk with the main server. Then the program creates 3 threads to represent 3 registers. In each thread, they all run with the same function. First, they get an order from the kernel module by reading the data from FIFO sent from the kernel. Then it sends the order to the main server and waits for the reply. To send the data to the main server, each thread has to grab a semaphore before it sends its order to the main server and release the semaphore to the others when it receives the reply. One of the reasons that we have to use this semaphore to let only one threads have access to the socket at a time is if all threads get access to the socket and try to read and write data from the server, these threads could not know that when they should read and write the data so that we observe the wrong sequence of display. The flowchart of client user space program is shown in figure 2.

For the kernel module for the clients, when we setup the module, first we setup the switch on the auxiliary board so that we can control the module to start and stop sending orders to the user space program via FIFO. Then, it creates 3 real time tasks to feed the orders. All tasks have the same function. First, they check if the B0 switch is pressed, if so they would send a data to the FIFO; otherwise, they would not send anything. At the end they sample a new period for its real time periodic task. The reason that we do this is because we want to have each thread run at a difference time in each loop. The flowchart of client kernel module is shown in figure 3. The overall diagram of the system is shown in figure 4.

Experiments and results

In this experiment, we run the system with 4 boards. One board is for the main server and the others for the clients. We could have used more boards to run in the system as a client but due to the limitation of the number of the board in the laboratory that we have to share with other students, we decide to run this experiment with only 4 boards as described above. For the following experiment, when we start the main server, initially the database is setup to be able to sell 2 types of tickets and each type has 500 tickets as shown in figure 5 so when we first start the program the server would show that it has 2 types of tickets and each of them has 500 tickets left as shown in figure 6. For the client boards, once we start a client by installing kernel module and start user space program, the program is designed to start sending orders to the main server after switch B0 is pressed. When the client sends the order to the server, it made a random number between

1 and 2 to represent type of ticket along with another random number from 1 to 3 to represent number of ticket requested in the order.

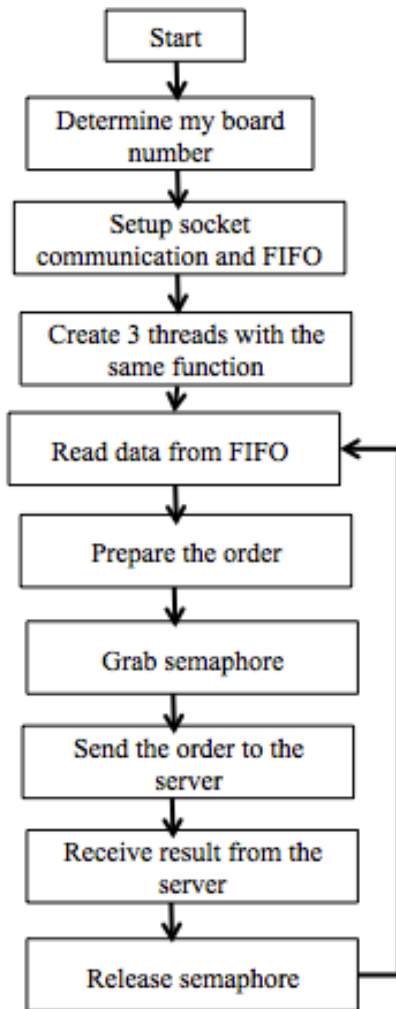


Figure 2 Flowchart of client user space

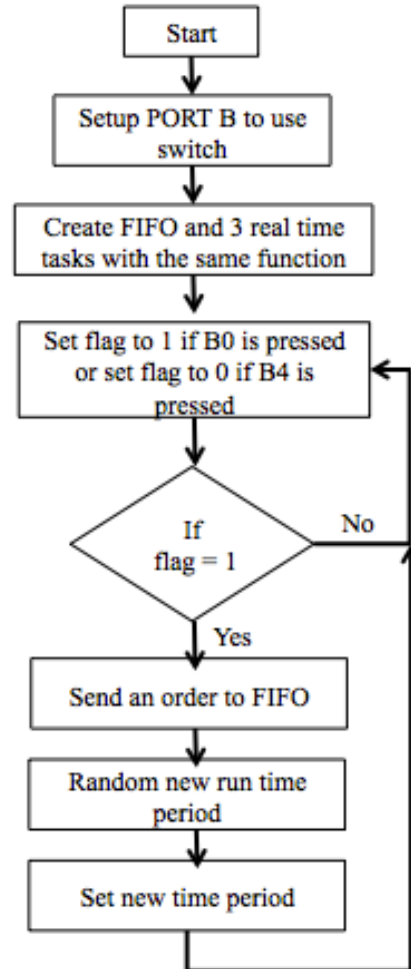


Figure 3 Flowchart of client kernel module

Once the order is sent, the client would wait to receive the result back from the server whether the order is approved or the tickets are sold out. From an experiment, we captured a screenshot of text files showing the output of the programs. As in figure 6, it shows the beginning of screenshot of the main server. In figure 7, it shows the sample output of a client program. We can see that it displays the order sent to the server and message received back from the server. When the order is approved, the server would reply back with a message approved and the transaction ID of the order. If the ticket is not available, the server would reply back with the message ticket sold out. Figure 8 and figure 9 are also an example screenshot from 2 clients but from difference boards.

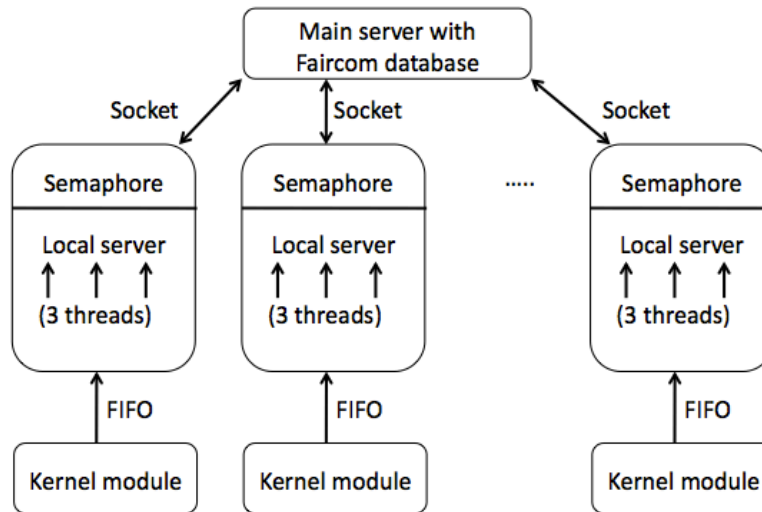


Figure 4 the overall diagram of the system

Type of ticket	Quantity
1	500
2	500

Figure 5 The ticket table in the database

```

INIT
    Logon to server...
DEFINE
    create Ticket table
MANAGE
    Delete records...
    Add records...
Ticket 1 remaining: 500
Ticket 2 remaining: 500
Received: # 18 id 1 ticket 2 qty 2 trns 1
Ticket 1 remaining: 500
Ticket 2 remaining: 500
Trying to sell 2 order(s) of ticket 2.....
Ticket 1 remaining: 500
Ticket 2 remaining: 498
Received: # 18 id 1 ticket 2 qty 2 trns 2
Ticket 1 remaining: 500
Ticket 2 remaining: 498
Trying to sell 2 order(s) of ticket 2.....
Ticket 1 remaining: 500
Ticket 2 remaining: 496
Received: # 18 id 2 ticket 2 qty 2 trns 3
Ticket 1 remaining: 500
Ticket 2 remaining: 496
Trying to sell 2 order(s) of ticket 2.....
Ticket 1 remaining: 500
Ticket 2 remaining: 494
Received: # 18 id 3 ticket 1 qty 1 trns 4
Ticket 1 remaining: 500
  
```

Figure 6 An example screenshot from the main server at the beginning

Sent: # 17 id 3 ticket 2 qty 1 trns 99
Received: Approved trns:99

Sent: # 17 id 1 ticket 1 qty 1 trns 100
Received: Ticket 1 SOLD OUT!

Sent: # 17 id 3 ticket 1 qty 2 trns 101
Received: Ticket 1 SOLD OUT!

Sent: # 17 id 3 ticket 2 qty 2 trns 102
Received: Approved trns:102

Sent: # 17 id 2 ticket 2 qty 3 trns 103
Received: Approved trns:103

Sent: # 17 id 1 ticket 1 qty 2 trns 104
Received: Ticket 1 SOLD OUT!

Sent: # 17 id 1 ticket 1 qty 2 trns 105
Received: Ticket 1 SOLD OUT!

Sent: # 17 id 1 ticket 1 qty 1 trns 106
Received: Ticket 1 SOLD OUT!

Sent: # 17 id 3 ticket 2 qty 1 trns 107
Received: Approved trns:107

Figure 7 An example screenshot from client1

Sent: # 18 id 3 ticket 1 qty 3 trns 242
Received: Ticket 1 SOLD OUT!

Sent: # 18 id 2 ticket 2 qty 1 trns 243
Received: Approved trns:243

Sent: # 18 id 1 ticket 2 qty 2 trns 244
Received: Approved trns:244

Sent: # 18 id 2 ticket 1 qty 3 trns 245
Received: Ticket 1 SOLD OUT!

Sent: # 18 id 1 ticket 2 qty 1 trns 246
Received: Approved trns:246

Sent: # 18 id 2 ticket 2 qty 2 trns 247
Received: Approved trns:247

Sent: # 18 id 2 ticket 1 qty 3 trns 248
Received: Ticket 1 SOLD OUT!

Sent: # 18 id 2 ticket 2 qty 1 trns 249
Received: Approved trns:249

Figure 8 An example screenshot from client2

```

Sent: # 15 id 1 ticket 2 qty 1 trns 146
Received: Approved trns:146

Sent: # 15 id 1 ticket 2 qty 1 trns 147
Received: Approved trns:147

Sent: # 15 id 3 ticket 1 qty 2 trns 148
Received: Ticket 1 SOLD OUT!

Sent: # 15 id 3 ticket 1 qty 3 trns 149
Received: Ticket 1 SOLD OUT!

Sent: # 15 id 2 ticket 1 qty 1 trns 150
Received: Ticket 1 SOLD OUT!

Sent: # 15 id 1 ticket 1 qty 3 trns 151
Received: Ticket 1 SOLD OUT!

Sent: # 15 id 1 ticket 1 qty 1 trns 152
Received: Ticket 1 SOLD OUT!

Sent: # 15 id 3 ticket 1 qty 2 trns 153
Received: Ticket 1 SOLD OUT!

Sent: # 15 id 1 ticket 2 qty 1 trns 154
Received: Approved trns:154

```

Figure 9 An example screenshot from client3

```

Received: # 15 id 1 ticket 2 qty 1 trns 147
Ticket 1 remaining: 0
Ticket 2 remaining: 22
Trying to sell 1 order(s) of ticket 2.....
Ticket 1 remaining: 0
Ticket 2 remaining: 21
Received: # 18 id 2 ticket 1 qty 3 trns 245
Ticket 1 remaining: 0
Ticket 2 remaining: 21
Trying to sell 3 order(s) of ticket 1.....
Ticket 1 remaining: 0
Ticket 2 remaining: 21
Received: # 17 id 3 ticket 2 qty 1 trns 99
Ticket 1 remaining: 0
Ticket 2 remaining: 21
Trying to sell 1 order(s) of ticket 2.....
Ticket 1 remaining: 0
Ticket 2 remaining: 20
Received: # 15 id 3 ticket 1 qty 2 trns 148
Ticket 1 remaining: 0
Ticket 2 remaining: 20
Trying to sell 2 order(s) of ticket 1.....
Ticket 1 remaining: 0
Ticket 2 remaining: 20
Received: # 18 id 1 ticket 2 qty 1 trns 246
Ticket 1 remaining: 0
Ticket 2 remaining: 20
Trying to sell 1 order(s) of ticket 2.....
Ticket 1 remaining: 0
Ticket 2 remaining: 19

```

Figure 10 Another example screenshot from the server

To compare the result from clients and the server, we include another screenshot of the main server to see order received from board number 15 in figure 10. In the figure, we can see that at transaction 147 of client #15 it asks for a ticket 2 so the server have enough ticket. The main server updates a new number in the database after processing and reply back to the client that the order is approved. On the other hands, transaction 148 of client #15 asks for 2 tickets of ticket 2. The server doesn't have enough ticket so it won't process the order and reply back to the client that the ticket is sold out.

In order to make sure that the number of ticket sold at the clients and the main server are match, we manually count the tickets sold at all clients and compare with the number of ticket available initially when we start the program. In this case ticket 1 and 2 are initially have 500 tickets so when we count the tickets sold at all clients they should sum up to 1000 tickets. For this experiment, we perform the test 5 times because we have to manually count the total tickets by hand with some help from Microsoft Excel but for each test sample it takes such a long time to calculate. Previously, we performed 15 sample tests but we found some errors in the main server during order processing so we had to go back to fix the problem and rerun the test again. We decided to do the retest only 5 times because of time constrain and from the originally test even the first 5 test sample they were all wrong so if we can have the no problem system with the first 5 rerun this should show that this system can work.

Form the experiment, we can see that the sums up of the total number of ticket sold at all clients are equal to the total number of ticket that the server initially has. The summary of the tickets is shown in figure 11-1 to 11-5

Ticket type	Initial number at the server	Tickets sold at			Total at clients
		Client1	Client2	Client3	
1	500	87	262	151	500
2	500	111	240	149	500
Total	1000	198	502	300	1000

Figure 11-1 Summary form experiment1

Ticket type	Initial number at the server	Tickets sold at			Total at clients
		Client1	Client2	Client3	
1	500	265	150	85	500
2	500	244	146	110	500
Total	1000	509	296	195	1000

Figure 11-2 Summary from experiment2

Ticket type	Initial number at the server	Tickets sold at			Total at clients
		Client1	Client2	Client3	
1	500	211	156	133	500
2	500	192	161	147	500
Total	1000	403	317	280	1000

Figure 11-3 Summary from experiment3

Ticket type	Initial number at the server	Tickets sold at			Total at clients
		Client1	Client2	Client3	
1	500	199	157	144	500
2	500	186	164	150	500
Total	1000	385	321	294	1000

Figure 11-4 Summary from experiment4

Ticket type	Initial number at the server	Tickets sold at			Total at clients
		Client1	Client2	Client3	
1	500	157	214	129	500
2	500	89	146	265	500
Total	1000	246	360	394	1000

Figure 11-5 Summary from experiment5

Discussion

From the result, we can demonstrate that we can the server can work with several clients sending purchasing order simultaneously. The server can receive, process and then reply back successfully. One indicator that shows the system can work correctly is the total number of ticket sold at all clients and the total of tickets approved at the server are match. Also, the total number of tickets sold and the number of tickets initially has in the database are equal.

After project presentation in the last class, we modified the client kernel module to generate orders at a random time as suggested. Also we get rid of the use of semaphore at the client user space when the threads want to read from the FIFO. However, we could not get rid of the semaphore of the threads to get access to socket. When we tried to not using the semaphore we could not control which thread should get the reply from the main server. The result of not using the semaphore is shown in figure 12 and 13. From the figure, we can see that replies from the server do not match with the transaction number. This is because we could not control the threads when to read and display the message correctly. When we use the semaphore, we do not observe this problem any more.

In conclusion, we have created a ticket purchasing system working with Faircom database server successfully. We have created a kernel model and a use space program from the clients and another user space program for the main server. The main server can communicate with Faircom database and process the orders correctly. For the clients, they can send purchasing orders and receive messages from the main server without any problem. All clients can accept orders at the same time and have the main server process the order without any issues. We can use the switches on the auxiliary board to start and stop the kernel module to send orders to the clients, and also the threads in the kernel can produce orders at a random period of time.

```
csqnf@nfs1:~  
Ticket 1 remaining: 41  
Ticket 2 remaining: 142  
Trying to sell 2 order(s) of ticket 2.....  
Ticket 1 remaining: 41  
Ticket 2 remaining: 140  
Received: # 13 id 3 ticket 2 qty 3 trns 63  
Ticket 1 remaining: 41  
Ticket 2 remaining: 140  
Trying to sell 3 order(s) of ticket 2.....  
Ticket 1 remaining: 41  
Ticket 2 remaining: 137  
Received: # 13 id 1 ticket 2 qty 3 trns 64  
Ticket 1 remaining: 41  
Ticket 2 remaining: 137  
Trying to sell 3 order(s) of ticket 2.....  
Ticket 1 remaining: 41  
Ticket 2 remaining: 134  
Received: # 13 id 1 ticket 2 qty 2 trns 65  
Ticket 1 remaining: 41  
Ticket 2 remaining: 134  
Trying to sell 2 order(s) of ticket 2.....  
Ticket 1 remaining: 41  
Ticket 2 remaining: 132
```

Figure 12 a screenshot showing of the main server when the program does not use semaphore

```
csqnf@nfs1:~  
  
Sent: # 13 id 2 ticket 2 qty 2 trns 62  
Received: Approved trns:60  
  
Sent: # 13 id 3 ticket 2 qty 3 trns 63  
Received: Approved trns:61  
  
Sent: # 13 id 1 ticket 2 qty 3 trns 64  
Received: Approved trns:62  
  
Sent: # 13 id 1 ticket 2 qty 2 trns 65  
Received: Approved trns:63  
  
Received: Approved trns:64  
  
Received: Approved trns:65
```

Figure 13 a screenshot showing of the main server when the program does not use semaphore

CODE

project1kernel.c

```
#ifndef MODULE
#define MODULE
#endif

#ifndef __KERNEL__
#define __KERNEL__
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/io.h>
#include <rtai_sem.h>
#include <rtai_sched.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <rtai_fifos.h>
#include <linux/random.h>

MODULE_LICENSE("GPL");

unsigned long *ptr, *pbdr, *pbddr;
static RT_TASK task1, task2, task3, task4;
static SEM semap;
RTIME period;
int x1, x2, x3, P1, P2, P3;
//RTIME ttt;

//structure for FIFO data
typedef struct str_purche
{
    int m_id, qty;
} purchaseStr;

static void Thread1(int data)
{
    purchaseStr order_detail;
    int swflag;

    //prepare order
    order_detail.m_id = data;
    order_detail.qty = 1;
    while(1)
```

```

    {
        //do stuff

        if((*pbdr | 0xE0) == 0xFE) swflag = 1;
        if((*pbdr | 0xE0) == 0xEF) swflag = 0;
        if(swflag == 1){
            int status = rtf_put(0, &order_detail, sizeof(order_detail));
        }

        //printf("Put %d with status %d\n",timedata, status);

        //rt_sleep(nano2count(1000000));

        rt_task_wait_period();

        //Random and set a new time period to the thread
        get_random_bytes(&x1, sizeof(x1)); //random number
        P1 = (x1 % 100)+1; //set it to 1-100
        //set new time period
        if(data == 1)
            rt_task_make_periodic(&task1, rt_get_time() + 10*period,
P1*period);
        else if(data == 2)
            rt_task_make_periodic(&task2, rt_get_time() + 10*period,
P1*period);
        else rt_task_make_periodic(&task3, rt_get_time() + 10*period,
P1*period);
    }
}

int init_module(void)
{
    purchaseStr myData;

    //setup PORT B
    ptr = (unsigned long*)__ioremap(0x80840000,4096,0);
    pbdr = ptr + 1;
    pbddr = ptr + 5;
    *pbddr = 0xE0; //Setup input-output

    rtf_create(0,1*sizeof(myData)); //create fifo 0

    rt_set_periodic_mode();
    period = start_rt_timer(nano2count(1000000)); //set base time to 1ms

    //Create real time task
    rt_task_init(&task1, Thread1, 1,256, 0, 0, 0);

```

```

    rt_task_make_periodic(&task1, rt_get_time()+10*period, 1*period);
    rt_task_init(&task2, Thread1, 2,256, 0, 0, 0);
    rt_task_make_periodic(&task2, rt_get_time()+10*period, 1*period);
    rt_task_init(&task3, Thread1, 3,256, 0, 0, 0);
    rt_task_make_periodic(&task3, rt_get_time()+10*period, 1*period);

    rt_task_resume(&task1);
    rt_task_resume(&task2);
    rt_task_resume(&task3);

    return 0;
}

void cleanup_module(void)
{
    rt_task_delete(&task1);
    rt_task_delete(&task2);
    rt_task_delete(&task3);

    stop_rt_timer();
}

```

project1client.c

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>

#include <fcntl.h>
#include <pthread.h>
#include <semaphore.h>
#include <ifaddrs.h>

#define MSG_SIZE 40                                // message size

```

```

typedef struct str_purche    // data structure for fifo
{
    int m_id, qty;
} purchaseStr;

sem_t sem1,sem2;

int sock, n;
int transac_id;
unsigned int length;
struct sockaddr_in anybody, from;
//char buffer[MSG_SIZE];    // to store received messages or messages to be sent.
int boolval = 1;            // for a socket option

int fd_fifo_in ,read_status; //variables for FIFO
purchaseStr fifobuffer;

int my_board;

void error(const char *msg)
{
    perror(msg);
    exit(0);
}

void thtask(void *ptr){

    char buffer[MSG_SIZE];

    do
    {
        //sem_wait(&sem1);

        read_status = read(fd_fifo_in,&fifobuffer,sizeof(fifobuffer));
        if (read_status < 0){
            printf("FIFO read error\n");
            exit(1);
        }

        transac_id++; //counting transaction ID

        sprintf(buffer, "# %d id %d ticket %d qty %d trns %d\n",my_board,
fifobuffer.m_id,(rand() % 2) + 1, (rand() % 3) + 1, transac_id);
        //printf("received order from id %d, qty = %d\n",fifobuffer.m_id,
fifobuffer.qty);

```

```

        //printf("%s", buffer);

    //    sem_post(&sem1);

    sem_wait(&sem2); //grab semaphore

    // send request to the server
    n = sendto(sock, buffer, strlen(buffer), 0,
               (const struct sockaddr *)&anybody, length);
    if (n < 0)
        error("Sendto");
    printf("Sent: %s", buffer);

    // receive message back
    n = recvfrom(sock, buffer, MSG_SIZE, 0, (struct sockaddr *)&from,
&length);
    if (n < 0)
        error("recvfrom");
    printf("Received: %s \n", buffer);

    sem_post(&sem2); //releaes semaphore

} while (buffer[0] != '!'); //make it run forever
}

int main(int argc, char *argv[])
{

    pthread_t thread1, thread2, thread3;

    /*if (argc != 2)
    {
        printf("usage: %s port\n", argv[0]);
        exit(1);
    }*/

    ///init semaphore
    sem_init(&sem1, 0, 1);
    sem_init(&sem2, 0, 1);

    transac_id = 0;

    /*****Find my board number*****/
    struct ifaddrs * ifAddrStruct=NULL;
    struct ifaddrs * ifa=NULL;
    void * tmpAddrPtr=NULL;

```

```

char addressBuffer[INET_ADDRSTRLEN];
char addressBuffer_tmp[INET_ADDRSTRLEN];

char *boardno_s;

getifaddrs(&ifAddrStruct);

ifa = ifAddrStruct;
ifa = ifa->ifa_next;
if (ifa->ifa_addr->sa_family==AF_INET) { // check it is IP4
    // is a valid IP4 Address
    tmpAddrPtr=&((struct sockaddr_in *)ifa->ifa_addr->sin_addr;
    //char addressBuffer[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, tmpAddrPtr, addressBuffer, INET_ADDRSTRLEN);
    printf("My IP is %s\n", addressBuffer);
    //IPaddress = addressBuffer;
}

strcpy(addressBuffer_tmp, addressBuffer);
boardno_s = strtok(addressBuffer_tmp, ".");
boardno_s = strtok(NULL, ".");
boardno_s = strtok(NULL, ".");
boardno_s = strtok(NULL, ".");
my_board = atoi(boardno_s);
printf("My board no. is %d\n", my_board);

if (ifAddrStruct!=NULL) freeifaddrs(ifAddrStruct);

/*****Create Socket*****/

sock = socket(AF_INET, SOCK_DGRAM, 0); // Creates socket. Connectionless.
if (sock < 0)
    error("socket");

// change socket permissions to allow broadcast
if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &boolval, sizeof(boolval)) <
0)
{
    printf("error setting socket options\n");
    exit(-1);
}

anybody.sin_family = AF_INET; // symbol constant for Internet domain
//anybody.sin_port = htons(atoi(argv[1])); // port field
anybody.sin_port = htons(3000); //using PORT 3000

```



```

anybody.sin_addr.s_addr = inet_addr("10.3.52.255");    // broadcast address

length = sizeof(struct sockaddr_in);                    // size of structure

////////// Initialize FIFO //////////

fd_fifo_in = open("/dev/rtf/0",O_RDWR);
if (fd_fifo_in < 0){
    printf("can't open FIFO\n");
    exit(1);
}
////////////////////////////////////

//////////create threads//////////
pthread_create(&thread1, NULL, (void *)&ttask, NULL);
pthread_create(&thread2, NULL, (void *)&ttask, NULL);
pthread_create(&thread3, NULL, (void *)&ttask, NULL);

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);

///NEVER GET TO THIS POINT///
close(sock);                                           // close socket.
return 0;
}

```

project1.c

```

#ifdef _WIN32_WCE
#undef UNICODE
#undef _UNICODE
#define main my_main
#endif

/* Preprocessor definitions and includes */

#include "ctdbsdk.h" /* c-tree headers */

#define LOCK_SUPPORT

#ifndef ctCLIENT
#ifndef NOTFORCE
#undef LOCK_SUPPORT

```

```

#endif
#endif

#define END_OF_FILE INOT_ERR /* INOT_ERR is ctree's 101 error. See cterrc.h */

////////// MY ADDED //////////
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <ifaddrs.h>

#define MSG_SIZE 40 // message size

int data_tmp,order_qty,order_ticket;
int status_flag;

void error(const char *msg)
{
    perror(msg);
    exit(0);
}
//////////

/* Global declarations */

CTHANDLE hSession;
CTHANDLE hDatabase;
CTHANDLE hTable;
CTHANDLE hRecord;

/* Function declarations */

#ifdef PROTOTYPE
VOID Initialize(VOID), Define(VOID), Manage(VOID), Done(VOID);
VOID Add_Ticket_Records(VOID), Display_Records(VOID);
VOID Update_Ticket_Record(VOID), Create_Ticket_Table(VOID);
VOID Delete_Records(CTHANDLE), Check_Table_Mode(CTHANDLE);

```

```

VOID Handle_Error(CTSTRING);
#else
VOID Initialize(), Define(), Manage(), Done();
VOID Add_Ticket_Records(), Display_Records();
VOID Delete_Records(), Check_Table_Mode();
VOID Update_Ticket_Record(), Create_Ticket_Table();
VOID Handle_Error();
#endif

/*
 * main()
 *
 * The main() function implements the concept of "init, define, manage
 * and you're done..."
 */

#ifdef PROTOTYPE
NINT main (NINT argc, pTEXT argv[])
#else
NINT main (argc, argv)
NINT argc;
TEXT argv[];
#endif
{

        ///// declare variables for socket /////
        int sock, length, n;
        int boolval = 1; // for a socket option
        socklen_t fromlen;
        struct sockaddr_in server;
        struct sockaddr_in addr;
        char buffer[MSG_SIZE]; // to store received messages or messages to
be sent.

        char s_buffer[MSG_SIZE];
        char station_s[5], id_s[5], qty_s[5], ticket_s[5], transcation[5];
        char *token_s;

        order_ticket = 1;
        //////////////////////////////////////

        ///////////Initialize FairCom Database//////////

        Initialize();

```

```

        Define();

    Manage();

    //////////Initialize Socket////////

        sock = socket(AF_INET, SOCK_DGRAM, 0); // Creates socket.
Connectionless.
        if (sock < 0)
            error("Opening socket");

        length = sizeof(server);           // length of structure
        bzero(&server,length);             // sets all values to zero.
memset() could be used
        server.sin_family = AF_INET;       // symbol constant for
Internet domain
        server.sin_addr.s_addr = INADDR_ANY; // IP address of the
machine on which

        // the server is running
        server.sin_port = htons(3000);     // using port 3000

        // binds the socket to the address of the host and the port number
        if (bind(sock, (struct sockaddr *)&server, length) < 0)
            error("binding");

        // change socket permissions to allow broadcast
        if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &boolval,
sizeof(boolval)) < 0)
        {
            printf("error setting socket options\n");
            exit(-1);
        }

        fromlen = sizeof(struct sockaddr_in); // size of structure

    //////////Receive and handle the orders////////
while (1)
{
    // to "clean up" the buffer. The messages aren't always the same length...
    bzero(buffer,MSG_SIZE);

    // receive from a client
    n = recvfrom(sock, buffer, MSG_SIZE, 0, (struct sockaddr *)&addr,
&fromlen);

```

```

        if (n < 0)
            error("recvfrom");
    printf("Received: %s", buffer);

    //usleep(10000);

    token_s = strtok(buffer, " ");
    token_s = strtok(NULL, " ");
    strcpy(station_s, token_s); //Station ID
    token_s = strtok(NULL, " ");
    token_s = strtok(NULL, " ");
    strcpy(id_s, token_s); //Thread ID from a client
    token_s = strtok(NULL, " ");
    token_s = strtok(NULL, " ");
    strcpy(ticket_s, token_s); //Ticket requested
    token_s = strtok(NULL, " ");
    token_s = strtok(NULL, " ");
    strcpy(qty_s, token_s); //Quantity requested
    token_s = strtok(NULL, " ");
    token_s = strtok(NULL, " ");
    strcpy(transcation, token_s); //Transaction ID
    //printf("Received order from station%s id%s ticket:%s qty:%s\n", station_s,
id_s,ticket_s, qty_s);

    order_qty = atoi(qty_s);
    order_ticket = atoi(ticket_s);

    /* display and check total ticket left */
    Display_Records();

    /* update ticket number */
    Update_Ticket_Record();

    /* Show total left*/
    Display_Records();

    if(status_flag == 1) //this flag is from Update_ticket_record()
        //sprintf(s_buffer,"%s order(s) of ticket %s done\n",qty_s,ticket_s);
        sprintf(s_buffer,"Approved trns:%s\n", transcation);
    else
        sprintf(s_buffer,"Ticket %s SOLD OUT!\n",ticket_s);
    n = sendto(sock, s_buffer, 40, 0,
    (struct sockaddr *)&addr, fromlen); //send result to client
    if (n < 0)

```

```

        error("sendto");
    }

    return(0);
}

/*
 * Initialize()
 *
 * Perform the minimum requirement of logging onto the c-tree Server
 */

#ifdef PROTOTYPE
VOID Initialize(VOID)
#else
VOID Initialize()
#endif
{
    CTDBRET  retval;

    printf("INIT\n");

    if ((retval = ctdbStartDatabaseEngine())) /* This function is
required when you are using the Server DLL model to start the underlying Server. */
        Handle_Error("Initialize(): ctdbStartDatabaseEngine()"); /* It does nothing in
all other c-tree models */

    /* allocate session handle */
    if ((hSession = ctdbAllocSession(CTSESSION_CTREE)) == NULL)
        Handle_Error("Initialize(): ctdbAllocSession()");

    hDatabase = hSession; /* database not used in this tutorial */

    /* connect to server */
    printf("\tLogon to server...\n");
    if (ctdbLogon(hSession, "FAIRCOMS", "ADMIN", "ADMIN"))
        Handle_Error("Initialize(): ctdbLogon()");
}

/*
 * Define()
 *
 * Open the table, if they exist. Otherwise create and open the table
 */

```

```

#ifdef PROTOTYPE
VOID Define(VOID)
#else
VOID Define()
#endif
{
    printf("DEFINE\n");

    Create_Ticket_Table();
}

/*
 * Manage()
 *
 * This function performs record adds and updates using locking
 */

#ifdef PROTOTYPE
VOID Manage(VOID)
#else
VOID Manage()
#endif
{
    printf("MANAGE\n");

    /* allocate a record handle */
    if ((hRecord = ctdbAllocRecord(hTable)) == NULL)
        Handle_Error("Manage(): ctdbAllocRecord()");

    /* delete any existing records */
    Delete_Records(hRecord);

    /* populate the table with data */
    Add_Ticket_Records();

    /* display contents of table */
    Display_Records();
}

/*
 * Done()
 *
 * This function handles the housekeeping of closing tables and

```

```

* freeing of associated memory
*/

#ifdef PROTOTYPE
VOID Done(VOID)
#else
VOID Done()
#endif
{
    printf("DONE\n");

    /* close table */
    printf("\tClose table\n");
    if (ctdbCloseTable(hTable))
        Handle_Error("Done(): ctdbCloseTable()");

    /* logout */
    printf("\tLogout...\n");
    if (ctdbLogout(hSession))
        Handle_Error("Done(): ctdbLogout()");

    /* free handles */
    ctdbFreeRecord(hRecord);
    ctdbFreeTable(hTable);
    ctdbFreeSession(hSession);

    /* If you are linked to the Server DLL, then we should stop our Server at the end of the
    program. */
    /* It does nothing in all other c-tree models */
    ctdbStopDatabaseEngine();
}

/*
* Create_CustomerMaster_Table()
*
* Open table CustomerMaster, if it exists. Otherwise create it
* along with its indices and open it
*/

#ifdef PROTOTYPE
VOID Create_Ticket_Table(VOID)
#else
VOID Create_Ticket_Table()
#endif
{

```



```

CTHANDLE pField1, pField2;
CTHANDLE pIndex;
CTHANDLE pIseg;

/* define table CustomerMaster */
printf("\tcreate Ticket table\n");

/* allocate a table handle */
if ((hTable = ctdbAllocTable(hDatabase)) == NULL)
    Handle_Error("Create_Ticket_Table(): ctdbAllocTable()");

/* open table */
if (ctdbOpenTable(hTable, "tblticket", CTOPEN_NORMAL))
{
    /* define table fields */
    pField1 = ctdbAddField(hTable, "ticket_id", CT_STRING, 4);
    pField2 = ctdbAddField(hTable, "ticket_qty", CT_STRING, 10);

    if (!pField1 || !pField2 )
        Handle_Error("Create_Ticket_Table(): ctdbAddField()");

    /* define index */
    pIndex = ctdbAddIndex(hTable, "tblticket_idx", CTINDEX_FIXED, NO, NO);
    pIseg = ctdbAddSegment(pIndex, pField1, CTSEG_SCHSEG);
    if (!pIndex || !pIseg)
        Handle_Error("Create_Ticket_Table(): ctdbAddIndex()|ctdbAddSegment()");

    /* create table */
    if (ctdbCreateTable(hTable, "tblticket", CTCREATE_NORMAL))
        Handle_Error("Create_Ticket_Table(): ctdbCreateTable()");

    /* open table */
    if (ctdbOpenTable(hTable, "tblticket", CTOPEN_NORMAL))
        Handle_Error("Create_Ticket_Table(): ctdbOpenTable()");
}
else
{
    Check_Table_Mode(hTable);

    /* confirm the index exists, if not then add the index
    *
    * this scenario arises out of the fact that this table was created in tutorial 1
    * without indexes. The index is now created by the call to ctdbAlterTable
    */

    if (ctdbGetIndexByName(hTable, "tblticket_idx") == NULL)

```

```

    {
        pField1 = ctdbGetFieldByName(hTable, "tblticket");
        pIndex = ctdbAddIndex(hTable, "tblticket_idx", CTINDEX_FIXED, NO, NO);
        pIseg = ctdbAddSegment(pIndex, pField1, CTSEG_SCHSEG);
        if (!pIndex || !pIseg)
            Handle_Error("Create_Ticket_Table(): ctdbAddIndex()|ctdbAddSegment()");

        if (ctdbAlterTable(hTable, CTDB_ALTER_NORMAL) != CTDBRET_OK)
            Handle_Error("Create_Ticket_Table(): ctdbAlterTable()");
    }
}
}

/*
 * Check_Table_Mode()
 *
 * Check if existing table has transaction processing flag enabled.
 * If a table is under transaction processing control, modify the
 * table mode to disable transaction processing
 */

#ifdef PROTOTYPE
VOID Check_Table_Mode(CTHANDLE hTable)
#else
VOID Check_Table_Mode(hTable)
CTHANDLE hTable;
#endif
{
    CTCREATE_MODE mode;

    /* get table create mode */
    mode = ctdbGetTableCreateMode(hTable);

    /* check if table is under transaction processing control */
    if ((mode & CTCREATE_TRNLOG))
    {
        /* change file mode to disable transaction processing */
        mode ^= CTCREATE_TRNLOG;
        if (ctdbUpdateCreateMode(hTable, mode) != CTDBRET_OK)
            Handle_Error("Check_Table_Mode(); ctdbUpdateCreateMode");
    }
}

/*

```

```

* Delete_Records()
*
* This function deletes all the records in the table
*/

#ifdef PROTOTYPE
VOID Delete_Records(CTHANDLE hRecord)
#else
VOID Delete_Records(hRecord)
CTHANDLE hRecord;
#endif
{
    CTDBRET  retval;
    CTBOOL   empty;

    printf("\tDelete records...\n");

    /* enable session-wide lock flag */
    if (ctdbLock(hSession, CTLOCK_WRITE_BLOCK))
        Handle_Error("Delete_Records(): ctdbLock()");

    empty = NO;
    retval = ctdbFirstRecord(hRecord);
    if (retval != CTDBRET_OK)
    {
        if (retval == END_OF_FILE)
            empty = YES;
        else
            Handle_Error("Delete_Records(): ctdbFirstRecord()");
    }

    while (empty == NO) /* while table is not empty */
    {
        /* delete record */
        if (ctdbDeleteRecord(hRecord))
            Handle_Error("Delete_Records(): ctdbDeleteRecord()");

        /* read next record */
        retval = ctdbNextRecord(hRecord);
        if (retval != CTDBRET_OK)
        {
            if (retval == END_OF_FILE)
                empty = YES;
            else
                Handle_Error("Delete_Records(): ctdbNextRecord()");
        }
    }
}

```

```

    }
    if (ctdbUnlock(hSession))
        Handle_Error("Delete_Records(): ctdbUnlock()");
}

/*
 * Add_CustomerMaster_Records()
 *
 * This function adds records to table CustomerMaster from an
 * array of strings
 */

typedef struct {
    CTSTRING id, qty;
} CUSTOMER_DATA;

////////// Initialize total ticket number //////////
CUSTOMER_DATA data[] = {
    "1", "500",
    "2", "500"
};
//////////

#ifdef PROTOTYPE
VOID Add_Ticket_Records(VOID)
#else
VOID Add_Ticket_Records()
#endif
{
    CTDBRET retval;
    CTSIGNED i;
    CTSIGNED nRecords = sizeof(data) / sizeof(CUSTOMER_DATA);

    printf("\tAdd records...\n");

    /* add data to table */
    for (i = 0; i < nRecords; i++)
    {
        /* clear record buffer */
        ctdbClearRecord(hRecord);

        retval = 0;
        /* populate record buffer with data */
        retval |= ctdbSetFieldAsString(hRecord, 0, data[i].id);
    }
}

```

```

    retval |= ctdbSetFieldAsString(hRecord, 1, data[i].qty);

    if (retval)
        Handle_Error("Add_Ticket_Records(): ctdbSetFieldAsString()");

    /* add record */
    if (ctdbWriteRecord(hRecord))
        Handle_Error("Add_Ticket_Records(): ctdbWriteRecord()");
}
}

/*
 * Display_Records()
 *
 * This function displays the contents of a table. ctdbFirstRecord() and
 * ctdbNextRecord() fetch the record. Then each field is parsed and displayed
 */

#ifdef PROTOTYPE
VOID Display_Records(VOID)
#else
VOID Display_Records()
#endif
{
    CTDBRET retval;
    TEXT    ticketid[4+1];
    TEXT    ticketqty[47+1];
    int x;

    // printf("\tDisplay records...");

    /* read first record */
    retval = ctdbFirstRecord(hRecord);
    if (retval == END_OF_FILE)
        return;

    x = 1;
    while (retval == CTDBRET_OK)
    {
        retval = 0;
        retval |= ctdbGetFieldAsString(hRecord, 0, ticketid, sizeof(ticketid));
        retval |= ctdbGetFieldAsString(hRecord, 1, ticketqty, sizeof(ticketqty));
        if (retval)
            Handle_Error("Display_Records(): ctdbGetFieldAsString()");
    }
}

```

```

//  printf("\n\t\t%-8s%10s\n",ticketid, ticketqty);
printf("Ticket %s remaining: %s\n",ticketid, ticketqty);
if(order_ticket == x)
    data_tmp = atoi(ticketqty);

/* read next record */
retval = ctdbNextRecord(hRecord);
if (retval == END_OF_FILE)
    break; /* reached end of file */

if (retval != CTDBRET_OK)
    Handle_Error("Display_Records(): ctdbNextRecord()");

    x++;
}
}

/*
 * Update_CustomerMaster_Records()
 *
 * Update one record under locking control to demonstrate the effects
 * of locking
 */

#ifdef PROTOTYPE
VOID Update_Ticket_Record(VOID)
#else
VOID Update_Ticket_Record()
#endif
{
    char buffer[10],inx_finder[5];
//  printf("\tUpdate record...\n");

/* enable session-wide lock flag */
if (ctdbLock(hSession, CTLOCK_WRITE_BLOCK))
    Handle_Error("Update_Ticket_Record(): ctdbLock()");

/* find record by customer number */
if (ctdbClearRecord(hRecord))
    Handle_Error("Update_Ticket_Record(): ctdbClearRecord()");

sprintf(inx_finder,"%d",order_ticket);
//  printf("index:%s ",inx_finder);

```

```

//if (ctdbSetFieldAsString(hRecord, 0, "2"))
if (ctdbSetFieldAsString(hRecord, 0, inx_finder))
    Handle_Error("Update_Ticket_Record(): ctdbSetFieldAsString()");
if (ctdbFindRecord(hRecord, CTFIND_EQ))
    Handle_Error("Update_Ticket_Record(): ctdbFindRecord()");

// printf("%d - %d", data_tmp,order_qty);

if(data_tmp-order_qty >= 0){
    data_tmp = data_tmp-order_qty;
    status_flag = 1;
}
else {
    data_tmp = data_tmp;
    status_flag = 0;
}

printf("Trying to sell %d order(s) of ticket %d.....\n", order_qty,order_ticket);

sprintf(buffer,"%d",data_tmp);

if (ctdbSetFieldAsString(hRecord, 1, buffer))
    Handle_Error("Update_Ticket_Record(): ctdbSetFieldAsString()");
/* rewrite record */
if (ctdbWriteRecord(hRecord))
    Handle_Error("Update_Ticket_Record(): ctdbWriteRecord()");
else
{
    //printf("\tPress <ENTER> key to unlock\n");
    //getchar();
}

if (ctdbUnlock(hSession))
    Handle_Error("Update_Ticket_Record(): ctdbUnlock()");
}

/*
* Handle_Error()
*
* This function is a common bailout routine. It displays an error message
* allowing the user to acknowledge before terminating the application
*/

#ifdef PROTOTYPE

```

```

VOID Handle_Error(CTSTRING errmsg)
#else
VOID Handle_Error(errmsg)
CTSTRING errmsg;
#endif
{
    printf("\nERROR: [%d] - %s \n", ctdbGetError(hSession), errmsg);
    printf("*** Execution aborted *** \nPress <ENTER> key to exit...");

    ctdbLogout(hSession);

    ctdbFreeRecord(hRecord);
    ctdbFreeTable(hTable);
    ctdbFreeSession(hSession);

    getchar();

    exit(1);
}

```