OPERATING SYSTEM ARCHITECTURE

Final Project

Computer Science L2

Shoykyat Sharafyabi

Kanan Mikayilov

30/05/20

Objective:

The objective of this project is to create a Haiku generator server. The server is triggered by signals: when it receives a signal, it prints a haiku on the console. Haikus can be of two categories: *'Japanese'* (category 1) or *'western'* (category 2).

Version 1:

Condition:

The first version of Haiku server entails:

- the Haiku server consists of a simple server, which receives signals, and prints to the screen the matching type of haiku to be printed.

- a Haiku client, which randomly chooses one of the two signals 'SIGINT' and 'SIGQUIT' and sends it to the server. This operation is repeated 100 times.

The client and the server are each a dedicated program in C language. The association between signal_type and haiku_category is stored in a matrix. Use: SIGINT for haiku_category: 'Japanese'; SIGQUIT for haiku_category: 'Western'.

Implementation:

Client program:

In the client program, there is created a shared memory segment to take later the PID of the server program, to send signals to the server. And then randomly chosen signals between SIGINT and SIGQUIT are sent to the server.

create_segment():

This function creates a shared memory segment by using unique keys and returns the shared memory id, in our case, it accesses to already existing shared memory segment because it was created in the server.c program.

```
int create_segment(){
    key_t shm_key = ftok ("/dev/null", 65);       // ftok() to create unique key

    int shm_id = shmget (shm_key, sizeof (int), 0644 | IPC_CREAT);
    if (shm_id < 0){
        perror ("shmget\n");
        exit (1);
    }
    return shm_id;
}
```

main():

In the main function, there was created an array of integers with signal numbers inside of it. After there was created a shared memory segment and serverPID was taken from the shared address. Finally, in the 100 iteration loop, the random signals were sent to the serverPID, at each iteration program sleeps 2 seconds. At the end, shared value and shared memory segment are removed.

```
int main(){
      //init the signals to send them later to the server
      int signals[2] = {SIGINT, SIGQUIT};
      srand(time(NULL));
      //creating shared memory to get the pid of the server
      int shm_id = access_segment();
      int *p = (int *) shmat (shm_id, NULL, 0);
      int serverPid = *p;

      char c,h;
      int random;
      printf("\033[31m");
      printf("Sending signals...\n");
      printf("\033[0m");

      //sending signals to the server by randomly choosing them from the array
      for(int i = 0; i < 100; i++){
        if(i > 5){
           printf("You want to quit? (y/n): ");
           scanf("%c", &h);
           scanf("%c", &c);
        }
        if(h == 'Y' || h == 'y') break;

        sleep(1);
        random = rand()%2;
        kill(serverPid, signals[random]);
      }

      kill(serverPid, SIGTSTP);

      printf("\033[31m");
      printf("\nTerminating...\n");
      printf("\033[0m");
      //removing shared memory and shared value
      shmdt(p);
     shmctl (shm_id, IPC_RMID, 0);
      return 0;
}
```

Server program:

This program waits for signals and handles them, for this purpose signal_handlers were added, for SIGINT and SIGQUIT signals to handle them as needed.

POSIX_SIGNAL structure:

The structure consists of integer named **signo** to mention the signal number, and char pointer to mention the category name corresponding to signal number.

```
#define NTAB(t) sizeof(t)/sizeof(POSIX_SIGNAL)
volatile sig_atomic_t count;

typedef struct {
     int signo;       //signal number
     char *catname;   // category name
}POSIX_SIGNAL;

POSIX_SIGNAL tab[] = {
     { SIGINT, "Japanese", },
     { SIGQUIT, "Western", }
};
```

signal_handler(int signo);

This function takes signal number as a parameter and compares it with **signo** of the structure going through the structure array tab[] and if signal numbers are the same, it prints the round, signal name, category name.

```
void signal_handler(int sig){
     for(int i = 0; i < NTAB(tab); i++){
          if(tab[i].signo == sig){
               printf("Round: %d \t Signal: %s \t Haiku category:
%s\n",count, sig == 2 ? "SIGINT " : "SIGQUIT", tab[i].catname);
               break;
          }
     }
     count++;
}
```

main():

In the main function shared memory segment was created to send the pid of the process to the client later. After a sigaction structure object was created to set up the signal handlers for SIGINT and SIGQUIT signals and after the server program just waits for signals in a 100 iteration loop.

In the end, shared memory and shared value were removed.

```
int main(){
     count = 0;
     pid_t serverPid =  getpid(); // getting the pid of the process to send
it to the client

     // creating shared memory segment to seng pid to the client through it
     int shm_id = create_segment();
     int *p = (int *) shmat (shm_id, NULL, 0);
    *p = 0;
    *p += serverPid;

    //sigaction structure to set the signal_handlers to signals
    struct sigaction sigact;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = signal_handler;
    sigaction( SIGINT, &sigact, NULL );
    sigaction( SIGQUIT, &sigact, NULL );


    //waiting for signals
    while(count < 100) {
     pause();
    }

    //removing shared memory and shared value
     shmdt(p);
    shmctl (shm_id, IPC_RMID, 0);
     return 0;
}
```

Execution:

```
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT_OSA_L2/Version1$ ./server
Round: 0        Signal: SIGQUIT         Haiku category: Western
Round: 1        Signal: SIGINT          Haiku category: Japanese
Round: 2        Signal: SIGQUIT         Haiku category: Western
Round: 3        Signal: SIGINT          Haiku category: Japanese
Round: 4        Signal: SIGQUIT         Haiku category: Western
Round: 5        Signal: SIGQUIT         Haiku category: Western
Round: 6        Signal: SIGINT          Haiku category: Japanese
Round: 7        Signal: SIGINT          Haiku category: Japanese
Round: 8        Signal: SIGINT          Haiku category: Japanese
Round: 9        Signal: SIGINT          Haiku category: Japanese
Round: 10       Signal: SIGQUIT         Haiku category: Western

[14]+  Stopped                 ./server
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT_OSA_L2/Version1$
```

```
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT_OSA_L2/Version1$ ./client
Sending signals...
You want to quit? (y/n): n
You want to quit? (y/n): n
You want to quit? (y/n): n
You want to quit? (y/n): n
You want to quit? (y/n): n
You want to quit? (y/n): y

Terminating...
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT_OSA_L2/Version1$
```

Version 2:

Condition:

The second version of Haiku server entails:

- a haiku reader program, which creates a message queue for exchanging haiku text. All haikus are stored in two files, one for Japanese Haikus, one for western Haikus. A method read_haiku(int category) reads the message queue of type 1 when the category is 1 ('Japanese') and the message queue of type 2 when the category is 2 ('western') In version 2, the reader must read 3 haikus of each type.

- a haiku writer program, which goes through the various haiku categories (represented as an array). Haiku_writer is a dedicated program. First, store the haikus in two directories: 'Japanese' or 'western', one for each category. Each haiku is in an individual file. The haiku writer writes messages of type '1' that contain Japanese haikus, and of type 2 that contain 'western' haikus. All available haikus must be read from the files and stored in the message queue.

Implementation:

Writer Program:

There are 4 functions implemented:

```
void readline(char *filename, int linenumber, char line[]);

int create_queue();

int write_value( char* text, int category);

void readlinesAndwriteToQueue();
```

In the writer program, there was implemented structure message queue with two attributes: message type **(mtype)** and message text **(mtext)** with array size 256.

```
/*
 * structure for the message queue
 */
typedef struct message {
    long mtype;  //message type
    char mtext[MAX]; //message text
}msg;
```

readline()

It's a function to read the line we mentioned from the file and copy it to the string we passed to the function as a parameter.

```
void readline(char *filename, int linenumber, char line[]){

    FILE* file = fopen(filename, "r");
    // char line[256];
    int i = 0;
    while (fgets(line, MAX, file)) {
        i++;
        if(i == linenumber ){
            break;
        }
    }
    fclose(file);
}
```

create_queue():

It's a function to create a queue with two attributes such as **key** and **msg_id**,

key is a variable to hold the IPC key of a message queue and **msg_id** is a variable to hold message queue id.

Afterward, with the help of ftok() the project pathname was converted and the project identifier converted to a Sys_V(unique key).

And msgget() to create the message queue identifier and to return the associated message with the value of the key argument.

The errors with msgget() and ftok() are checked and in case of some errors, it's shown in the terminal.

```
int create_queue(){
    key_t key; // variable to hold IPC key of message queue
    int id; // variable to hold message queue id
    key = ftok("/dev/null", 65);
    if (key == -1)
        perror (" ftok fails ");
    //creates a message queue and returns a message queue identifier
    id = msgget (key, IPC_CREAT | 0666);
    if (id == -1)
        perror (" msgget fails ");
    return id;
}
```

write_value():

This function was created to write text to the message queue with two categories:

The first category is for Japanese. || The Second category is for Western.

```
int write_value( char* text, int category){
    msg msg;
    int id;

    if ( (id = create_queue () ) == -1 )
        return ( -1);
    else
        printf ("Entering Queue with ID : %d \n" , id);

    msg.mtype = (long) category;
    strncpy(msg.mtext, text, MAX);
    if ( msgsnd ( id , &msg , strlen(msg.mtext) , 0) == -1 ) {
        perror (" msgsnd failed "); return ( -1) ;}
    else {
        printf ("Message:  %sWith category: %ld has been sent Successfully \n\n" ,
msg.mtext , msg.mtype );
        return (0) ;
    }
}
```

readlinesAndWriteToQueue()

This function takes lines from Japanese and western files and puts them into the message queue with a category, which was written by the user.

```
void readlinesAndwriteToQueue()
{
      char *filename[] = {"./japanese.txt", "./western.txt"};
      char line[MAX];

      for(int i=0; i < 2; i++)// categories
      {

            for(int k=1; k < 4; k++){
                  readline(filename[i], k, line);
                  write_value(line, i+1);
            }
      }
}
```

Reader Program:

access_queue()

This function is used to access an already existing message queue. There are also **key** and **msg_id** attributes, the usage of them has already explained above.

There is msgget(), which looks for an existing message queue and returns a message queue identifier, which is associated with the value of the key argument.

```
int access_queue(){
    key_t key; // variable to hold IPC key of the message queue
    int id; // variable to hold message queue id
    key = ftok ("/dev/null", 65); // converts a pathname and a project
identifier to a (Sys_V) IPC-key (unique key)
    if (key == -1)
        perror (" ftok fails ");

    //looks for an existing message queue
    // and returns a message queue identifier associated with the value of the
key argument
    id = msgget (key, 0);
    if (id == -1)
        perror (" msgget fails ");
    return id;
}
```

read_value():

The read_value() reads messages from the message queue with a known type.

Inside of the “if” loop message id accesses an existing message and inside another “if” loop, it receives message queue from the queue with a given id. msgrcv() function takes as arguments message queue id, address of the message, max size of the message, type of message we want to get from the queue, and to get error messages if there are no messages in the queue, we mentioned IPC_NOWAIT in the parameters to msgrcv() function.

```
void read_value(int type){
    msg m;
    int n, id;

    if ( ( id = access_queue () ) == -1) //accessing an already existing
message queue
        exit (0) ;
    else
        printf ( "Accessing Queue with ID : %d \n" , id );

    printf("Received message: \n");

    // receive a message m from a message queue with id = id
    if ( ( n = msgrcv ( id , &m , MAX , type, IPC_NOWAIT ) ) == -1 ){
        perror ( "msgrcv");
        exit(-1) ;
    } else {
        m.mtext[n] = '\0';
        printf ( "\t Category: %s\n\t Message: %s" , m.mtype == 1 ? "Japanese"
: "Western", m.mtext ) ;
    }

}
```

main()

In the main function, the user can only choose two categories, otherwise, it prints an error message.

```
int main(){
    int category;
    printf("1 for Japanese || 2 for Western\n");
    printf("Enter the category number you want to read the messages from
message queue: ");
    while(1){
        scanf("%d", &category);
        if(category <= 2 && category > 0) break;
        else
            printf("The category should be either 1 or 2: ");
    }
    read_value(category);
    return 0;
}
```

Execution:

Inside of version two, there was created a MakeFile, which helps to reduce the time spent to execute the program :)

```
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT/Version2$ make
gcc reader.c -o reader
gcc writer.c -o writer
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT/Version2$ ./writer
Entering Queue with ID : 4
Message:  Autumn moonlight - a worm digs silently into the chestnut.
With category: 1 has been sent Successfully

Entering Queue with ID : 4
Message:  Old pond a frog jumps the sound of water.
With category: 1 has been sent Successfully

Entering Queue with ID : 4
Message:  Autumn wind - mountain's shadow wavers.
With category: 1 has been sent Successfully

Entering Queue with ID : 4
Message:  Snow in my shoe Abandoned Sparrow's nest
With category: 2 has been sent Successfully

Entering Queue with ID : 4
Message:  Whitecaps on the bay: A broken signboard banging In the April wind.
With category: 2 has been sent Successfully

Entering Queue with ID : 4
Message:  Lily: out of the water out of itself
With category: 2 has been sent Successfully

kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT/Version2$
```

```
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT/Version2$ ./reader
1 for Japanese || 2 for Western
Enter the category number you want to read the messages from message queue: 2
Accessing Queue with ID : 4
Received message:
        Category: Western
        Message: Snow in my shoe Abandoned Sparrow's nest
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT/Version2$ ./reader
1 for Japanese || 2 for Western
Enter the category number you want to read the messages from message queue: 2
Accessing Queue with ID : 4
Received message:
        Category: Western
        Message: Whitecaps on the bay: A broken signboard banging In the April wind.
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT/Version2$ ./reader
1 for Japanese || 2 for Western
Enter the category number you want to read the messages from message queue: 1
Accessing Queue with ID : 4
Received message:
        Category: Japanese
        Message: Autumn wind - mountain's shadow wavers.
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT/Version2$ ./reader
1 for Japanese || 2 for Western
Enter the category number you want to read the messages from message queue: 1
Accessing Queue with ID : 4
Received message:
        Category: Japanese
        Message: Autumn moonlight - a worm digs silently into the chestnut.
kanan@kanan:~/Desktop/L2_S2_2019/Operating_System_Architecture/PROJECT/Version2$
```

Version 3:

Condition:

The third version of Haiku server binds the bricks from version 1 and version 2, to print a haiku of the given type when the matching signal is received.

-In Haiku server, the method read_haiku(int category) from the reader is called which suitable value for the category when the matching signal is received: SIGINT for haiku_category 1: 'Japanese'; SIGQUIT for haiku_category 2: 'western'.

- In Haiku writer, check that each category entails an entry. Otherwise, refill it with the haikus read from the file.

Server program:

This program waits for signals and according to them reads the messages from queue, if there is no message in the queue the program refills the message queue. Almost all of the functions were taken from previous versions, so many of them are identical, some changes were added.

```
// structure for signal
typedef struct {
     int signo;
     int catnum;
}POSIX_SIGNAL;

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

//structure for the message queue
typedef struct message {
    long mtype;  //message type
    char mtext[MAX]; //message text
}msg;

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

POSIX_SIGNAL tab[] = {
     { SIGINT, 1 },
     { SIGQUIT, 2 }
};
```

create_segment() , readline(), create_queue(), access_queue(), write_value() functions are the same as in the first and second versions.

readlinesAndwriteToQueue():
This function updated and now it takes as a parameter category and reads the lines from one category at a time and writes them to the queue.

```
void readlinesAndwriteToQueue(int id, int cat)
{
      char *filename[] = {"./japanese.txt", "./western.txt"};
      char line[MAX];

      for(int k=1; k < 9; k++){

        if(cat == 2 && k>6) break;
            readline(filename[cat-1], k, line);
            write_value(id, line, cat);
      }

}
```

read_value():
This function is almost the same as in version 2, but now in case that there will not be any messages of desired type in the message queue, the function calls write_value() function by passing the desired type.

```
void read_value(int id, int type){
    msg m;
    int n;

    printf ( "Accessing Queue with ID : %d \n" , id );

    printf("Received message: \n");

    // receive a message m from a message queue with id = id
    if ( ( n = msgrcv ( id , &m , MAX , type, IPC_NOWAIT ) ) == -1 ){
        perror ( "msgrcv");
        readlinesAndwriteToQueue(id, type);
        printf("Loading messages to queue...\n");
        printf("Repeat the action\n\n");
        // exit(-1) ;
    } else {
        m.mtext[n] = '\0';
        printf ( "\t Category: %s\n\t Message: %s" , m.mtype == 1 ? "Japanese" :
"Western", m.mtext ) ;
    }

}
```

signal_handler():
This function is also almost the same as in version 1, just in this version we call read_value() function by passing category corresponding to the signal.

```
void signal_handler(int sig){
    int category;

    int id = access_queue();
    if(id == -1)
        exit(0);


    for(int i = 0; i < NTAB(tab); i++){
        if(tab[i].signo == sig){
            category = tab[i].catnum;
            break;
        }
    }

    if(sig == SIGTSTP){
        remove_queue(id);
        kill(getpid(), SIGKILL);
    }

    read_value(id, category);
}
```

main():
In this function, the pid of the process is sent by a shared memory segment to the client, then the sigaction was set to handle some signals, and after the program waits for the signals.

```
int main(){

     int q_id;
     pid_t serverPid =  getpid(); // getting the pid of the process
     // creating shared memory segment to seng pid to the client through it
     int shm_id = create_segment();
     int *p = (int *) shmat (shm_id, NULL, 0);
    *p = 0;
    *p += serverPid;
```

```
    if(q_id = create_queue() == -1)
        return (-1);

    //sigaction structure to set the signal_handlers to signals
    struct sigaction sigact;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = signal_handler;
    sigaction( SIGINT, &sigact, NULL );
    sigaction( SIGQUIT, &sigact, NULL );
    sigaction( SIGTSTP, &sigact, NULL );

    //waiting for signals
    while(1) {
      pause();
    }
      return 0;
}
```

Client program:

signal_handler():
In the client program, there is signal_handler to make the program catch the signals and after sending them to the server, in the case if client program was stopped by SIGTSTP signal, this signal is sent to the server also to terminate it, and after the client process is killed by SIGKILL signal.

```
//function to handle the signals SIGINT, SIGQUIT, SIGTSTP

void signal_handler(int sig){

    int shm_id = access_segment();
    int *p = (int *) shmat (shm_id, NULL, 0);
    int serverPid = *p;

    kill(serverPid, sig);

    pid_t pid = getpid();
    if(sig == SIGTSTP) {
        //removing shared memory and shared value
        shmdt(p);
```

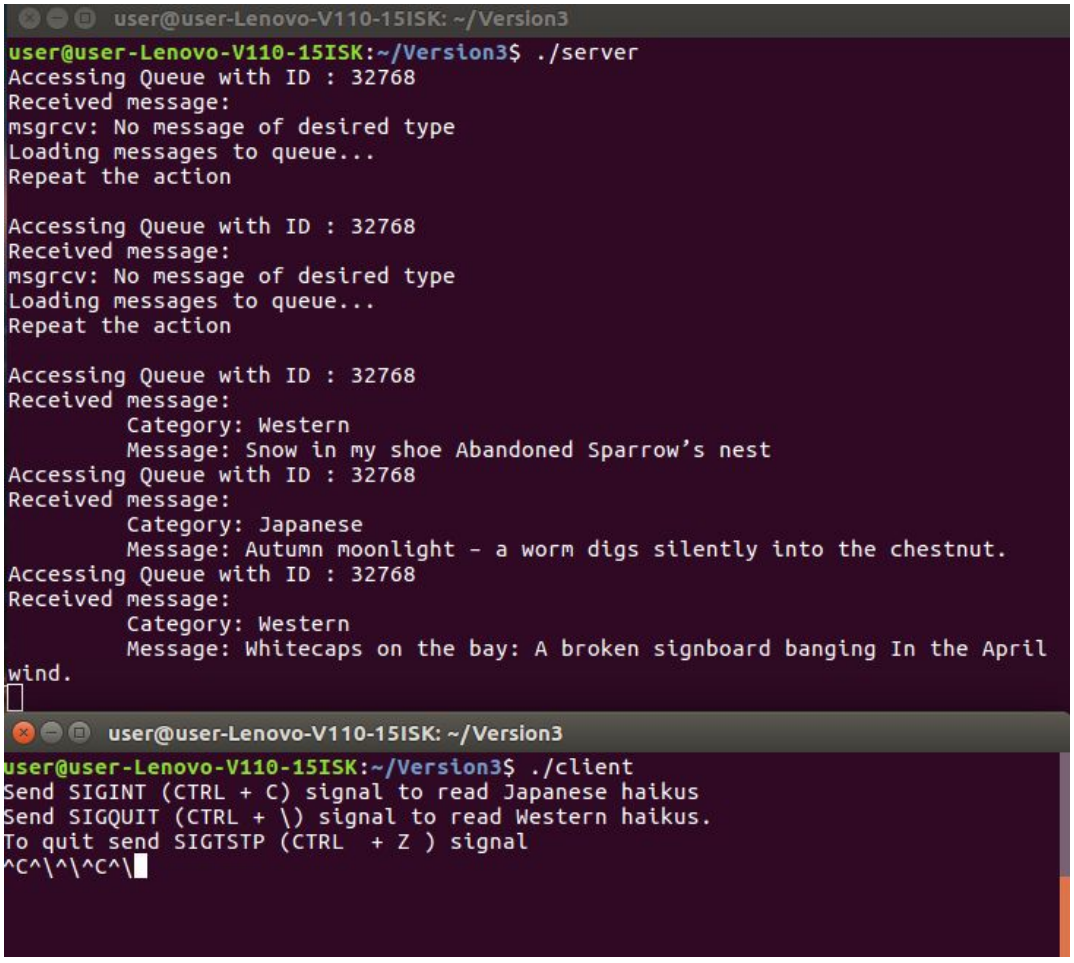
```
        shmctl (shm_id, IPC_RMID, 0);
        // printf("%d\n", pid);
        kill(pid, SIGKILL);
    }
}
```

main():
In the main function we set the sigaction to handle some signals and after the program waits for signals.

```
int main(){

    //sigaction structure to set the signal_handlers to signals
    struct sigaction sigact;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = signal_handler;
    sigaction( SIGINT, &sigact, NULL );
    sigaction( SIGQUIT, &sigact, NULL );
    sigaction( SIGTSTP, &sigact, NULL );
    printf("Send SIGINT (CTRL + C) signal to read Japanese haikus\nSend SIGQUIT (CTRL +
\\) signal to read Western haikus.\n");
    printf("To quit send SIGTSTP (CTRL  + Z ) signal\n");

    int shm_id = access_segment();

    //sending signals to the server by randomly choosing them from the array
    while(1){
        pause();
    }
    return 0;
}
```

Execution:

When we run server and client and then the client sends signals of both types, in the message queue there are no messages of any type, we send both signals first to make the server send messages to the queue, and after when we send these signals everything works fine. In addition, when user terminates from the client program by clicking **CTRL+Z** server terminates also.


user@user-Lenovo-V110-15ISK: ~/Version3
user@user-Lenovo-V110-15ISK:~/Version3$ ./server
Accessing Queue with ID : 32768
Received message:
msgrcv: No message of desired type
Loading messages to queue...
Repeat the action
Accessing Queue with ID : 32768
Received message:
msgrcv: No message of desired type
Loading messages to queue...
Repeat the action
Accessing Queue with ID : 32768
Received message:
Category: Western
Message: Snow in my shoe Abandoned Sparrow's nest
Accessing Queue with ID : 32768
Received message:
Category: Japanese
Message: Autumn moonlight - a worm digs silently into the chestnut.
Accessing Queue with ID : 32768
Received message:
Category: Western
Message: Whitecaps on the bay: A broken signboard banging In the April
wind.
user@user-Lenovo-V110-15ISK: ~/Version3
user@user-Lenovo-V110-15ISK:~/Version3$ ./client
Send SIGINT (CTRL + C) signal to read Japanese haikus
Send SIGQUIT (CTRL + \) signal to read Western haikus.
To quit send SIGTSTP (CTRL + Z) signal
^C^\^\^C^\