Universidad Nacional
ARTURO JAURETCHE

Sistemas Distribuidos

Trabajo Práctico N°2

Alumna:Cabral, Sabrina Lourdes.

Los objetivos, del presente trabajo es familiarizar a los estudiantes con los conceptos de procesamiento concurrente y paralelo, en sistemas distribuidos. Para ello se presentaron 4 ejemplos de aplicaciones desarrolladas en C, y en cada ejemplo se utilizó las librerias Pyhread o OpenM. Por otro lado, se introdujo el pincipio de funcionamiento de socket multihilo para la comunicación entre el servidor y el cliente.

Para comenzar a desarrollar el Laboratorio, se descarago el material proporcionado por la catedra, en la carpeta que esta montada en cada maquina virtual, que compone el cluster. Luego, se accedio desde cada vm a la carpeta y se comprobo que los archivos esten.

Maestro

```
root@Linux10:/media/sf_Laboratorio# exi
-bash: exi: orden no encontrada
root@Linux10:/media/sf_Laboratorio# exit
cerrar sesión
linuxtest@Linux10:~$ cd /media/sf_Laboratorio/
```

```
root@Linux10:/media/sf_Laboratorio# ls
c-chatroom-master  ChatRoom  chatroom-master  OpenMP  Pthreads  Sockets_Multihilo
root@Linux10:/media/sf_Laboratorio#
```

Nodo 0

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
linuxtest@Linux10Nodo0:~$ su -
Contraseña:
root@Linux10Nodo0:~# cd /media/sf_Laboratorio/_
```

```
root@Linux10Nodo0:/media/sf_Laboratorio# ls
c-chatroom-master  ChatRoom  chatroom-master  OpenMP  Pthreads  Sockets_Multihilo
root@Linux10Nodo0:/media/sf_Laboratorio# _
```

Nodo 1

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
linuxtest@Linux10Nodo1:~$ su -
Contraseña:
root@Linux10Nodo1:~# cd /media/sf_Laboratorio/_
```

```
root@Linux10Nodo1:/media/sf_Laboratorio# ls
c-chatroom-master  ChatRoom  chatroom-master  OpenMP  Pthreads  Sockets_Multihilo
root@Linux10Nodo1:/media/sf_Laboratorio#
```

Pthread

Para este caso, el ejemplo fue ejecutado en el Maestro, el mismo consistía en la paralelización de la suma de vectores. Para lograr que la suma sea correcta es necesario mantener sincronizados los hilos que se están ejecutando. Es por ello que se

empleó como mecanismo de sincronización Mutex, que permitió que un hilo por vez pueda acceder a guardar en la variable suma, el valor de la suma local de cada hilo.

```
root@Linux10:/media/sf_Laboratorio/Pthreads# ls
phread  pthread.c
root@Linux10:/media/sf_Laboratorio/Pthreads#
```

Para compilar el código se utilizó el comando gcc phread.c -o creaciónHilospthread -lpthread.

```
root@Linux10:/media/sf_Laboratorio/Pthreads# gcc pthread.c -o creacionHilospthread -lpthread
pthread.c: In function 'main':
pthread.c:38:9: warning: implicit declaration of function 'atoi' [-Wimplicit-function-declaration]
  size = atoi(argv[1]);
         ^~~~
root@Linux10:/media/sf_Laboratorio/Pthreads# _
```

Una vez creado el ejecutable, se debe especificar el tamaño de la matriz y el número de hilos que ejecutaran el programa.

```
root@Linux10:/media/sf_Laboratorio/Pthreads# ./creacionHilospthread 4 4
Primer bloque 3 ,3
ultimo bloque 3 ,3
Primer bloque 2 ,2
ultimo bloque 2 ,2
Primer bloque 1 ,1
ultimo bloque 1 ,1
Primer bloque 0 ,0
ultimo bloque 0 ,0
El resultado es: 10.
3.7689208984375 milliseconds
```

En la figura anterior, se puede ver el resultado de la ejecución.

OpenMP

Para el caso de la librería OpenMp se utilizaron dos casos de ejemplo, el primero es un código que crea n cantidad de hilos, el numero de hilos es determinado antes de la ejecución del programa utilizando la siguiente línea OMP_NUM_THREADS=número de hilos, y cado hilo se encarga de imprimir en pantalla los mensajes "Thread en marcha" y "Thread ha terminado", finalmente, se ejecutará una línea que no está paralelizada, que indica el numero del hilo que se ejecutó + 10.

Para compilar el ejemplo se utilizó el siguiente comando gcc ej1openmp.c -o ejemplo1 -fopenmp.

```
root@Linux10:/media/sf_Laboratorio/OpenMP# gcc ej1openmp.c -o ejemplo1 -fopenmp
root@Linux10:/media/sf_Laboratorio/OpenMP# _
```

En caso de no indicar el numero de hilos que se van a ejecutar, se creara por defecto uno.

```
root@Linux10:/media/sf_Laboratorio/OpenMP# ./ejemplo1
Thread 0 de 1 en marcha
El thread 0 ha terminado
A(0) = 10
A(1) = 0
A(2) = 0
A(3) = 0
A(4) = 0
A(5) = 0
A(6) = 0
A(7) = 0
A(8) = 0
A(9) = 0
A(10) = 0
A(11) = 0
A(12) = 0
A(13) = 0
A(14) = 0
A(15) = 0
A(16) = 0
A(17) = 0
A(18) = 0
A(19) = 0
A(20) = 0
A(21) = 0
A(22) = 0
A(23) = 0
```

Ejemplo indicando el numero de hilos que se deben crear.

```
root@Linux10:/media/sf_Laboratorio/OpenMP# export OMP_NUM_THREADS=4
```

```
root@Linux10:/media/sf_Laboratorio/OpenMP# ./ejemplo1
Thread 0 de 4 en marcha
El thread 0 ha terminado
Thread 3 de 4 en marcha
El thread 3 ha terminado
Thread 2 de 4 en marcha
El thread 2 ha terminado
Thread 1 de 4 en marcha
El thread 1 ha terminado
A(0) = 10
A(1) = 11
A(2) = 12
A(3) = 13
A(4) = 0
A(5) = 0
A(6) = 0
A(7) = 0
A(8) = 0
A(9) = 0
A(10) = 0
A(11) = 0
A(12) = 0
A(13) = 0
A(14) = 0
A(15) = 0
A(16) = 0
A(17) = 0
A(18) = 0
A(19) = 0
A(20) = 0
A(21) = 0
A(22) = 0
A(23) = 0
root@Linux10:/media/sf_Laboratorio/OpenMP# _
```

Ejemplo 2 OpenMP

Para la ejecución de del código se utilizó el nodo maestro, y al igual que ejemplo anterior, se empleó el comando gcc eje2plo.c -o ejemplo2 -fopenmp.

```
root@Linux10:/media/sf_Laboratorio/OpenMP# gcc ej2openmp.c -o ejemplo2 -fopenmp
ej2openmp.c: In function ‘main’:
ej2openmp.c:21:9: warning: implicit declaration of function ‘atoi’ [-Wimplicit-function-declaration]
  size = atoi(argv[1]);
         ^~~~
root@Linux10:/media/sf_Laboratorio/OpenMP# _
```

Resultado de la ejecución.

```
root@Linux10:/media/sf_Laboratorio/OpenMP# ./ejemplo2 2 2
El resultado es 4
root@Linux10:/media/sf_Laboratorio/OpenMP#
```

Socket

```
root@Linux10:/media/sf_Laboratorio/OpenMP# cd ..
root@Linux10:/media/sf_Laboratorio# cd Sockets_Multihilo/
root@Linux10:/media/sf_Laboratorio/Sockets_Multihilo#
```

```
root@Linux10:/media/sf_Laboratorio/Sockets_Multihilo# nano socket_client.c _
```

El ejemplo propuesto por la catedra, consiste en un programa cliente-servidor, donde el cliente creara hilos pthread y estos realizan solicitudes de conexión al servidor, para ello crea un socket de conexión y envía la solicitud al servidor, con los datos del cliente más el mensaje Hello.

Para que el programa se ejecute como cliente-servidor en máquinas distintas, es necesario realizar una modificación en la dirección IP del servidor, se debe reemplazar la 127.0.0.1 por la IP de la máquina que será servidor, en este caso el maestro 192.169.0.10.

```
  GNU nano 3.2                        socket_client.c                        Modificado

  int clientSocket;
  struct sockaddr_in serverAddr;
  socklen_t addr_size;
  // Create the socket.
  clientSocket = socket(PF_INET, SOCK_STREAM, 0);
  printf("Creo el socket\n");
  //Configure settings of the server address
 // Address family is Internet
  serverAddr.sin_family = AF_INET;
  //Set port number, using htons function
  serverAddr.sin_port = htons(8080);
 //Set IP address to localhost
  serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
  memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);
    //Connect the socket to the server using the address
    addr_size = sizeof serverAddr;
  connect(clientSocket, (struct sockaddr *) &serverAddr, addr_size);
    strcpy(message,"Hello");
   if( send(clientSocket , message , strlen(message) , 0) < 0)
    {
            printf("Send failed\n");
    }
    //Read the message from the server into the buffer
    if(recv(clientSocket, buffer, 1024, 0) < 0)
    {
       printf("Receive failed\n");
    }
    //Print the received message
    printf("Data received: %s\n",buffer);
    close(clientSocket);
    pthread_exit(NULL);
}

^G Ver ayuda  ^O Guardar    ^W Buscar      ^K Cortar txt  ^J Justificar  ^C Posición    M-U Deshacer
^X Salir      ^R Leer fich. ^\ Reemplazar  ^U Pegar txt   ^T Ortografía  ^_ Ir a línea  M-E Rehacer
```

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <fcntl.h> // for open
#include <unistd.h> // for close
#include<pthread.h>
void * cientThread(void *arg)
{
  printf("In thread\n");
  char message[1000];
  char buffer[1024];
  int clientSocket;
  struct sockaddr_in serverAddr;
  socklen_t addr_size;
  // Create the socket.
  clientSocket = socket(PF_INET, SOCK_STREAM, 0);
  printf("Creo el socket\n");
  //Configure settings of the server address
 // Address family is Internet
  serverAddr.sin_family = AF_INET;
  //Set port number, using htons function
  serverAddr.sin_port = htons(8080);
 //Set IP address to localhost
  serverAddr.sin_addr.s_addr = inet_addr("192.168.0.10");
  memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);
    //Connect the socket to the server using the address
    addr_size = sizeof serverAddr;
  connect(clientSocket, (struct sockaddr *) &serverAddr, addr_size);
    strcpy(message,"Hello");
```

En el servidor, se genera un socket para poder conectarse con el cliente, posteriormente crea hilos de ejecución que se encargaran de saludar al cliente.

Se debe compilar cada archivo, tanto el cliente como el servidor, para generar lo ejecutables.

```
root@Linux10:/media/sf_Laboratorio/Sockets_Multihilo# gcc socket_server.c -o servidor -lnsl -pthread
root@Linux10:/media/sf_Laboratorio/Sockets_Multihilo#
```

```
root@Linux10:/media/sf_Laboratorio/Sockets_Multihilo# gcc socket_client.c -o cliente -lnsl -pthread
root@Linux10:/media/sf_Laboratorio/Sockets_Multihilo# _
```

Finalmente, en la maquina maestro se debe ejecutar el servidor, y en cada esclavo se ejecutarán los clientes.

Maestro (servidor)

```
root@Linux10:/media/sf_Laboratorio/Sockets_Multihilo# ./servidor
Socket successfully binded..
Listening
Entra aca
_
```

Esclavo 1 (cliente)

```
root@Linux10Nodo1:~# cd /media/sf_Laboratorio/
root@Linux10Nodo1:/media/sf_Laboratorio# ls
c-chatroom-master  ChatRoom  chatroom-master  OpenMP  Pthreads  Sockets_Multihilo
root@Linux10Nodo1:/media/sf_Laboratorio# cd Sockets_Multihilo/
root@Linux10Nodo1:/media/sf_Laboratorio/Sockets_Multihilo# ./cliente_
```

```
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
Creo el socket
In thread
Creo el socket
In thread
Creo el socket
Data received: Hello Client
Data received: Hello Client
Data received: Hello Client
Data received: Hello Client
Data received: Hello Client
Data received: Hello Client
Data received: Hello Client
Data received: Hello Client
```

Se crea un socket por cada petición a conexión que reciba el maestro.

Chat-room con Pthread

Al igual que en el ejercicio anterior, se debe modificar en el código del cliente, la dirección IP 127.0.0.1 por la IP de la máquina que será servidor, en este caso es la IP del maestro, 192.168.0.10.

```
    str_trim_lf(nickname, LENGTH_NAME);
}
if (strlen(nickname) < 2 || strlen(nickname) >= LENGTH_NAME-1) {
    printf("\nName must be more than one and less than thirty characters.\n");
    exit(EXIT_FAILURE);
}

// Create socket
sockfd = socket(AF_INET , SOCK_STREAM , 0);
if (sockfd == -1) {
    printf("Fail to create a socket.");
    exit(EXIT_FAILURE);
}

// Socket information
struct sockaddr_in server_info, client_info;
int s_addrlen = sizeof(server_info);
int c_addrlen = sizeof(client_info);
memset(&server_info, 0, s_addrlen);
memset(&client_info, 0, c_addrlen);
server_info.sin_family = PF_INET;
server_info.sin_addr.s_addr = inet_addr("127.0.0.1");
server_info.sin_port = htons(8888);

// Connect to Server
int err = connect(sockfd, (struct sockaddr *)&server_info, s_addrlen);
if (err == -1) {
    printf("Connection to Server error!\n");
    exit(EXIT_FAILURE);
}

// Names

^G Ver ayuda  ^O Guardar     ^W Buscar      ^K Cortar txt  ^J Justificar  ^C Posición    M-U Deshacer
^X Salir      ^R Leer fich.  ^\ Reemplazar  ^U Pegar txt   ^T Ortografía  ^_ Ir a línea  M-E Rehacer
```

```
printf("Please enter your name: ");
if (fgets(nickname, LENGTH_NAME, stdin) != NULL) {
    str_trim_lf(nickname, LENGTH_NAME);
}
if (strlen(nickname) < 2 || strlen(nickname) >= LENGTH_NAME-1) {
    printf("\nName must be more than one and less than thirty characters.\n");
    exit(EXIT_FAILURE);
}

// Create socket
sockfd = socket(AF_INET , SOCK_STREAM , 0);
if (sockfd == -1) {
    printf("Fail to create a socket.");
    exit(EXIT_FAILURE);
}

// Socket information
struct sockaddr_in server_info, client_info;
int s_addrlen = sizeof(server_info);
int c_addrlen = sizeof(client_info);
memset(&server_info, 0, s_addrlen);
memset(&client_info, 0, c_addrlen);
server_info.sin_family = PF_INET;
server_info.sin_addr.s_addr = inet_addr("192.168.0.10");
server_info.sin_port = htons(8888);

// Connect to Server
int err = connect(sockfd, (struct sockaddr *)&server_info, s_addrlen);
if (err == -1) {
    printf("Connection to Server error!\n");
    exit(EXIT_FAILURE);
}
```

Para compilar el código se emplea el comando Make, que permite la generación automática de los códigos compilados.

```
root@Linux10:/media/sf_Laboratorio/chatroom-master# make
gcc -O3 -Wall -c src/server.c
In file included from src/server.c:12:
src/server.h: In function ‘newNode’:
src/server.h:17:5: warning: ‘strncpy’ specified bound 16 equals destination size [-Wstringop-truncat
ion]
     strncpy(np->ip, ip, 16);
     ^~~~~~~~~~~~~~~~~~~~~~~

In function ‘newNode’,
    inlined from ‘main’ at src/server.c:135:25:
src/server.h:17:5: warning: ‘strncpy’ specified bound 16 equals destination size [-Wstringop-truncat
ion]
     strncpy(np->ip, ip, 16);
     ^~~~~~~~~~~~~~~~~~~~~~~

gcc -O3 -Wall -pthread -o server.out server.o
gcc -O3 -Wall -c src/client.c
gcc -O3 -Wall -c src/string.c
gcc -O3 -Wall -pthread -o client.out client.o string.o
root@Linux10:/media/sf_Laboratorio/chatroom-master#
```

En el maestro, debe ejecutar el codigo del servidor y en cada esclavo se ejecutará el codigo del cliente.

Maestro

```
root@Linux10:/media/sf_Laboratorio/chatroom-master# ./server.out
Start Server on: 0.0.0.0:8888
_
```

Cliente (Nodo 0)

```
root@Linux10Nodo0:/media/sf_Laboratorio/chatroom-master# ./client.out
Please enter your name: sabrina
Connect to Server: 192.168.0.10:8888
You are: 192.168.0.11:48382
> _
```

Cliente (Nodo 1)

```
root@Linux10Nodo1:/media/sf_Laboratorio/chatroom-master# ./client.out
Please enter your name: Lourdes
Connect to Server: 192.168.0.10:8888
You are: 192.168.0.12:57722
>
```

Una vez ejecutado el código del cliente, se solicitará un nombre de sesión para el chat. El servidor, podrá ver no solo podrá ver los usuarios que se conecten, sino que también vera los mensajes que se envíen los clientes.

```
root@Linux10:/media/sf_Laboratorio/chatroom-master# ./server.out
Start Server on: 0.0.0.0:8888
Client 192.168.0.11:48382 come in.
sabrina(192.168.0.11)(4) join the chatroom.
Client 192.168.0.12:57722 come in.
Lourdes(192.168.0.12)(5) join the chatroom.
Send to sockfd 4: "Lourdes(192.168.0.12) join the chatroom."
```

Envío de mensajes entre los nodos.

```
root@Linux10Nodo1:/media/sf_Laboratorio/chatroom-master# ./client.out
Please enter your name: Lourdes
Connect to Server: 192.168.0.10:8888
You are: 192.168.0.12:57722
> hola, como estas?
>
```

```
root@Linux10Nodo0:/media/sf_Laboratorio/chatroom-master# ./client.out
Please enter your name: sabrina
Connect to Server: 192.168.0.10:8888
You are: 192.168.0.11:48382
Lourdes(192.168.0.12) join the chatroom.
Lourdes♦ hola, como estas? from 192.168.0.12
> Todo bien, y vos?
>
```

Servidor, imprimiendo por pantalla las conexiones y los mensajes que se enviaron.

```
root@Linux10:/media/sf_Laboratorio/chatroom-master# ./server.out
Start Server on: 0.0.0.0:8888
Client 192.168.0.11:48382 come in.
sabrina(192.168.0.11)(4) join the chatroom.
Client 192.168.0.12:57722 come in.
Lourdes(192.168.0.12)(5) join the chatroom.
Send to sockfd 4: "Lourdes(192.168.0.12) join the chatroom."
Send to sockfd 4: "Lourdes♦ hola, como estas? from 192.168.0.12"
Send to sockfd 5: "sabrina♦ Todo bien, y vos? from 192.168.0.11"
_
```

El servidor crea un socket de conexión TCP para los puertos 8888 de cada maquina que solicite la conexión, del lado del cliente además de solicitar una conexión al servidor, ejecuta las funciones de envío y recepción de mensajes, para ello se crean hilos con pthreads que se encargan de ejecutar estas funciones.

Chat-Room con OpenMp

Una vez realizadas las pruebas propuestas por la catedra, se modificó el código del chat-room para permitir la paralelización sincrónica. Para ello se comentó las líneas de código de pthreads, y se paralelizo las funciones de envío y recepción de mensajes del lado del cliente, además se estableció que se debían crear solo dos hilos de ejecución, uno para el envío y el otro para la respuesta.

Se establecieron secciones de paralelización, en donde de un hilo por vez podrá ejecutar el código.

```
root@Linux10:/media/sf_Laboratorio/c-chatroom-master/src# nano client.c_
```

```
        exit(EXIT_FAILURE);
    }
     // Names
    getsockname(sockfd, (struct sockaddr*) &client_info, (socklen_t*) &c_addrlen);
    getpeername(sockfd, (struct sockaddr*) &server_info, (socklen_t*) &s_addrlen);
    printf("Connect to Server: %s:%d\n", inet_ntoa(server_info.sin_addr), ntohs(server_info.sin_por$
    printf("You are: %s:%d\n", inet_ntoa(client_info.sin_addr), ntohs(client_info.sin_port));

    send(sockfd, nickname, LENGTH_NAME, 0);

int OMP_NUM_THREADS=2;
#pragma omp parallel  num_threads(OMP_NUM_THREADS)
{
#pragma omp sections
{
#pragma omp section
{
        send_msg_handler();
        exit(EXIT_FAILURE);
}
#pragma omp section
{
        recv_msg_handler();
        exit(EXIT_FAILURE);
}
}

}
/*#pragma omp parallel default(none) num_threads(2)
{
        recv_msg_handler();
        exit(EXIT_FAILURE);

^G Ver ayuda  ^O Guardar    ^W Buscar      ^K Cortar txt ^J Justificar ^C Posición   M-U Deshacer
^X Salir      ^R Leer fich. ^\ Reemplazar  ^U Pegar txt  ^T Ortografía ^_ Ir a línea M-E Rehacer
```

En el código del servidor, se paralelizo tanto la creación de los sockets de conexión como como el pasaje de los mensajes entre los destinatarios.

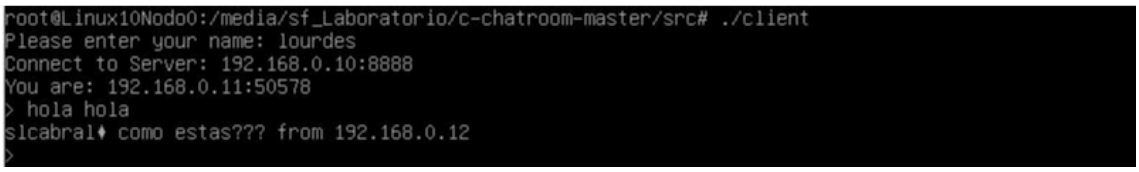
```
    // Initial linked list for clients
    root = newNode(server_sockfd, inet_ntoa(server_info.sin_addr));
    now = root;
    int OMP_NUM_THREADS=4;
    while (1) {
        #pragma omp parallel num_threads(OMP_NUM_THREADS)
        {

        client_sockfd = accept(server_sockfd, (struct sockaddr*) &client_info, (socklen_t*) &c_addr$

        // Print Client IP
        getpeername(client_sockfd, (struct sockaddr*) &client_info, (socklen_t*) &c_addrlen);
        printf("Client %s:%d come in.\n", inet_ntoa(client_info.sin_addr), ntohs(client_info.sin_po$

        // Append linked list for clients
        ClientList *c = newNode(client_sockfd, inet_ntoa(client_info.sin_addr));
        printf("la lista es ");
        c->prev = now;
        now->link = c;
        now = c;
        client_handler( (void *)c);
```

Para la ejecución del programa se puede utilizar Make, pero antes debe ser modificado el archivo makefile, para que este permita la compilación de OpenMp, caso contrario se puede realizar la compilación del archivo con gcc server.c -o server -fopenmp en la máquina del maestro.

```
root@Linux10:/media/sf_Laboratorio/c-chatroom-master/src# gcc server.c -o server -fopenmp
root@Linux10:/media/sf_Laboratorio/c-chatroom-master/src# ./server
Start Server on: 0.0.0.0:8888
```

Ejecución en el Nodo 1 (cliente)

Ejecución en el Nodo 0 (cliente)



Maestro

Conclusión

Luego de realizar cada una de las pruebas, se puede concluir que, si bien Pthreads permite un mejor manejo de los hilos, Openmp es más sencillo y escalable, ya que permite definir regiones completas que deberán ser ejecutadas en paralelo, permitiendo paralelismo dentro de paralelismo. Además, que no es necesario realizar modificaciones complejas en el código.