DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II - Threads

12 / 11 / 2014 Sistemas Operativos

Grupo número

Integrante	LU	Correo electrónico
Straminsky, Axel	769/11	axelstraminsky@gmail.com
Chapresto, Matias	201/12	matiaschapresto@gmail.com
Torres, Sebastian	723/06	sebatorres1987@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En el presente trabajo vamos a tratar de resolver el enunciado propuesto por la cátedra mediante el uso de *threads*.

1.1. Objetivo

El objetivo es implementar un servidor que acepte múltiples conexiones de clientes (usando sockets) y hacer que los mismos se comporten de determinada manera para poder emular la interacción entre alumnos, rescatistas y un aula que debe ser desalojada lo antes posible para practicar un simulacro de incendio, donde debemos tener en cuenta los movimientos de traslado de los estudiantes, ya que hay diversas condiciones.

1.2. Threads y Pthreads

Los threads son ampliamente utilizados para implementación de servidores **HTTP**, clientes de mensajería instantanea, UI, entre otras áreas. Por otro lado, **Pthreads** es una **especificación** y no una librería. Pthreads indica qué funciones debería proveer una librería de threads, sin importar el lenguaje de programación que se esté usando.

Una de las ventajas de usar threads es que todos comparten el mismo conjunto de datos que el thread padre que los crea. Además son muchísimo más eficientes que los subprocesos creados utilizando ***fork()*** en cuando a performance.

Pero a la vez su gran ventaja es su gran desventaja, porque al compartir el mismo conjunto de datos tenemos que tener cuidado cuando los mismos son accedidos o modificados. Comienzan a tomar reelevancia factores como el scheduling, race conditions, entre otros.

En este trabajo además de implementar el servidor multi con threads, se va a tener que tener especial cuidado a la sincrinización de los mismos para que la ejecución del servidor no tenga comportamientos inesperados.

2. Detalles de implementación

En esta sección vamos a dar detalle de nuestra implementación del servidor multicliente. Para sincronizar todos los threads se usan las variables presentes a continuación:

```
/* mutex para sincronizar la actualizacion del aula */
pthread_mutex_t mutex_actualizar_aula;

/* mutex para los rescatistas */
pthread_mutex_t mutex_colocar_mascara;
pthread_mutex_t mutex_esperar_rescatista;

/* variables de condicion para los rescatistas y el grupo de alumnos */
pthread_cond_t condicion_hay_rescatistas;
pthread_cond_t condicion_desalojar_grupo_alumnos;
int cant_personas_con_mascara;
bool estan_los_5;
```

../codigo/servidor_multi.c

A lo largo de la sección vamos a ir justificando la presencia de cada una de ellas.

2.1. Implementando el servidor servidor_multi.c

Para implementar el servidor con soporte para múltiples clientes nos basamos en el presente en **servidor_mono.c**, tal y como sugirió la cátedra. La idea principal es que al momento de conectarse el cliente, levantemos un thread con la rutina que ejecutaría el mismo en un servidor mono cliente. A continuación se muestra el código:

```
#include <signal.h>
#include <errno.h>

#include "biblioteca.h"

/* Estructura que almacena los datos de una reserva. */
typedef struct {
    int posiciones[ANCHO_AULA][ALTO_AULA];
    int cantidad_de_personas;
    int rescatistas_disponibles;
} t_aula;


/* Estructura de parametros para los threads */
typedef struct {
    int t_socket;
    t_aula* aula;
} thread_args;
```

../codigo/servidor_multi.c

Para poder pasarle parámetros a la rutina asociada a los threads, tuvimos que cambiar su interfáz, es decir, antes la misma era:

```
void* atendedor_de_alumno(int socket_fd, t_aula *el_aula)
```

../codigo/servidor_mono.c

y la cambiamos por:

```
void* atendedor_de_alumno(void* parameters)
```

../codigo/servidor_multi.c

Luego de ese cambio de interfáz el código de **atendedor_de_alumno** está preparado para ser llamado desde un thread. Como habíamos mencioado previamente, la idea sería crear un thread mientras me sigan llegando pedidos de conexión al socket del server.

La función principal presenta la siguiente implementación:

```
int main(void)
{
    int socketfd_cliente, socket_servidor, socket_size;
    struct sockaddr_in local, remoto;
```

```
    /* inicializamos los mutex y variables de condicion */
    pthread_mutex_init(&mutex_actualizar_aula, NULL);
    pthread_mutex_init(&mutex_colocar_mascara, NULL);
    pthread_mutex_init(&mutex_esperar_rescatista, NULL);

    /* Inicio las variables de condicion */
    pthread_cond_init(&condicion_hay_rescatistas, NULL);
    pthread_cond_init(&condicion_desalojar_grupo_alumnos, NULL);
    cant_personas_con_mascara = 0;
    estan_los_5 = false;

    /* Crear un socket de tipo INET con TCP (SOCK_STREAM). */
    if ((socket_servidor = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("creando socket");
    }

    /* Crear nombre, usamos INADDR_ANY para indicar que cualquiera puede conectarse aqui. */
    local.sin_family = AF_INET;
    local.sin_addr.s_addr = INADDR_ANY;
    local.sin_port = htons(PORT);

    if (bind(socket_servidor, (struct sockaddr *)&local, sizeof(local)) == -1) {
        perror("haciendo bind");
    }

    /* Escuchar en el socket y permitir 5 conexiones en espera. */
    if (listen(socket_servidor, 5) == -1) {
        perror("escuchando");
    }

    t_aula el_aula;
    t_aula_iniciar_vacia(&el_aula);

    /// Aceptar conexiones entrantes.
    socket_size = sizeof(remoto);
    for(;;) {
        if ((socketfd_cliente = accept(socket_servidor, (struct sockaddr*) &remoto,
                                       (socklen_t*) &socket_size)) == -1) {
            printf("!! Error al aceptar conexion\n");
        } else {
            // Creo un thread por cada cliente que quiere conectarse
            pthread_t thread;
            thread_args* args = malloc(sizeof(thread_args));
            args->t_socket = socketfd_cliente;
            args->aula = &el_aula;
            if (pthread_create(&thread, NULL, atendedor_de_alumno, (void*) args) != 0) {
                printf("Error al crear thread!!");
                return -1;
            }
        }
    }

    /* Explicar en el informe que pthread_join y estas funciones de limpieza no se terminan llamando
    , pero que en una
    implementacion "en serio" se deberian llamar */
    pthread_mutex_destroy(&mutex_actualizar_aula);
    pthread_mutex_destroy(&mutex_colocar_mascara);
    pthread_mutex_destroy(&mutex_esperar_rescatista);
    pthread_cond_destroy(&condicion_hay_rescatistas);
    pthread_cond_destroy(&condicion_desalojar_grupo_alumnos);

    pthread_exit(NULL);
```

../codigo/servidor_multi.c

Vemos como en el **for(;;)** se crean los threads, uno por cada conexión con un cliente. Acá es donde cobra valor la estructura definida previamente, *thread_args*, que la usamos para pasarle los parámetros necesario que la función **atendedor_de_alumno** necesita para funcionar.

2.2. Herramientas usadas/cambiadas para realizar pruebas

Para poder testear la implementación se hizo un script en *bash* para que corra una cierta cantidad de clientes en simultáneo. Además se hicieron unas ligeras modificaciones al *server_tester.py* provisto por la cátedra. El cambio consiste en hacer que cada vez que se ejecute *server_tester.py* se elija un nombre de país distinto. Luego, si ejecutamos 20 clientes en simultáneo vamos a tener una visión mas clara de quienes están ejecutando ya que no todos se llaman de la misma manera.

El script de *bash* es el siguiente:

```
#!/bin/bash

i=1
while [ $i -le $1 ]
do
    python server_tester.py &
    i=$(($i + 1))
done
```

../codigo/test_server.sh

El uso es sencillo, se debe ejecutar pasándole como parametro la cantidad de alumnos que queremos que ingresen al aula, o más técnicamente, la cantidad de clientes que queremos que se conecten al servidor.

Por otro lado, en *server_tester.py* se incluyó el modulo *random* al principio para poder obtener un país al azar de la n-tupla de países.

```
#! /usr/bin/env python

import socket
import sys
import random
import time
from paises import paises


HOST = 'localhost'
PORT = 5555
CLIENTES = 1
```

../codigo/server_tester.py

La obtención del nuevo país para el cliente se hace de la siguiente manera:

```
lugar_inicial = (random.randint(0, 9), random.randint(0,9))
clientes = [Cliente(paises[random.randint(0, len(paises)-1)], lugar_inicial)]
```

../codigo/server_tester.py

La idea de las últimas líneas era crear una posición inicial random para cada cliente creado. El motivo no es otro que poder tener diferentes casos de prueba, y no sólo que todos intenten ingresar a la misma posición. Notar que si bien esta implementación no garantiza que no se repitan nombres de países, al menos es mucho mejor que tener que lidiar con múltiples clientes conectados mostrando todos el mismo nombre. Con estas pequeñas modificaciones se nos facilitó mucho el testing del servidor.

3. Referencias