# IC6600: Principios de Sistemas Operativos

## Proyecto II - III Semestre 2017

### Gabriela Garro A - 2014107501

## Índice

# Introducción

## Compilación

```
gcc -pthread -o bftp bftp.c client.c server.c
```

## Uso del programa

```
./bftp
```

### Opciones:

- **bftp://<usuario>:<contraseña>@<dirección-ip>/<url-ruta>** : Conectar con una máquina remota
- **open <dirección-ip>** : establece una conexión remota
- **close** : cierra la conexión actual
- **pwd** : muestra el directorio activo remoto

# Componentes principales

En esta sección, se muestra código del código fuente y se explica su flujo y funcionamiento.

## Estructura general

El archivo `bftp.c` incluye a los archivos `server.h` y `client.h`.

`server.c` crea un thread que, a medida que recibe más solicitudes de cliente, crea threads para administrar a cada uno.

`client.c` contiene todas las funciones que se llaman desde `bftp.c` para comunicarse con el servidor.

## bftp.c

### main()

El main hace lo siguiente:

- Iniciar el servidor (solo si no se compiló como programa cliente)
- Pedir el login del usuario.
- Delegar a la función `manage_cmds()` el manejo del resto del input del usuario.

```
#include "client.h"
#include "server.h"

#define MAX_USERNAME_LENGTH 48
#define MAX_PASSWORD_LENGTH 32
#define MAX_IPADRESS_LENGTH 45
#define MAX_URL_LENGTH 512

// Variables globales
bool debug = true;
bool auto_login = false;
char username[MAX_USERNAME_LENGTH];
char password[MAX_PASSWORD_LENGTH];
char ip_address[MAX_IPADRESS_LENGTH];
char url[MAX_URL_LENGTH];

char input[MAX_PROMPT_LENGTH];
```

```
void print_help();
void print_error(char * error);
void manage_cmds();

int main(int argc, char * argv[]) {
	// Conexión inicial al principio del programa
	printf("Basic File Transfer Protocol\n");

	//Si no es bftp2 (cliente), no correr el servidor
	if (strcmp(argv[0], "./bftp2") != 0) server_init();

	printf("Ingrese la dirección de conexión remota:
bftp://<usuario>:<contraseña>@<dirección-ip>/<url-ruta>\n");
	printf("> ");
	if (auto_login) {
		printf("bftp://yo:micontraseña@192.168.2.7/una/ruta\n");
		strcpy(username, "yo");
		strcpy(password, "micontraseña");
		strcpy(ip_address, "192.168.2.6");
		strcpy(url, "una/ruta");
		client_init(username, password, ip_address, url);
	}
	else {
		scanf("bftp://%[^:]:%[^@]@%[^/]/%s", username, password,
ip_address, url);
		// Revisar que el formato logró almacenar los datos
		if (username[0] == 0 || password[0] == 0 || ip_address[0] == 0
|| url[0] == 0) {
			print_error("Formato de conexión incorrecto.");
		}
		else {
			client_init(username, password, ip_address, url);
		}
	}

	// Ciclo de otros comandos
	bool quit = false;
	while (!quit) {
```

```
            manage_cmds();
        }
        return 0;
}
```

### manage_cmds()

Esta función ejecuta la función del cliente que indica el input de usuario.

```
void manage_cmds() {
	//Obtener comando
	char arg1[MAX_PROMPT_LENGTH];
	char arg2[MAX_PROMPT_LENGTH];
	printf("> ");
	fgets(input, MAX_PROMPT_LENGTH - 1, stdin);

	strcpy(arg1, strtok(input, " \n"));

	// Identificar el comando
	// Quit
	if (strcmp(arg1, "quit") == 0) {
		exit(0);
	}
	// Close ***********************************
	else if (strcmp(arg1, "close") == 0) {
		client_close();
	}
	// Open ************************************
	else if (strcmp(arg1, "open") == 0) {
		//Recuperar siguiente argumento
		strcpy(arg2, strtok(NULL, " \n"));

		if (arg2[0] != 0) {
			strcpy(ip_address, arg2);
			client_open(arg2);
		}
		else {
			print_error("Falta argumento.");
		}
	}
	// pwd *************************************
	else if (strcmp(arg1, "pwd") == 0) {
		printf("Directorio actual: %s\n", client_pwd());
	}
```

```
	// cd ***********************************
	else if (strcmp(arg1, "cd") == 0) {
		//Recuperar siguiente argumento
		strcpy(arg2, strtok(NULL, " \n"));
		client_cd(arg2);
	}
	// Help *********************************
	else if (strcmp(arg1, "help") == 0) { // help
		print_help();
	}
	// Default ******************************
	else {
		printf("Comando incorrecto. Use help para ver los comandos
disponibles.\n");
	}
}
```

## server.h

Al utilizar este archivo, primero se debe llamar a server_init(), el cual inicializa el thread del servidor y comienza a esperar a clientes. Cuando un cliente se conecte, se crea un thread hijo para administrar la conexión.

```
bool debug_server;
const char * port;
pthread_t server_thread;
// Para pasar parámetros al thread del servidor
struct workerArgs {
    int socket;
};
char url[MAX_URL_LENGTH];
char server_buffer [MAX_PROMPT_LENGTH + 1 ]; // +1 for '\0'
int nbytes; // Num de bytes recibidos

int server_init();
void * server_main_thread(void *args);
void * server_client_thread(void *args);
char * server_pwd();
```

```
int server_change_dir(char * dir);
int server_cd();
int server_recv(int socket);
int server_send(int socket, char * msg);
```

### server_init()

```
/* Inicializa el thread del servidor */
int server_init() {
    debug_server = true;
    port = "8889";

     //pthread_t server_thread;

     // Ignorar la señal SIGPIPE de cuando un cliente se desconecta
     // para que no mate el proceso
     sigset_t new;
    sigemptyset (&new);
    sigaddset(&new, SIGPIPE);
    if (pthread_sigmask(SIG_BLOCK, &new, NULL) != 0) {
        perror("\n** ERROR: Unable to mask SIGPIPE");
        exit(-1);
    }

     // Crear el thread del servidor que a su vez hace más threads para
atender clientes
     if (pthread_create(&server_thread, NULL, server_main_thread, NULL) <
0) {
        perror("\n** ERROR: Servidor: No se puede crear thread del
servidor");
        exit(-1);
    }

    //pthread_join(server_thread, NULL);
    //pthread_exit(NULL);
    return 0;
}
```

### server_main_thread()

```
/* Función del thread del servidor que hace más threads para atender a
clientes */
void * server_main_thread(void *args){
    int serverSocket;
    int clientSocket;
    pthread_t worker_thread;
    struct addrinfo hints, *res, *p;
    struct sockaddr_storage *clientAddr;
    socklen_t sinSize = sizeof(struct sockaddr_storage);
    struct workerArgs *wa;
    int yes = 1;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
     // Retorna la dirección para que se le pueda hacer bind
    hints.ai_flags = AI_PASSIVE;

    if (getaddrinfo(NULL, port, &hints, &res) != 0) { // El parámetro del
host es NULL
        perror("\n** ERROR: Servidor: getaddrinfo() falló");
        pthread_exit(NULL);
    }

     // Crear los sockets
    for (p = res; p != NULL; p = p->ai_next) {
        if ((serverSocket = socket(p->ai_family, p->ai_socktype,
p->ai_protocol)) == -1) {
            perror("\n** ERROR: Servidor: No se pudo abrir el socket");
            continue;
        }
        if (setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR, &yes,
sizeof(int)) == -1) {
            perror("\n** ERROR: Servidor: Socket setsockopt() falló");
            close(serverSocket);
            continue;
        }
```

```
        if (bind(serverSocket, p->ai_addr, p->ai_addrlen) == -1) {
            perror("\n** ERROR: Servidor: Socket bind() falló");
            close(serverSocket);
            continue;
        }
        if (listen(serverSocket, 5) == -1) {
            perror("\n** ERROR: Servidor: Socket listen() falló");
            close(serverSocket);
            continue;
        }
        break;
    }

    freeaddrinfo(res);

    if (p == NULL) {
        fprintf(stderr, "\n** ERROR: Servidor: No se encontró un socket
al cual hacer bind.\n> ");
        pthread_exit(NULL);
    }

    // Esperar clientes
    while (1) {
        //if (debug_server) printf("Servidor: Esperando
conexiones...\n>");
           // El thread se bloquea hasta que se hace una conexión
        clientAddr = malloc(sinSize);
        if ((clientSocket = accept(serverSocket, (struct sockaddr *)
clientAddr, &sinSize)) == -1) {
            free(clientAddr); // Si la conexión falla, no se muere el
thread del servidor :)
            perror("\n** ERROR: Servidor: No se pudo aceptar conexión");
            continue;
        }

           // Se conectó con un cliente
           // Hay que pasarle al thread el struct workerArgs, que tiene
el socket para conectarse
```

```
        wa = malloc(sizeof(struct workerArgs));
        wa->socket = clientSocket;

           // Crear un thread para esta conexión
        if (pthread_create(&worker_thread, NULL, server_client_thread,
wa) != 0) {
            perror("\n** ERROR: Servidor: No se pudo crear un thread para
la conexión");
                // Liberar todo
            free(clientAddr);
            free(wa);
            close(clientSocket);
            close(serverSocket);
            pthread_exit(NULL);
        }
    }

    pthread_exit(NULL);
}
```

### server_client_thread()

```
/* Thread que atiende a cada cliente */
void * server_client_thread (void *args) {
    struct workerArgs *wa;
    int socket, nbytes;
    char tosend[100]; // Mensaje a enviar
     // Traer la conexión del socket
    wa = (struct workerArgs*) args;
    socket = wa->socket;

     // Este thread es independiente y no se tiene que comunicar con otro
cliente
    pthread_detach(pthread_self());

    if (debug_server) fprintf(stderr, "\n* Servidor: Socket %d
conectado\n> ", socket);
```

```
    // Loop que espera mensajes de los clientes
    while (1) {
        int flag = server_recv(socket);
        if (flag == -1) { // Si falló
            free(wa);
            close(socket);
            pthread_exit(NULL);
        }
        server_buffer[nbytes] = '\0'; //Por si acaso
        printf("\n* Socket %d: %s \n", socket, server_buffer);

        //Enviar mensaje con lo que pide
        //char * pwd = server_pwd();
        if (strcmp(server_buffer, "pwd") == 0) {
            char * pwd = "dir";
            flag = server_send(socket, pwd);
            if (flag == -1) {
                free(wa);
                pthread_exit(NULL);
            }
        }
        else {
            printf("No implementado\n");
            server_send(socket, "No implementado");
        }

    }
    pthread_exit(NULL);
}
```

## client.h

```
char username[MAX_USERNAME_LENGTH];
char password[MAX_PASSWORD_LENGTH];
char ip_address[MAX_IPADRESS_LENGTH];
char url[MAX_URL_LENGTH];

bool debug_client;
```

```
int sock; //Descriptor del socket
struct sockaddr_in server; // Struct con la información del servidor
/*  struct addrinfo {
            int ai_flags;
            int ai_family;
            int ai_socktype;
            int ai_protocol;
            size_t ai_addrlen;
            struct sockaddr *ai_addr;
            char *ai_canonname;
            struct addrinfo *ai_next;
     };
*/
char client_buffer[MAX_BUFFER_LENGTH + 1]; // +1 for '\0'
int nbytes; // Num de bytes recibidos

int client_init(char _username[], char _password[], char _ip_address[],
char _url[]);
int client_close();
int client_open(char _ip_address[]);
char * client_pwd();
int client_cd(char * dir);
```