



Universidad Tecnológica Nacional – Facultad Regional Buenos Aires  
Ingeniería en Sistemas de Información  
Sistemas Operativos (082027)



Trabajo Práctico 0 - Resolución

# Sistemas Operativos

UTN-FRBA

Versión 1.0

# Introducción

Para resolver el TP0 lo primero que se necesita es crear y poder compilar un “Hola Mundo” en C para comprobar que todo esté funcionando. Para eso, hay que crear un archivo `tp0.c` con:

```
#include <stdio.h>

int main() {
    printf("Hola mundo");
}
```

y ejecutar desde la terminal “`gcc tp0.c -o tp0`”. No deberíamos ver ningún error. Si lo ejecutamos con “`./tp0`” deberíamos ver el mensaje impreso.

## Funciones auxiliares:

Antes de empezar vamos a crear algunas funciones auxiliares que nos van a ser útiles.

- Configurar el logger:

```
void configure_logger() {
    logger = log_create("tp0.log", "TP0", true, LOG_LEVEL_INFO);
}
```

Es importante no olvidarse de agregar `#include <commons/log.h>` al principio. Y llamar a esta misma función desde el `main()`

- Finalizar el proceso:

```
void exit_gracefully(int return_nr) {
    log_destroy(logger);
    exit(return_nr);
}
```

- Finalizar el proceso mostrando un mensaje de error y liberando los recursos

```
void _exit_with_error(int socket, char* error_msg, void * buffer) {
    if (buffer != NULL) {
        free(buffer);
    }
    log_error(logger, error_msg);
}
```

```
close(socket);
exit_gracefully(1);
}
```

*Nota: para poder compilar el proyecto, ahora vamos a necesitar agregar el flag "-lcommons" a gcc. Ej: "gcc tp0.c -lcommons -o tp0". En caso que falle, probablemente sea porque la biblioteca no está instalada correctamente.*

## Conectándonos al servidor:

Una vez que tengamos listo el programa en C que compila sin problemas es hora de conectarnos al servidor. Para eso es necesario crear un socket y especificarle los datos.

- Host: **tp0.utnso.com**
- Puerto: **8080**

Primero que nada necesitamos crear una función que reciba el host y el puerto y reciba un socket (en C los sockets se representan con un int):

```
#include <sys/socket.h>
#include <netdb.h>

int connect_to_server(char * ip, char * port) {
    ...
}
```

Dentro de la función conectar tenemos que hacer tres cosas:

- Crear el socket y especificar a dónde nos queremos conectar

```
struct addrinfo hints;
struct addrinfo *server_info;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
getaddrinfo(ip, port, &hints, &server_info);

int server_socket = socket(server_info->ai_family,
server_info->ai_socktype, server_info->ai_protocol);
```

- Conectarnos!

```
int res = connect(server_socket, server_info->ai_addr,
server_info->ai_addrlen);
```

- Chequear que la conexión fue un éxito

```
freeaddrinfo(server_info);
if (res < 0) {
    _exit_with_error(server_socket, "No me pude conectar al servidor", NULL);
}

log_info(logger, "Conectado!");
return server_socket;
```

## Recibiendo el primer mensaje

Según el enunciado una vez que nos conectamos, el proceso debe recibir un mensaje de 16 bytes. Para eso, vamos a crear la función:

```
void wait_hello(int socket) {
    ...
}
```

Para eso lo primero que necesitamos un buffer de 16 bytes + 1 del "\0":

```
char * hola = "SYSTEM UTNSO 0.1";
char * buffer = (char*) calloc(sizeof(char), strlen(hola) + 1);
```

Después simplemente esperamos a que el mensaje llegue:

```
int result_recv = recv(socket, buffer, strlen(hola), MSG_WAITALL);
```

Y en caso que haya llegado de forma correcta lo imprimimos en pantalla y liberamos el buffer:

```
if(result_recv <= 0) {
    _exit_with_error(socket, "No se pudo recibir hola", buffer);
}
```

```

}

if (strcmp(buffer, hola) != 0) {
    _exit_with_error(socket, "No se pudo recibir hola", buffer);
}

log_info(logger, "Mensaje de hola recibido: '%s'", buffer);
free(buffer);

```

## Solicitando la información del alumno:

Lo próximo que el programa debe hacer, es solicitar nuestra información por pantalla. Lo resolverá la función:

```

Alumno read_hello() {
    ...
}

```

Primero necesitamos crear la estructura que guardará la información:

```

Alumno alumno = { .nombre = "", .apellido = "" };

```

Y con readline pedir los datos:

```

char * legajo = readline("Legajo: ");
alumno.legajo = atoi(legajo);
free(legajo);

char * nombre = readline("Nombre: ");
memcpy(alumno.nombre, nombre, strlen(nombre));
free(nombre);
char * apellido = readline("Apellido: ");
memcpy(alumno.apellido, apellido, strlen(apellido));
free(apellido);

```

Para que esto funcione no debemos olvidarnos agregar `-lreadline` a los parámetros de compilación de gcc y `#include <readline/readline.h>`

## Enviando el primer mensaje

Una vez recibido el mensaje ya podemos enviar la primer estructura que muestra el enunciado. Para eso creamos una función:

```
void send_hello(int socket, Alumno alumno) {  
    ...  
}
```

Y la enviamos por el socket:

```
alumno.id_mensaje = 99;  
if (send(socket, &alumno, sizeof(Alumno), 0) <= 0) {  
    _exit_with_error(socket, "No se enviar el hola", NULL);  
}
```

Si te estas preguntando porqué funciona el send y no es necesario “*serializar*” nada de la estructura es porque usa el atributo `__attribute__((packed))`. Esto solamente funciona siempre y cuando la estructura no tenga punteros y los datos se envíen a procesadores con arquitecturas compatibles. En caso que alguna de las condiciones no se cumplan, es necesario crear un buffer y serializar los datos antes de poder enviarlos.

## Recibiendo el contenido

Crearemos la función que va a devolver el contenido a calcular el MD5:

```
void * wait_content(int socket) {  
    ...  
}
```

Lo primero que debemos hacer es crear el buffer y esperar el header:

```
log_info(logger, "Esperando el encabezado del contenido(%ld bytes)",  
sizeof(ContentHeader));  
ContentHeader * header = (ContentHeader*) malloc(sizeof(ContentHeader));  
  
if (recv(socket, header, sizeof(ContentHeader), 0) <= 0) {  
    _exit_with_error(socket, "No se pudo recibir el encabezado del  
contenido", header);  
}
```

```
}
```

Chequear que el id recibido sea correcto

```
if (header->id != 18) {  
    _exit_with_error(socket, "Id incorrecto, deberia ser 18", header);  
}
```

Recibir los datos:

```
log_info(logger, "Esperando el contenido (%d bytes)", header->len);  
  
void * buf = calloc(sizeof(char), header->len + 1);  
  
if (recv(socket, buf, header->len, MSG_WAITALL) <= 0) {  
    free(buf);  
    _exit_with_error(socket, "Error recibiendo el contenido", header);  
}  
log_info(logger, "Contenido recibido '%s'", (char*) buf);  
free(header);  
return buf;
```

Una buena práctica sería chequear primero que el tamaño a reservar no sea demasiado grande. En caso que haya algún problema un mensaje que se recibió de forma inválida podría hacer que se reserve mucha memoria y haga fallar el programa.

## Calculando el MD5

Cuando tenemos el contenido tenemos que calcular su MD5. Existen varias formas de hacerlo. Una forma bastante básica es guardar los datos en un archivo y ejecutar md5sum. Pero existe una forma más fácil que es usando openssl, una biblioteca que nos brinda varias funciones. Entre algunas esta md5. Entonces, creamos una función:

```
void send_md5(int socket, void * content) {  
    ...  
}
```

Lo primero que tenemos que hacer es reservar un espacio para el md5 calculado

```
void * digest = malloc(MD5_DIGEST_LENGTH);
```

Creamos las estructuras que openssl usa y le enviamos los datos para que realice el cálculo.

```
MD5_CTX context;  
MD5_Init(&context);  
MD5_Update(&context, content, strlen(content) + 1);  
MD5_Final(digest, &context);
```

*Nota: para que esto funcione, debemos agregar el include "#include <openssl/md5.h>" y el parametro "-lcrypto" a gcc*

Una vez hecho, creamos las estructuras a enviar:

```
ContentHeader header = { .id = 33, .len = MD5_DIGEST_LENGTH };  
int message_size = sizeof(ContentHeader) + MD5_DIGEST_LENGTH;  
void * buf = malloc(message_size);  
memcpy(buf, &header, sizeof(ContentHeader));  
memcpy(buf + sizeof(ContentHeader), digest, MD5_DIGEST_LENGTH);
```

Y finalmente enviamos el mensaje

```
log_info(logger, "Enviando MD5");  
int result_send = send(socket, buf, message_size, 0);  
free(buf);  
  
if (result_send <= 0) {  
    _exit_with_error(socket, "No se pudo enviar el md5", NULL);  
}
```

Nota: Como sabemos que el tamaño del MD5 siempre es 32, podríamos haberlo agregado a la estructura como un char[32].

## Recibiendo la confirmación

Finalmente, lo último que queda es recibir la confirmación del servidor de que el MD5 que enviamos es correcto. Para eso creamos la función:

```
void wait_confirmation(int socket) {
```



```
...  
}
```

Recibimos el resultado:

```
int result = 0;  
log_info(logger, "Esperando confirmacion");  
  
if (recv(socket, &result, sizeof(int), 0) <= 0) {  
    _exit_with_error(socket, "No se pudo recibir confirmacion", NULL);  
}
```

Y comprobamos que sea 1:

```
if (result != 1) {  
    _exit_with_error(socket, "El md5 no coincidio", NULL);  
}  
  
log_info(logger, "Los MD5 concidieron!");
```

En caso que el resultado que llegue sea 1 significa que el programa terminó de forma correcta. Ahora podemos finalizar:

```
close(socket);  
exit_gracefully(0);
```