Programación de aplicaciones — Servidor de documentación

Sistemas Operativos 2024/2025

Jesús Torres

Vamos a desarrollar una herramienta para compartir documentos en Internet. Pero antes de ponernos con lo más complicado, vamos a empezar creando un programa que muestra el contenido de un archivo por la salida estándar. Después no familiarizaremos con la API de *sockets* y haremos que los archivos están disponibles en la red.

Contenidos

Consejos y recomendaciones

Las recomendaciones y consejos generales sobre programación en C++ que pueden ayudar en el desarrollo de la práctica están disponibles en un documento diferente a este. Te recomendamos que los tengas presentes mientras lees y resuelves las distintas partes de este guion.

Haremos referencia a estos consejos múltiples veces, por lo que te será fácil recordarlos y aplicarlos. Estas referencias se indican con: [1]

1. Primeros pasos

1.1. Objetivo

Vamos a comenzar creando un programa llamado `docserver` que acepta los siguientes argumentos de línea de comandos:

```
docserver [-v | --verbose] [-h | --help] ARCHIVO
```

Cuando se ejecuta sin opciones, el programa muestra el contenido del archivo `ARCHIVO` en la **salida estándar**, utilizando el siguiente formato:

```
Content-Length: <TAMAÑO>                                    ①

<contenido del archivo...>                                  ②
```

① `<TAMAÑO>` se sustituye por el tamaño de `ARCHIVO`.
② `<contenido del archivo...>` se sustituye por el contenido de `ARCHIVO`.

1.1.1. Opciones de línea de comandos

Si se indica la opción:

- `-h` o `--help`, el programa solo muestra un mensaje de ayuda sobre el uso del programa y termina con éxito.
- `-v` o `--verbose`, activará el modo detallado. En este modo el que el programa funciona con normalidad, pero, adicionalmente, muestra mensajes informativos por la **salida de error** sobre las funciones de la librería del sistema que va utilizando. Por ejemplo:

  ```
  open: se abre el archivo "ARCHIVO"
  read: se leen 1234 bytes del archivo "ARCHIVO"
  ```

  Esta opción no afecta al formato de salida del programa en la **salida estándar** y va a resultarnos muy útil para depurar el programa.

1.1.2. Manejo de errores

En todas las situaciones de error, el programa debe mostrar un mensaje de error por la **salida de error**. En el caso de los errores debidos a funciones de la librería del sistema, el mensaje debe incluir el código de error `errno` y el texto descriptivo obtenido mediante `std::strerror()`.

Las funciones que vamos a desarrollar deben ser reutilizables y, por tanto, deben retornar el código de error a quien las llamó, en lugar de mostrarlo ellas. Los mensajes de error deben ser mandados a imprimir en `main()` o lo más cerca posible de `main()`.

> **!** Importante
>
> Para entender cómo se deben propagar los errores desde las funciones hacia `main()`, lee atentamente la Sección 5.1 “Propagación de códigos de error” de [1].

Algunos de los errores son **errores fatales** y deben provocar la terminación del programa con un **código de salida distinto de 0**. Sin embargo, el programa debe terminar retornando por `main()` –tal y como se recomienda en C++– nunca usando `std::exit()`, `exit()` ni funciones similares.

Son **errores fatales** del programa:

- Si no se indica un nombre de archivo en los argumentos.
- Si se indica una opción desconocida o es conocida pero no se ha indicado un argumento.
- Si cualquiera de las funciones de la librería del sistema que necesitamos para leer el archivo falla, excepto si se indique otra cosa.

Por el contrario, los **errores leves** hacen que el programa termine con éxito y, aparte de mostrar un mensaje de error por la **salida de error**, deben mostrar un mensaje por la **salida estándar**.

Los siguientes son **errores leves** en esta primera parte:

- Si el archivo no puedo abrir porque no se tienen permisos para leerlo. En este caso la salida del programa debe ser:

```
403 Forbidden
```

- Si el archivo no puedo abrir porque no existe. En este caso la salida del programa debe ser:

```
404 Not Found
```

1.2. Implementación

El flujo de ejecución del programa es muy sencillo:

> `parse_args()`: Procesar los argumentos de la línea de comandos
> **error-fatal if** error al procesar los argumentos
>
> **if** las opciones contienen `-h` o `--help`
> Mostrar ayuda
> Terminar sin error
> **end if**
>
> **error-fatal if** *NO* se ha indicado un nombre de archivo en los argumentos
>
> `read_all()`: Leer archivo en memoria
> **if** error de permisos o archivo no encontrado
> `send_response()`: Mostrar mensaje de error leve
> (ver Sección 1.1.2)
> **error**

elif otro error etectado
  error-fatal
else
  `send_response()`: Mostrar el contenido del archivo en la salida estándar (ver Sección 1.1)
end if

En este pseudocódigo se indican las funciones que se deben implementar y cuando se deben llamar. Cuando se indica **error-fatal**, significa que el programa mostrará un mensaje de error por la **salida de error** y terminar con un código de salida distinto de 0. Cuando se indica **error**, significa que el programa mostrará un mensaje de error por la **salida de error** y terminar con el código de salida 0.

Vamos a ver cada uno de estos pasos y las funciones indicadas con más detalle.

1.2.1. Procesar los argumentos de la línea de comandos

Para procesar los argumentos de la línea de comandos puedes desarrollar tu propia de la función `parse_args()`, tal y como se describe en la Sección 6 "Argumentos de la línea de comandos" de [1].

1.2.2. Leer el contenido del archivo

Vamos a implementar la siguiente función para leer todo el contenido del archivo indicado en la ruta `path`:

```
std::expected<std::string_view, int> read_all(const std::string& path);
```

Esta función debe desarrollarse de forma similar a las funciones `read_file()` y `open_file()` implementadas en las diferentes secciones de [1], pero con las siguientes diferencias:

- En lugar de leer el archivo `path` con `read()`, `read_all()` utilizará `mmap()` para mapear el archivo en memoria y devolver dicha región mapeada.

  El tamaño de la región mapeada y la dirección inicial de dicha región en memoria se devuelve mediante un objeto `std::string_view`, de forma que podremos trabajar con esa memoria en el resto del código como si fuera una cadena.

- Como se ilustra con la función `open_file()` en la Sección 5.3 "Propagación de errores con `std::expected`" en [1], `read_all()` debe devolver un `int` con el valor de `errno` en caso de error.

> **!** Importante
>
> Para aprender más sobre cómo usar `mmap()` y mapear archivos en memoria, consulta la documentación de los ejemplos del capítulo 17.

1.2.3. Escribir en la salida estándar

Para mostrar el contenido del archivo o los mensajes de los **errores leves** por la salida estándar, se puede utilizar `std::cout`. Sin embargo, para estar preparados para el desarrollo de la siguiente parte de la práctica, te recomendamos que utilices la siguiente función que tendrás que implementar:

```
void send_response(std::string_view header, std::string_view body = {});
```

Esta función debe mostrar por `std::cout` el contenido de `header` seguido de una línea en blanco y, a continuación, el contenido de `body`, siempre que no esté vacío. Por ejemplo, si `header` vale `Content-Length: 10`, y `body` vale `Hola mundo`, la salida del programa sería:

```
Content-Length: 10
                                                                    ①
Hola mundo
```

① Línea en blanco entre `header` y `body` añadida por `send_response()`.

1.2.3.1. Sobre `header`

En base a lo que hemos visto hasta ahora, la cadena en `header` puede valer `Content-Length: <TAMAÑO>`, `404 Not Found` o `403 Forbidden`, según corresponda, y debe terminar con un salto de línea, para que el formato de salida sea correcto.

Para construir la cadena de `header` cuando contiene valores como el tamaño del archivo, puedes utilizar `std::ostringstream` y el operador `<<` o `std::format()`.

Por ejemplo, `header` se puede construir de la siguiente manera:

```
std::ostringstream oss;
oss << "Content-Length: " << size << '\n';
std::string header = oss.str();
```

o, alternativamente, se puede utilizar `std::format()` de la siguiente forma:

```
std::string header = std::format("Content-Length: {}\n", size);
```

En ambos ejemplos, luego se puede pasar `header` a `send_response()` para mostrarlo por la **salida estándar**.

1.2.3.2. Sobre `body`

La cadena en `body` puede tener el contenido del archivo, si el archivo pudo leerse, o estar vacía, en el caso de un error no grave. Por ejemplo, si el archivo no se pudo abrir porque no se tienen permisos para leerlo, `header` valdría `Forbidden` y `body` estaría vacío, por lo que la salida del programa sería:

```
403 Forbidden
                                                                    ①
```

① Línea en blanco entre `header` y `body` añadida por `send_response()`.

1.2.4. Desmapear el archivo de la memoria

El `std::string_view` devuelto por `read_all()` apunta a la región de la memoria que mapea el archivo. Antes de finalizar el programa esta región debe ser desmapeada. Por tanto, al menos debes llamar a `munmap()` para liberar esta región mapeada antes de cada punto de salida del programa.

`munmap()` necesita la dirección de la región mapeada y su tamaño. Ambos valores se pueden obtener del `std::string_view` devuelto por `read_all()`.

> **!** Importante
>
> Para aprender más sobre cómo usar `munmap()`, consulta la documentación de los ejemplos del capítulo 17.

Sin embargo, en C++ se recomienda que la liberación de recursos ocurra automáticamente cuando una variable sale del ámbito y se destruya, por lo que se valorará positivamente hacerlo de la manera que comentaremos en el siguiente apartado.

1.2.4.1. SafeMap

Para desmapear la región mapeada automáticamente, puedes crear una clase `SafeMap` similar a la clase `SafeFD` descrita en la Sección 4 de [1]; pero que en lugar de gestionar un descriptor de archivo que gestione un `std::string_view` que apunte a la región de memoria mapeada.

Las diferencias entre ambas clases son:

- `SafeMap` no debe guardar un descriptor de archivo `int` en un atributo privado, sino un `std::string_view`.

  ```
  private:
      std::string_view sv_;
  ```

  El constructor de `SafeMap` y el resto de los métodos de la clase deben ser modificados teniendo esto en cuenta.

- `SafeMap` debe tener un destructor que llame a `munmap()` con la dirección y el tamaño de la región a la que apunta el `std::string_view`, para desmapear la región cuando sea destruido el objeto.

Con la clase `SafeMap` implementada, la definición de `read_all()` debe cambiar, porque ahora debe devolver un `SafeMap` en lugar de un `std::string_view` directamente.

```
std::expected<SafeMap, int> read_all(const std::string& path);
```

Para llamar a `send_response()` pasando el contenido del archivo, ahora se debe usar el método `get()` del objeto `SafeMap` devuelto por `read_all()`, para obtener el `std::string_view` que apunta a la región mapeada.

Con todo eso hecho, ya no es necesario llamar a `munmap()` explícitamente. Cuando la variable que contiene el `SafeMap` salga del ámbito, el destructor de `SafeMap` se encarga de llamar a `munmap()` automáticamente.

2. Servidor de documentos

2.1. Objetivo

En esta segunda parte el programa `docserver` debe aceptar los siguientes argumentos de línea de comandos:

```
docserver [-v | --verbose] [-h | --help]
          [-p <puerto> | --port <puerto>] ARCHIVO
```

Si se ejecuta sin opciones, el programa se queda escuchando en un puerto TCP utilizando *sockets*. Cuando recibe una conexión, el programa responde a través de la conexión, exactamente igual que en la práctica anterior. Una vez que ha respondido a la conexión, **el programa se queda escuchando en el puerto TCP para recibir más conexiones** y volver a responder, indefinidamente.

> **!** Importante
>
> La principal diferencia de esta versión con respecto al de la Sección 1 es que ahora el programa no muestra su respuesta por la **salida estándar**, sino que la envía a través del *socket* al otro proceso conectado.

2.1.1. Opciones de línea de comandos

Las opciones `-h` y `-v` operarán igual que en la versión anterior.

> **!** Importante
>
> Ten encuenta que ahora el programa usará funciones adicionales de la librería del sistema, como `socket()`, `bind()`, `listen()`, `accept()`, `send()` o `recv()`. Por lo que es importante que implementes el modo detallado `-v` para esos nuevos casos.
> Recuerda que la salida del modo detallado debe ser por la **salida de error**, no por el *socket*.

La nueva opción `-p` o `--port` permite indicar el puerto en el que el programa escuchará las conexiones:

- Si no se indica la opción `-p`, el programa escuchará en el puerto indicado en la variable de entorno `DOCSERVER_PORT`.
- Si esta variable de entorno no está definida, el programa escuchará en el puerto `8080` por defecto.

2.1.2. Manejo de errores

No hay muchos cambios en el manejo de errores con respecto a la versión anterior.

Igual que en la versión anterior, todos los errores deben tratarse enviando un mensaje de error por la **salida de error**. Además, en caso de **error fatal**, el programa debe terminar con un código de salida distinto de 0.

Por el contrario, ahora los **errores leves** ya no deben causar la terminación del programa, sino que deben provocar que el programa cierre la conexión actual y se ponga a la espera de una nueva conexión. En este caso, el mensaje de error que antes se mostraba por la **salida estándar** ahora debe enviarse a través del *socket* al otro extremo de la conexión.

Un nuevo tipo de **error leve** en esta versión ocurrirá si `send()` falla al enviar la respuesta a través del *socket* con el error `ECONNRESET`. Este error indica que el otro extremo de la conexión ha cerrado la conexión antes de recibir la respuesta. Por tanto, se mostrará un mensaje por la **salida de error** y se cerrará la conexión, pero no se terminará el programa.

2.1.3. Comprobación

Para comprobar que la transferencia funciona, podemos usar el comando `socat`. Este comando nos permite conectar al puerto 8080 –o al que corresponda– en la máquina local, donde escucha `docserver`, y ver que la respuesta que nos da es la esperada. Por ejemplo:

```
$ socat STDIO TCP:127.0.0.1:8080
Content-Length: 10

Hola mundo
$                                                                    (1)
```

① Recuerda que `docserver` debe cerrar la conexión tras enviar la respuesta, por lo que `socat` terminará y devolverá el control a la *shell*.

2.2. Implementación

El flujo de ejecución del programa es similar, pero ahora debe crear un *socket* y quedarse a la escucha en un bucle infinito para recibir conexiones y responder a través de ellas.

`parse_args()`: Procesar los argumentos de la línea de comandos
error-fatal if error al procesar los argumentos

if las opciones contienen `-h` o `--help`
  Mostrar ayuda
  Terminar sin error
end if

error-fatal if *NO* se ha indicado un nombre de archivo en los argumentos

`make_socket()`: Crear un *socket* y asignarle el puerto indicado
error-fatal if error al crear el *socket*

`listen_connection()`: Poner el *socket* a la escucha
error-fatal if error al poner el *socket* a la escucha

loop
  `accept_connection()`: Aceptar una conexión
  error-fatal if error al aceptar la conexión

  `read_all()`: Leer archivo en memoria
  if error de permisos o archivo no encontrado
    `send_response()`: Responder con el mensaje de error leve
    error
  elif otro error detectado
    error-fatal
  else
    `send_response()`: Responder con el contenido del archivo
  end if
  if error `ECONNRESET` en `send_response()`
    error
  else if otro error detectado
    error-falta
  end if

  Cerrar la conexión (`SafeFD se encarga`)
end loop

Ahora, cuando se indica **error** se hace referencia a que el programa debe mostrar un mensaje de error por la **salida de error**, cerrar la conexión y continuar esperando nuevas conexiones en el bucle principal.

Vamos a ver cada uno de los nuevos pasos y las funciones indicadas con más detalle.

2.2.1. Leer valores de variables de entorno

Las variables de entorno en C++ se pueden leer con la función `std::getenv()`. Como ilustramos en el ejemplo de la Sección 3.1 "Cadenas de caracteres" de [1], recomendamos crear una función `getenv()` propia en C++ que llame a `std::getenv()` y que devuelva un `std::string` con el valor de la variable de entorno o una cadena vacía, si la variable no está definida.

```
std::string getenv(const std::string& name);
```

2.2.2. Crear un *socket*

Para crear el *socket* que escuchará las conexiones, se implementará la función `make_socket()`, que se encargará de llamar a `socket()` para crear el *socket* y a `bind()` para asignarle el puerto indicado:

```
std::expected<SafeFD, int> make_socket(uint16_t port);
```

El *socket* creado debe ser de dominio `AF_INET`, tipo `SOCK_STREAM` y escuchar en el puerto `port`, en cualquier dirección IP disponible en la máquina.

> **!** Importante
>
> Para aprender cómo se usan `socket()` y `bind()`, consulta la documentación de los ejemplos de uso de *sockets* del capítulo 10.

La función debe devolver un `SafeFD` que contenga el descriptor de archivo del *socket* creado, en caso de éxito, o un `int` con el valor de `errno` en caso de error. Recuerda que esto se explica en la Sección 5.3 "Propagación de errores con `std::expected`" en [1].

2.2.3. Poner el *socket* a la escucha

El siguiente paso es poner el *socket* a la escucha para aceptar conexiones entrantes. Para ello, se implementará la función `listen_connection()` que llamará a `listen()` con el *socket* creado en `make_socket()`:

```
int listen_connection(const SafeFD& socket);
```

Esta función debe devolver un `int` con el valor de `errno` en caso de error o `ESUCCESS` si todo ha ido bien.

2.2.4. Aceptar una conexión

Posteriormente, dentro del bucle principal, se aceptará una conexión entrante con la función `accept_connection()`, que también implementaremos:

```
std::expected<SafeFD, int> accept_connection(const SafeFD& socket,
    sockaddr_in& client_addr);
```

Esta función debe usar la función `accept()` con el *socket* en `socket` para aceptar una conexión entrante.

La dirección y tamaño de `client_addr` debe pasarse como argumento a `accept()`, para que copie en ella la dirección del cliente que se ha conectado. Esto es necesario, por ejemplo, para poder mostrar la dirección del cliente conectado si está activo el modo detallado `-v`.

La función `accept()` devuelve un nuevo descriptor de archivos que sirve para comunicarse con el otro proceso a través de la conexión aceptada. Este descriptor de archivo debe ser retornado por `accept_connection()` en un `SafeFD`, para que el cierre de la conexión se haga automáticamente cuando el objeto `SafeFD` salga del ámbito.

En caso de error, `accept_connection()` debe devolver un `int` con el valor de `errno` del error.

> **!** Importante
>
> Para aprender como se usan `listen()` y `accept()` y se itera para aceptar conexiones, consulta la documentación de los ejemplos de uso de *sockets* del capítulo 10.

2.2.5. Responder a través de una conexión

Ahora la función `send_response()` debe enviar la respuesta a través del *socket* en lugar de mostrarla por la **salida estándar**, por lo que su declaración cambia a:

```
int send_response(const SafeFD& socket, std::string_view header,
    std::string_view body = {});
```

Donde `socket` es un `SafeFD` que contiene el descriptor de archivo del *socket* a través del que se enviará la respuesta. Este *socket* es el que se obtiene al aceptar una conexión con `accept()`.

La función `send_response()` debe enviar la respuesta al otro proceso usando `send()`. Pero `send()` puede fallar, motivo por el que ahora `send_response()` debe devolver un `int` con el valor de `errno` en caso de error o `ESUCCESS` si todo ha ido bien. Es decir, manejando los errores de la misma forma que en la función `read_file()` en la Sección 5.1 en [1].

Recuerda que si `send_response()` retorna `ECONNRESET` se trara de un **error leve**. Se debe mostrar un mensaje de error por la **salida de error** y cerrar la conexión, pero no se debe terminar el programa. El resto de las condiciones de error deben ser tratadas como errores fatales y provocar la terminación del programa.

2.2.6. Cerrar la conexión

Una vez que se ha respondido a través de una conexión, esta debe cerrarse con `close()`. Esto debería ocurrir automáticamente si el *socket* devuelto por `accept()` se guarda en un `SafeFD`, ya que el destructor de `SafeFD` se encarga de llamar a `close()` automáticamente cuando el objeto sale del ámbito.

3. Petición remota de documentos

3.1. Objetivo

En esta tercera parte vamos a permitir que el cliente indique el nombre del archivo que quiere ver, en lugar de tener que indicarlo en la línea de comandos de `docserver`. Por tanto, ahora el programa `docserver` debe aceptar los siguientes argumentos de línea de comandos:

```
docserver [-v | --verbose] [-h | --help]
          [-p <puerto> | --port <puerto>]
          [-b <ruta> | --base <ruta>]
```

Ya no hace falta aceptar un archivo como argumento, porque ahora el programa, tras aceptar la conexión, debe leer la petición del cliente, dónde se indica el archivo que le interesa. El programa debe responder con el contenido de ese archivo, si existe, o con un mensaje de error si no existe o no se tiene permiso para leerlo.

3.1.1. Opciones de línea de comandos

Las opciones `-h`, `-v` y `-p` operarán igual que en la versión anterior.

La nueva opción `-b` o `--base` permite indicar el directorio base donde buscará los archivos que le piden los clientes:

- Si no se indica la opción `-b`, el programa buscará los archivos en el directorio indicado en la variable de entorno `DOCSERVER_BASEDIR`.

- Si esta variable de entorno no está definida, el programa buscará los archivos en el directorio actual de trabajoo, que se obtiene con `getcwd()`.

3.1.2. Manejo de errores

En esta versión hay varios tipos nuevos de *errores leves*:

- Si al leer la petición del cliente con `recv()` se obtiene un error `ECONNRESET`, pues eso indica que el cliente ha cerrado la conexión antes de enviar la petición. Por tanto, se debe mostrar un mensaje de error por la **salida de error**, cerrar la conexión y esperar a la siguiente conexión
- Si la petición recibida del cliente está vacía o no sigue el formato esperado. En ese caso, el programa debe responder a través del *socket* con el siguiente mensaje de error, cerrar la conexión e iterar para aceptar y procesar la siguiente:

```
400 Bad Request
```

3.2. Implementación

El flujo de ejecución del programa es similar, pero ahora, tras aceptar una conexión, el programa debe leer la petición del cliente y procesarla para obtener el nombre del archivo que quiere ver.

…

`listen_connection()`: Poner el *socket* a la escucha
error-fatal if error al poner el *socket* a la escucha

loop
  `accept_connection()`: Aceptar una conexión
  error-fatal if error al aceptar la conexión

  `receive_request()`: Leer la petición del cliente
  if error `ECONNRESET` en `receive_request()`
    error
  else if otro error detectado
    error-fatal
  end if

  Procesar la petición del cliente
  if error de petición
    `send_response()`: Responder con el mensaje de error leve
    error
  end if

`read_all()`: Leer archivo en memoria
if error de permisos o archivo no encontrado

...

Cerrar la conexión (`SafeFD se encarga`)
end loop

Vamos a ver cada uno de los nuevos pasos y las funciones indicadas con más detalle.

3.2.1. Leer la petición del cliente

Para leer la petición del cliente, se implementará la función `receive_request()` que llamará a `recv()` con el *socket* devuelto por `accept_connection()`:

```
int receive_request(const SafeFD& socket, const std::string& request,
    size_t max_size);
```

Esta función debe intentar recibir usando la función `recv` un máximo de `max_size` bytes de la petición del cliente y almacenarla en el `std::string request`. En caso de éxito, `receive_request()` debe devolver ESUCCESS. Mientras que en caso de error, debe devolver un `int` con el valor de `errno`.

> **Nota**
>
> Esta función puede implementarse de forma similar a la función `read_file()` vista en [1], solo que llamando a `recv()` en lugar de a `read()` y utilizando `std::string` en lugar de `std::vector<uint8_t>` para el buffer.

La función debe llamarse con un valor de `max_size` relativamente alto –por ejemplo, 1024 o 4096– para intentar leer la petición del cliente en una sola llamada.

3.2.2. Procesar la petición del cliente

Una vez que se ha leído la petición del cliente, se debe procesar el `std::string` para obtener el nombre del archivo que se tiene que devolver.

Una petición está formada por varias líneas de texto, pero solo nos interesa la primera línea, que tiene la palabra `GET` seguida de la ruta del archivo, seguida de otras palabras que no nos interesan:

```
GET /ruta/del/archivo <otras palabras>
<otras líneas de texto>
```

> **Nota**
>
> Recuerda que puedes usar `socat` para enviar peticiones al servidor y comprobar que la respuesta es la esperada.
>
> ```
> $ socat STDIO TCP:127.0.0.1:8080
> GET /ruta/del/archivo                                    (1)
> Content-Length: 10                                       (2)
>
> Hola mundo
> $
> ```
>
> (1) Petición que enviamos al servidor para obtener el archivo `/ruta/del/archivo`.
> (2) Respuesta del servidor con el contenido del archivo.

Por tanto, para procesar la petición, se debe buscar la primera línea y extraer la ruta del archivo. Para ello se puede utilizar `std::istringstream`, que permite leer de una cadena de texto como si fuera un `std::istream`:

```
std::string str = "Hola mundo";
std::istringstream iss(str);

std::string hola_str, mundo_str;
iss >> hola_str >> mundo_str;
```

Observa que:

- Si la cadena está vacía, la petición no es válida (ver la Sección 3.1.2).
- La primera palabra debe ser `GET`, de lo contrario la petición no es válida (ver la Sección 3.1.2).
- La ruta del archivo siempre es absoluta, empezando por `/` –por ejemplo, `/README.txt`–. Esta ruta no se puede usar directamente con `read_all()`, sino que debe interpretarse como relativa al directorio base configurado (ver la Sección 3.1.1).

4. Contenido dinámico

4.1. Objetivo

El programa `docserver` que hemos desarrollado hasta ahora es capaz de servir archivos estáticos, pero no es capaz de servir contenido generado dinámicamente. Por ejemplo, no podemos hacer una petición que devuelva la fecha y hora actual, o cualquier otra información que no esté almacenada en un archivo.

Vamos a añadir esta característica de una manera sencilla. Dentro del directorio que estemos usando como directorio base, crearemos un directorio `bin/` que contenga los programas que se pueden ejecutar para generar contenido dinámico.

> **Nota**
>
> Este directorio `bin/` se puede crear a manualmente, no es necesario que el programa lo haga. Es algo que se supone que el usuario hace para que el programa pueda ejecutar los programas que generan contenido dinámico.

Los programas copiados en `bin/` deben ser ejecutables con los permisos adecuados. Cuando el cliente haga una petición a `docserver` con la ruta `/bin/ruta/al/programa`, el programa `docserver` reconocera que la ruta empieza por `/bin/` y sabrá que debe ejecutar el programa `ruta/al/programa` que se encuentra en el directorio `bin/` en el directorio base. Durante la ejecución, `docserver` capturará la **salida estándar** del programa y eso será lo que enviará al cliente a través del *socket*.

El programa `docserver` configurará una serie de variables de entorno con información para que el programa que se ejecute pueda saber más sobre la petición que se le ha hecho y el entorno en el que se está ejecutando:

- `REQUEST_PATH`: La ruta del archivo que el cliente indicó en la petición, que será la ruta del programa que se está ejecutando.
- `SERVER_BASEDIR`: La del directorio base que está usando `docserver`.
- `REMOTE_PORT`: El puerto desde el que se ha hecho la petición.
- `REMOTE_IP`: La dirección IP del cliente que ha hecho la petición.

4.1.1. Manejo de errores

En esta nueva funcionalidad, todos los errores posibles son **errores leves**:

- Si el programa que se intenta ejecutar no existe, se tratará igual que cuando el archivo no existe en la Sección 1.1.2.
- Si no se tiene permisos para ejecutar el programa, se tratará igual que cuando no se tienen permisos para leer un archivo en la Sección 1.1.2.
- Si el programa se ejecuta pero no termina con éxito, se manejarán enviando el siguiente mensaje al cliente y cerrando la conexión:

```
500 Internal Server Error
```

Si el programa no se puede ejecutar por cualquier otro motivo –como otros errores en `fork()` y `exec()`, se manejará mostrando un mensaje de error por la **salida de error**, incluyendo el código de error `errno` y el texto descriptivo obtenido mediante `std::strerror()`, y enviando el mismo mensaje que en el caso anterior al cliente y cerrando la conexión.

4.1.2. Comprobación

Para probar que la ejecución de programas funciona, podemos crear un programa sencillo en C++ o un *script* en BASH que devuelva la fecha y hora actual. Por ejemplo, crea un script `bin/time` en BASH que llame a `date` para mostrar la fecha y hora actual:

- Recuerda darle los permisos de ejecución necesarios con `chmod +x bin/time`.
- Aprovecha para incluir también en la respuesta un texto que contenga los valores de las variables de entorno que `docserver` configura: `REQUEST_PATH`, `SERVER_BASEDIR`, `REMOTE_PORT` y `REMOTE_IP`.

Una vez implementada esta funcionalidad, si haces una petición a `docserver` con la ruta `/bin/time`:

```
GET /bin/time
```

deberías poder ver la salida del *script*, con la fecha y hora actual y los valores de las variables de entorno en la respuesta.

4.2. Implementación

En la primera parte creamos la función `read_all()` que lee el contenido de un archivo en memoria:

```
std::expected<std::string_view, int> read_all(const std::string& path);
```

o

```
std::expected<SafeMap, int> read_all(const std::string& path);
```

Ahora vamos a crear una función similar que ejecute un programa y devuelva una cadena con su **salida estándar**:

```
struct execute_program_error {
    int exit_code;
    int error_code;
};

std::expected<std::string, execute_program_error>
execute_program(const std::string& path, const exec_environment& env);
```

donde:

- `path` es la ruta del programa que se va a ejecutar.

- `env` es un objeto de tipo `exec_environment` que tendrás que definir con los atributos apropiados para que contenga la información necesaria para configurar las variables de entorno comentadas en la Sección 4.1.

La función `execute_program()` debe retornar un `std::string` con la **salida estándar** del programa si todo ha ido bien, o un objeto `execute_program_error` con el código de salida del programa o el valor de `errno` si ha habido algún error:

- Si `error_code` es `ESUCCESS`, significa que el programa se ha ejecutado correctamente y `exit_code` contiene el código de salida del programa. Recuerda que es un error si el código de salida no es 0.
- Si `error_code` es distinto de `ESUCCESS`, significa que ha habido un error en `fork()`, `exec()` u otra función, y contiene el valor de `errno`. Recuerda que se usan los códigos `ENOENT` y `EACCES` para indicar que el programa no existe o no se tiene permisos para ejecutarlo, respectivamente.

Respecto a donde invocar a `execute_program()`, se debe hacer en el punto del pseudocódigo donde se llama a `read_all()` para leer el archivo. Si la ruta del archivo indicada en la petición del cliente empieza por `/bin/`, se debe llamar a `execute_program()`, mientras que en otro caso se debe llamar a `read_all()`.

> **Nota**
>
> En caso de éxito, la función `execute_program()` debe devolver la salida estándar del programa como un `std::string`, mientras que `send_response()` recibe en su argumento `body` un `std::string_view`. Esto no es problema, ya que un `std::string` se convierte implícitamente en un `std::string_view`.

4.2.1. Búsqueda del programa y comprobación de permisos

Antes de crear el procesos hijo, es buena idea verificar que el programa que se va a ejecutar existe y que se tienen permisos para ejecutarlo. Para eso, se puede usar la función `access()` con el modo `X_OK`.

Si esta comprobación falla, se debe volver la función con el valor de `errno` correspondiente.

4.2.2. Ejecución de programas

La ejecución del programa debe implementarse usando `fork()` y `exec()`.

El flujo de ejecución del programa padre debería ser el siguiente:

> Crear una tubería para capturar la salida estándar del proceso hijo.
> Crear un proceso hijo con `fork()`.
> **error if** error en `fork()`

Leer la salida de la tubería con `read()` hasta que devuelva 0.
error if error en `read()`

Esperar a que el proceso hijo termine con `waitpid()`.
error if el proceso hijo no terminó con éxito.
Devolver la salida estándar del proceso hijo como `std::string`.

Mientras que el flujo de ejecución del proceso hijo debería ser el siguiente:

Redirigir la salida estándar del proceso hijo a la tubería.
Configurar las variables de entorno (ver Sección 4.1).
Ejecutar el programa con `exec()`.
exit if error en `exec()`.

> **!** Importante
>
> Para aprender cómo se usan `fork()` y `exec()`, consulta la documentación de los ejemplos de procesos del capítulo 9.
> Para aprender cómo se usan tuberías para redirección la E/S estándar de entre proceso padre e hijo, consulta la documentación de los ejemplos de uso de tuberías del capítulo 10.

Respecto a los estados de salida del proceso hijo, se debe tener en cuenta que:

- Por convención, si el programa se ejecuta correctamente, el proceso hijo terminará con el código de salida '0'. Eso permite saber al proceso padre que el programa se ha ejecutado correctamente y que la salida estándar del proceso hijo que ha leído es adecuada para enviarla al cliente.
- Si el programa no termina con el código de salida '0', el proceso padre debe considerar que ha habido un error y devolver dicho código de salida como un error a través de `execute_program_error`.
- Si `exec()` falla y el programa no se puede ejecutar, también debemos terminar el proceso hijo con un código de salida distinto de '0', para que el proceso padre sepa que ha habido un error. En este caso se recomienda usar un código de salida con un valor relativamente alto, para evitar colisiones con los códigos de salida usados por los programas que logran ejecutarse. BASH, por ejemplo, en estos casos suele devolver el código 126 o 127.

> **i** Nota
>
> Recuerda que los códigos de salida válidos son valores enteros entre 0 y 255.

A. Servidor web básico

Enhorabuena, ya que si has hecho al menos hasta la Sección 2, prácticamente has implementado un servidor web básico.

Si no puedes abrir Chrome, Edge o Firefox y escribir `http://localhost:8080/archivo.txt` o `http://localhost:8080/bin/time` en la barra de direcciones para ver el contenido de un archivo o la salida de un programa, es porque el protocolo que hemos implementado difiere ligeramente del protocolo que usan los navegadores y servidores web.

Si tienes curiosidad porque `docserver` pueda servir archivos a través de un navegador, te proponemos que hagas los siguientes cambios.

A.1. Nuevo formato de respuesta

El nuevo formato de respuesta es el siguiente:

A.1.1. En caso de éxito

```
HTTP/1.1 200 OK
Content-Length: <TAMAÑO>
Content-Type: <TIPO>; charset=UTF-8";

<contenido del archivo...>
```

Donde:

- `<TAMAÑO>` es el tamaño del archivo en bytes.
- `<TIPO>` es el tipo MIME del archivo. Esto quiere decir que debemos mirar la extensión del archivo solicitado. Si el archivo termina en `.txt`, `.md` u otras extensiones que relacionamos con archivos de texto, el tipo es `text/plain`. Si el archivo termina en `.html`, `.htm` u otras extensiones que relacionamos con archivos HTML, el tipo es `text/html`.

Para probarlo podemos crear una archivo `index.html` en el directorio base con el siguiente contenido:

```
<!DOCTYPE html>
<html>
<head>
    <title>¡Hola mundo!</title>
</head>
<body>
    <h1>¡Hola mundo!</h1>
```

```
    <p>Este es un archivo HTML de prueba.</p>
</body>
</html>
```

Cuando hayas terminado de hacer todos los cambios propuestos, deberías poder ver esta página si hace una petición a `http://localhost:8080/index.html` desde un navegador.

A.1.2. En caso de error por no tener permiso

```
HTTP/1.1 403 Forbidden
```

A.1.3. En caso de error por no encontrar el archivo

```
HTTP/1.1 404 Not Found
```

A.1.4. En caso de error por petición incorrecta

```
HTTP/1.1 400 Bad Request
```

A.1.5. En caso de error interno del servidor

```
HTTP/1.1 500 Internal Server Error
```

A.2. Salto de línea

Los saltos de línea en HTTP deben ser `\r\n` en lugar de `\n`. Es decir:

- Los saltos de línea en la cadena que se entrega a `send_response()` a través de la variable `header` deben tener un `\r` antes de cada `\n`.
- El espacio en blanco que genera `send_response()` entre `header` y `body` también debe tener un `\r\n` en lugar de un solo `\n`.

Por ejemplo, las líneas de la 1 a la 4 de la siguiente respuesta deben usar un salto de línea en formato `\r\n`:

```
HTTP/1.1 200 OK
Content-Length: 45
Content-Type: text/plain; charset=UTF-8

Hola mundo
Hola mundo
Aquí termina el archivo
```

Mientras que en esta respuesta de error, las líneas 1 y 2 deben tener un salto de línea en formato `\r\n`:

```
HTTP/1.1 404 Not Found

```

A.3. Petición de documentos

Las peticiones enviadas por el navegador siguen el siguiente formato:

```
GET /ruta/del/archivo HTTP/1.1
<otras líneas de texto>
```

Los saltos de línea en las peticiones también deben ser `\r\n`, pero en la práctica es costumbre aceptar también los que usan solo `\n`.

Si has implementado la función `receive_request()` como se indica en la Sección 3, el programa ya debería estar preparado para recibir peticiones de este tipo.

Referencias

[1] J. Torres-Jorge, Programación de aplicaciones — Consejos al programar en C++, en: Prácticas de Sistemas Operativos, Universidad de La Laguna, 2024. https://github.com/ull-esit-sistemas-operativos/ssoo-practica-consejos-cpp/blob/so2425/_output/consejos-cpp.pdf.