Exercise — Epoll Server

version **#dirty**

Assistants C/Unix 2025 <assistants@tickets.assistants.epita.fr>

Contents

*https://intra.forge.epita.fr

File Tree

```
epoll_server/
├─ connection.c
├─ connection.h
├─ epoll_server.c   (to submit)
├─ epoll_server.h
├─ meson.build
└─ utils/
   ├─ xalloc.c
   └─ xalloc.h
```

Compilation : Your code must compile with the following flags

- -std=c99 -Werror -Wall -Wextra -Wvla

Main function : Required

Authorized functions : You are only allowed to use the following functions

- bind
- fcntl
- fcntl64
- socket
- send
- recv
- listen
- freeaddrinfo
- getaddrinfo
- setsockopt
- accept
- close
- epoll_create1
- epoll_ctl
- epoll_wait
- __ubsan_handle_pointer_overflow
- __ubsan_handle_sub_overflow
- __ubsan_handle_out_of_bounds
- __ubsan_handle_type_mismatch_v1

- __ubsan_handle_nonnull_arg
- __ubsan_handle_add_overflow
- __ubsan_handle_mul_overflow

Authorized headers : You are only allowed to use the functions defined in the following headers

- assert.h
- ctype.h
- errno.h
- fcntl.h
- unistd.h
- sys/epoll.h
- stdbool.h
- stdio.h
- stdlib.h
- string.h
- err.h

1 Introduction

Epoll is a Linux kernel syscall that allows you to monitor multiple file descriptors and receive an event when an I/O operation is possible on them. You **SHOULD** read the `epoll(7)` man page carefully, as it contains all the information you need about this syscall.

The main problem is that I/O operations on sockets can block. For example, a call to `recv(2)` while there is nothing to read in the socket will block until a message is received and ready to be read. During this time, if another client tries to communicate, the message cannot be handled until the server is unblocked by the first client. In this exercise, we will show you the modern way to solve this problem.

Going further...

We will first introduce you to polling with `epoll`. This part is mandatory and represents the core feature of this exercise.

We will then introduce you to different improvements over `epoll` such as non-blocking I/O to optimize your server even more. These parts will not be graded but you are encouraged to implement them nonetheless. You can see them as a way to go further.

2 Basic server

This section will be focused on a basic server that can receive a connection. First, if you have any questions you **SHOULD** read Beej's guide to network programming. This explanation is only a simplified part of the guide's explanation without all the explanations on the arguments.

2.1 getaddrinfo(3)

The first function that you will need is `getaddrinfo(3)`. This function allows you to get information about how to connect to a network service. It takes a hostname, a service, some hints and a structure that it will fill.

Tips

In this exercise we will be using an IPv4 TCP connection, you can read the `getaddrinfo(3)` to see which hints you might need.

2.2 socket(2)

This function will create a socket using the provided arguments and give you the corresponding file descriptor. Currently, this socket is not bound to any port on your machine. The next function will handle that.

Tips

You can read the `socket(2)` for more information.

2.3 bind(2)

This function will bind the socket that you just created to one of the network port on your machine.

Tips

This function is only useful if you are the server, a client can connect without binding the socket to a port.

You can read the `bind(2)` for more information.

2.4 connect(2)

When you are the client, you will need to use this function to initiate the connection to the other party. In this exercise, you will not use this function as you are the server.

Tips

You can read the `connect(2)` for more information.

2.5 listen(2)

This function will mark the socket as a passive for listening and configure how many connections will be allowed in the incoming queue.

Tips

You can use the `SOMAXCONN` macro as the backlog.

You can read the `listen(2)` for more information about the function and this macro.

2.6 accept(2)

The `accept` function blocks the execution of your program while waiting for a client to connect. When a client successfully connects, the function unblocks, and you can start exchanging information with them through the returned socket.

Tips

You can read the `accept(2)` for more information.

Be careful!

You can only accept one connection at a time. This means that the following connections depends on how long you handle the current connection.

We will fix that problem later with epoll.

2.7 send(2)/recv(2)

These two functions allow you to send and receive data. These functions are blocking: if you send a message to the client and it does not read (or you read while the client did not send) your code will stop indefinitely by default. There is a socket option to change this parameter.

Tips

You might not be able to send or receive all the data in one call, be sure to check the number of bytes sent/read.

You can read the `send(2)` and `recv(2)` for more information.

2.8 close(2)

Once you handled all the data from the connection that you needed, you will need to close the connection. You are expected to close all the sockets that you opened (in the same way that you free the memory that you allocate).

Tips

You can use `valgrind` with the option `--track-fds=yes` to check that you closed all of your file descriptors.

You can read the `close(2)` for more information.

3 Epoll

As we just saw, you can handle a single connection at a time using sockets directly. `epoll` will be able to scale easily to handle multiple sockets at a time.

`epoll(7)` is a mechanism for multiplexing: it gives one super-file descriptor that listens to all clients at once, by monitoring events on file descriptors.

It holds a list of file descriptors and waits for I/O events happening on them. In our case, we use `epoll` to handle communication with multiple clients by watching events on our sockets.

3.1 How it works

An *epoll instance* is a data structure held by the kernel which contains information about file descriptors:

- The interest list: the set of files descriptors that are currently being watched by epoll.
- The *ready* list: a subset of the interest list containing file descriptors that are ready for I/O. To explain the concept of *ready for I/O*, let us take an example with a socket: You can write bytes to a socket or read the content of a socket's buffer. If a server has a connected socket registered in an *epoll instance* and the connected client sends a message through this socket, `epoll` will notice activity on the socket. The *epoll instance* will know that data has arrived on the socket and will add the socket in the *ready list* which can later be retrieved.

Epoll is composed of three syscalls:

- `epoll_create1(2)`: returns a file descriptor referring to a newly created *epoll instance*.
- `epoll_ctl(2)`: lets you register, modify, or delete a file descriptor entry in the interest list of an *epoll* instance. Its last parameter is a struct in which the *events* field allows you to specify which type of events should be monitored for the given file descriptor. Among all possible values, the `EPOLLIN` and `EPOLLOUT` flags configure `epoll` to consider events for *reading* (resp. *writing*) operations and the `EPOLLET` flag sets the event distribution mode, which we will talk about later on.
- `epoll_wait(2)`: wait for events on the registered file descriptors. A call to `epoll_wait` will block until the specified timeout is over or at least one of the registered file descriptors is marked

ready. If a file descriptor was already marked ready before the call, the syscall returns immediately. It fills the memory area given in parameter (*events*) with *epoll's ready list*'s content (ready file descriptors) and returns the number of ready file descriptors.

Here is an example of a simple *epoll* initialization and file descriptor registration:

```
// Create an epoll instance
int epoll_instance = epoll_create1(0);
if (epoll_instance == -1)
{
    // Error handling
}

// Initialize the event struct containing the flags and optional user data
struct epoll_event event;
event.events = EPOLLIN;

// Add the socket to the epoll instance's interest list.
if (epoll_ctl(epoll_instance, EPOLL_CTL_ADD, socket_fd, &event) == -1)
{
    // Error handling
}


while (true)
{
    struct epoll_event events[MAX_EVENTS];
    int events_count = epoll_wait(epoll_instance, events, MAX_EVENTS, -1);
    if (events_count == -1)
    {
        // Error handling
    }

    for (int event_idx = 0; event_idx < events_count; event_idx++)
    {
        // Handle ready file descriptor
    }
}
```

Going further...

In the example above, we used `epoll_create1(2)`, the successor of `epoll_create(2)`. `epoll_create(2)`'s parameter (the number of expected file descriptors) is no longer used. On the other hand, `epoll_create1(2)` takes a flag as parameter to change the behavior of the returned file descriptor. We invite you to read the `epoll_create(2)` man page for further information.

3.2 Edge-Triggered vs. Level-Triggered Mode

`epoll(7)` has two triggering modes that control under which conditions `epoll_wait(2)` will stop blocking.

Tips

This subject is not covered in detail here. However, you **SHOULD** read the `epoll(7)` manpage and conduct your own research to ensure you understand these two modes.

3.2.1 Level-Triggered Mode

In level-triggered mode, `epoll_wait(2)` always returns if at least one of the registered file descriptors is available for an I/O operation. This mode is the default behavior of `epoll(7)`.

Tips

We recommend using this mode in this exercise.

3.2.2 Edge-Triggered Mode

Edge-triggered mode, denoted as `EPOLLET`, instructs `epoll_wait(2)` to return as soon as there is a new event available for your file descriptors.

4 Goal

The goal of this exercise is to create a basic server that will use *epoll* to properly handle multiple clients. Listed below are the general steps of your server's behavior:

1. Set up a socket listening for incoming connections and register it in your *epoll instance*.
2. Wait indefinitely for an event to occur: a client trying to connect, a connected client sending a message to the server or a client disconnecting.
3. Handle the event(s) by accepting or communicating with a client.
4. Loop back to 2.

The third point is where all the server's logic is implemented. This is where your server will accept clients, but also receive their messages and send its response.

Events should be handled differently depending on the socket type.

- Listening socket: an event will be triggered when at least one client tries to connect to your server. Your socket will be marked ready whenever a client is trying to connect, but you cannot know how many. If you call `accept(2)` when no client is trying to connect, you will block and wait for connections. When receiving an event, you must therefore accept a single client. The next ones will be handled by another event in your next call to `epoll_wait(2)`.

- Other sockets (connected to a client): you cannot know whether or not a client has sent its full message. Thus, your server must read until it has received the full message but you have to be careful not to block in `recv(2)`. For the same reasons as for the listening socket, you must make a single call to `recv(2)` when the socket is marked ready. If the message is not complete, you will have to save what you received and wait for another event on the socket to receive the rest of the data.

Tips

epoll_event.data is a handy way of associating data to a given socket.

Tips

A message is complete when it ends with a *\n*.

Be careful!

Do not accept or receive twice for the same event, you might block in `accept(2)` or `recv(2)` as you do not know how much data is left in the socket.

5 Assignment

Be careful!

Remember that you have to implement a `main()` function.

You have to integrate `epoll(7)` in the server you are making to be able to manage multiple clients at once.

If an error occurs with a client and your server is not impacted, you should close the connection with this client instead of crashing your server.

5.1 Meson

To compile your code as an executable, you will be using Meson (version 1.1.0) as the build system. If you need help you can read the documentation or rewatch the presentation of the project.

We will be using the following commands to compile your project:

```
meson setup build
meson compile -C build
```

5.2 Usage

Your program will be launched with two arguments. You should return `1` if you do not receive the ip and port.

```
42sh$ ./epoll_server ip port
42sh$ ./epoll_server 127.0.0.1 8000
```

Be careful!

We recommend you to use netcat to quickly test your server.

```
42sh$ nc 127.0.0.1 8000
Hello spider # what you typed on the keyboard
Hello spider # what you received from the server
```

Tips

You can read the `nc(1)` for more information about netcat.

5.3 Step 1

First, you will create an *epoll* instance at the beginning of your server.

Once your instance is created, you can register your sockets with `epoll_ctl(2)`. For the mandatory features, `EPOLLIN` is the only event flag you will need.

Going further...

`EPOLLOUT` is another event flag monitoring readiness for output, that is, the guarantee that a `send(2)` will not block. A blocking `send(2)` occurs when the inner buffer of the socket is full, but most of the time, sockets are ready for output. Handling that case is an optimization that will only be seen in bonuses. For now, you can consider `send(2)` will not block.

Be careful!

Even though you can assume `send(2)` will not block, it does not mean that your message will be sent with a single call. You still have to check if everything has been sent.

5.4 Step 2

Now that your instance is created, you have to adapt your server loop to use `epoll`: you must wait indefinitely until one or more sockets are marked ready and then handle the event.

In this step, we only ask you to integrate `epoll_wait(2)` and to handle events on your listening socket: you must check if an event occurred on your listening socket and accept a **single** client.

5.5 Step 3

In this step, you have to handle client sockets in your server loop: you must handle the communication with your clients.

This server is a broadcast server which means that when a client sends a message, the server must send it back to every client currently connected (including the original sender).

For now, you can consider that all messages will be sent and received at once and your client will stay connected to your server.

Tips

To test this feature, you can launch multiple instances of `netcat` connected to the same server and see whether messages sent from one client are received by all others clients.

5.6 Step 4

In this step you must handle two edge cases:

- Long messages: messages could be too long to be sent or received at once. Thus, you might need to wait for another event to occur on the socket in order to receive the whole message. You must store what you already received until you get the end of the message.
- Client disconnection: a client could have a problem during its execution and get disconnected. In this case, an input event will be detected and `recv(2)` should return `0`. From `recv(2)` man page: “When a stream socket peer has performed an orderly shutdown, the return value will be 0 (the traditional “end-of-file” return)”. In this case, you must close the client socket (this will also automatically unregister it from *epoll*).

Tips

To test the client disconnection case: launch a server and two clients, A and B. Send a message from A and then from B. Then, disconnect A. If *epoll* still detects events for A you have not handled client disconnection properly.

Seek strength. The rest will follow.