

PROCESS COMMUNICATION

Alex Pajuelo and Juan Jose Costa

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
BarcelonaTech

Cooperative processes

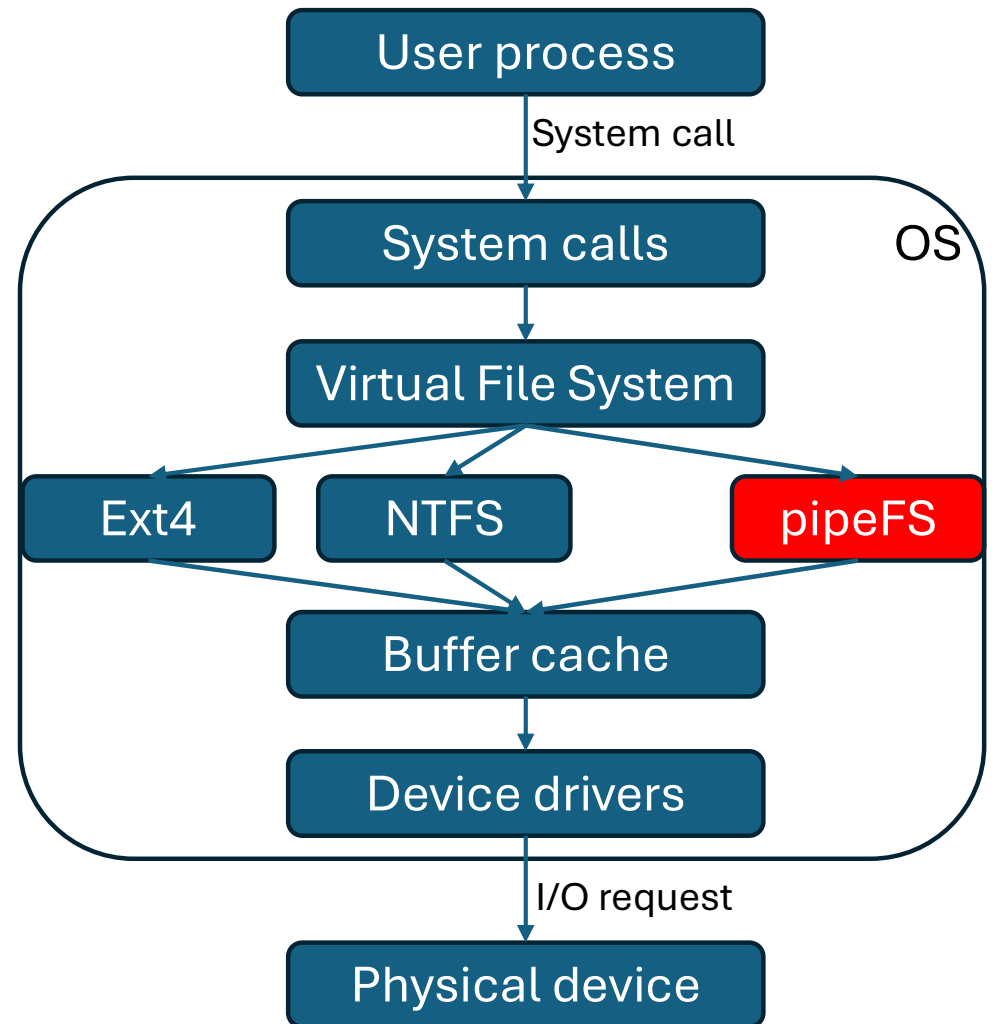
- Many times, a user executes several processes at the same time to success in a final task

```
> ls -l | wc -l
```

- Those processes cooperate to obtain the result
- In this case, some communication is needed among processes during the task
- The OS provides several mechanism to transfer information from one process to another
- Pipes and sockets

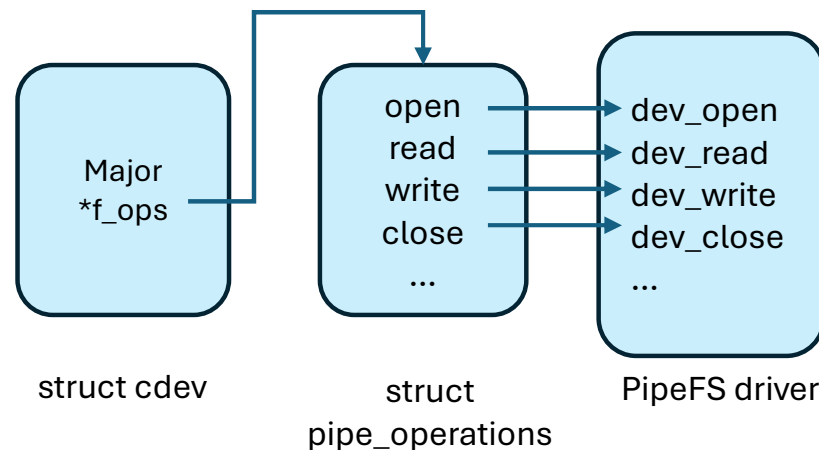
Pipes

- A pipe is a unidirectional in-memory data stream that operates as a First-In, First-Out (FIFO) buffer
- Pipes are implemented using the Virtual File System (VFS) framework, but the data never touches a physical disk—it exists purely as a set of page frames in kernel RAM.
- Remember that there are two kind of pipes: anonymous pipes (or simply pipes) and named pipes



PipeFS

- When the OS boots, the VFS explicitly loads the PipeFS file system driver and stores its Major for later use
- Remember that loading a driver means to allocate a struct cdev with the pair <Major, struct file_operations*>
- In this case, the struct file_operations* will point to the struct pipe_operations



sys_pipe syscall

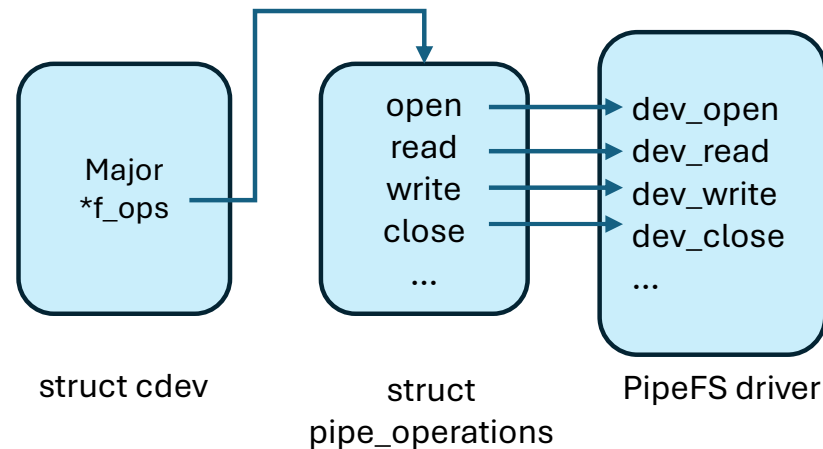
- sys_pipe syscall creates an anonymous pipe

```
int sys_pipe(int fd[2]);
```

- It returns 2 file descriptors: one for reading, fd[0], and another for writing in the pipe, fd[1]
- Sys_pipe, internally, calls vfs_pipe

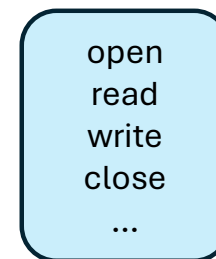
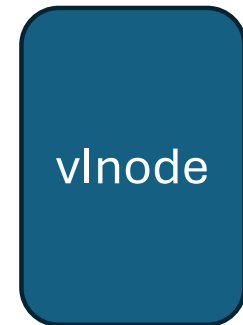
vfs_pipe

- First, `vfs_pipe` searches the corresponding struct `cdev` using the previously stored PipeFS Major. From the struct `cdev`, recovers the struct `pipe_operations`.



vfs_pipe

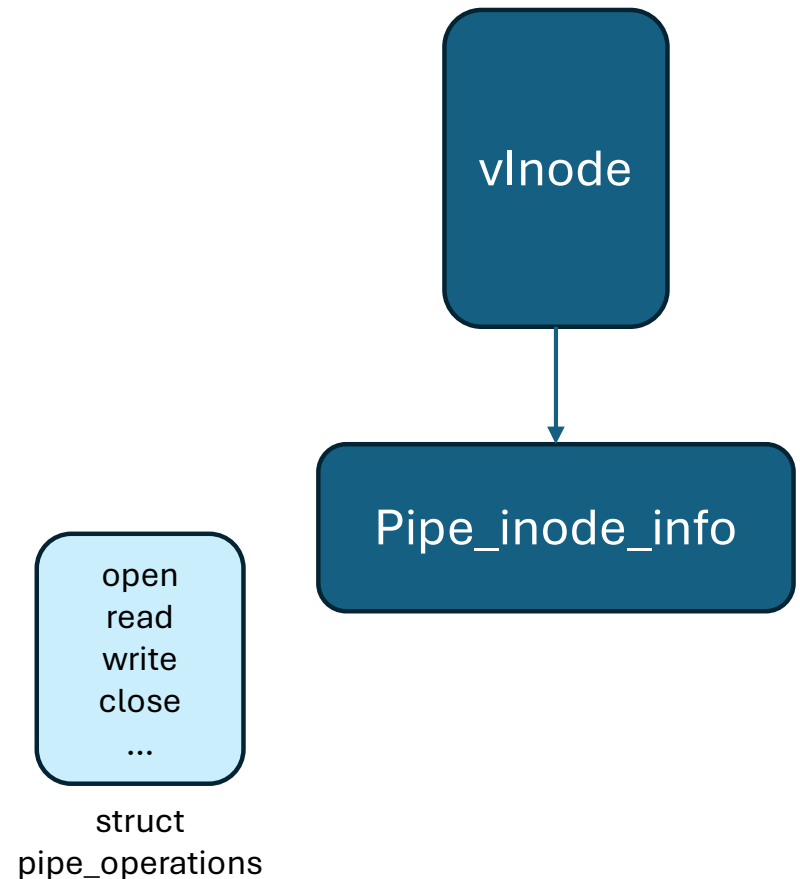
- After that, `vfs_pipe` creates a virtual Inode and passes it to the `pipe_open` function of the OpenFS driver



struct
pipe_operations

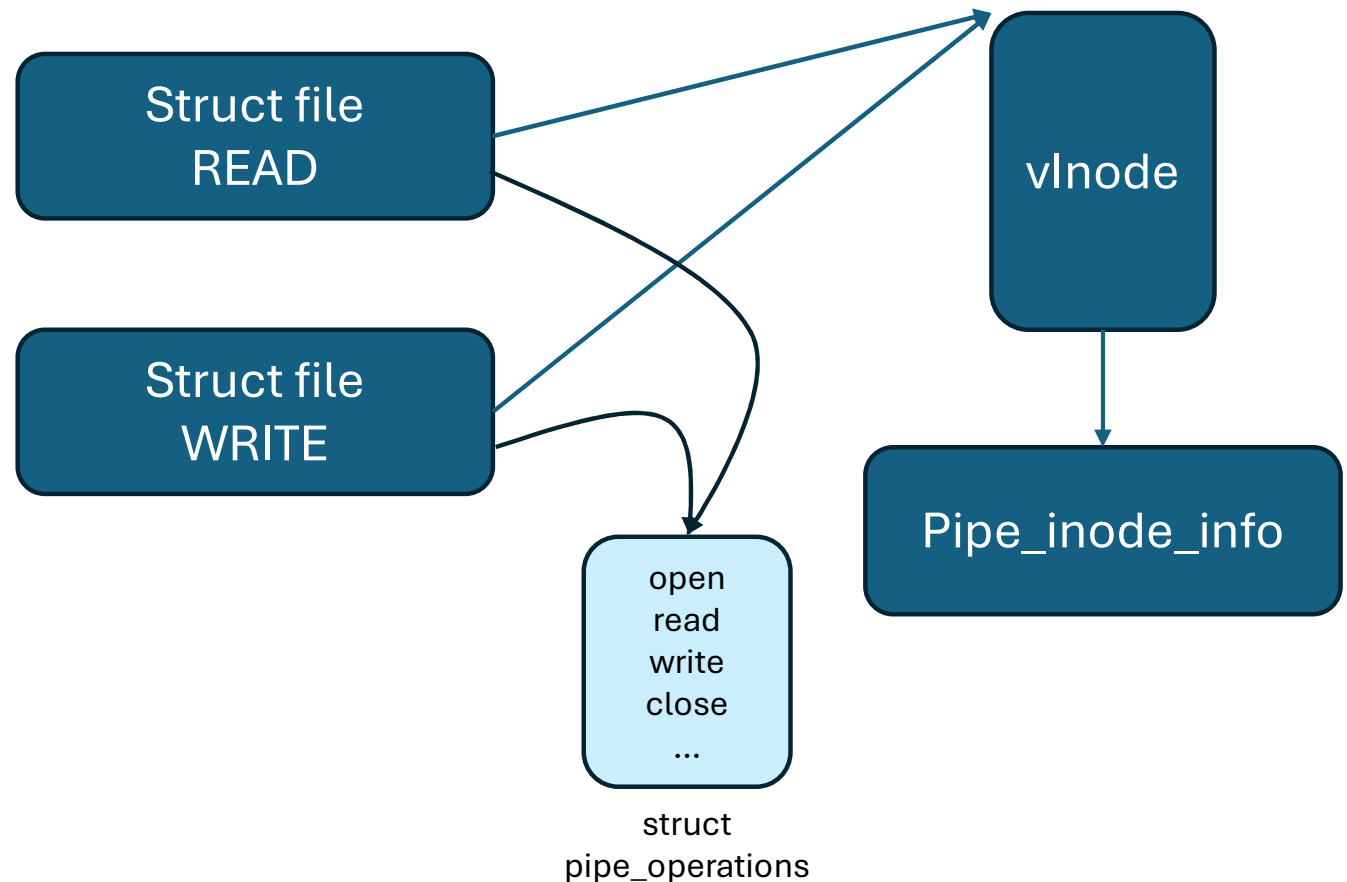
vfs_pipe

- Pipe_open creates a struct pipe_inode_info (described later) and stores a pointer to that struct in the virtual Inode



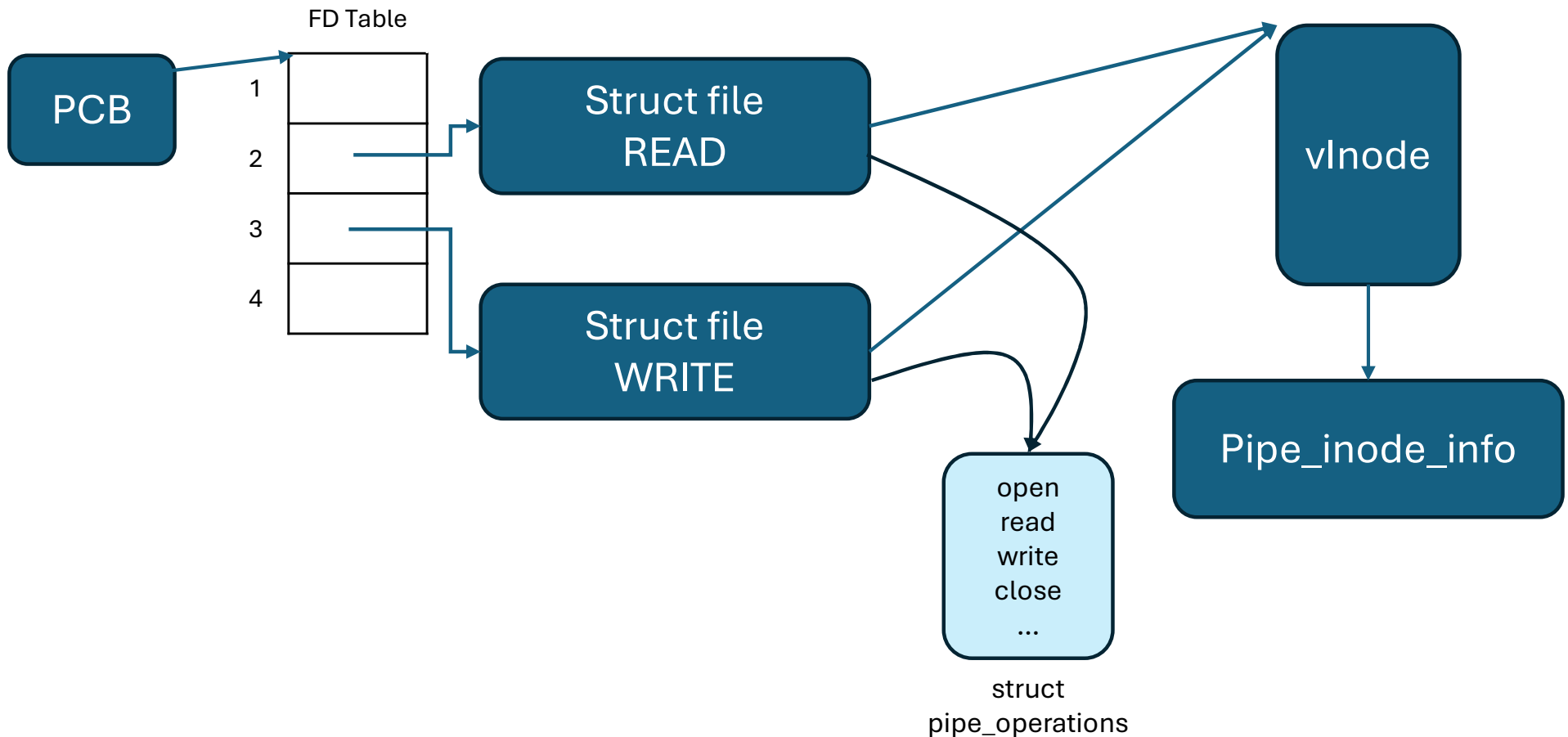
vfs_pipe

- After that, `vfs_pipe` allocates two struct file, one for reading and another for writing, pointing them to the previously allocated virtual Inode



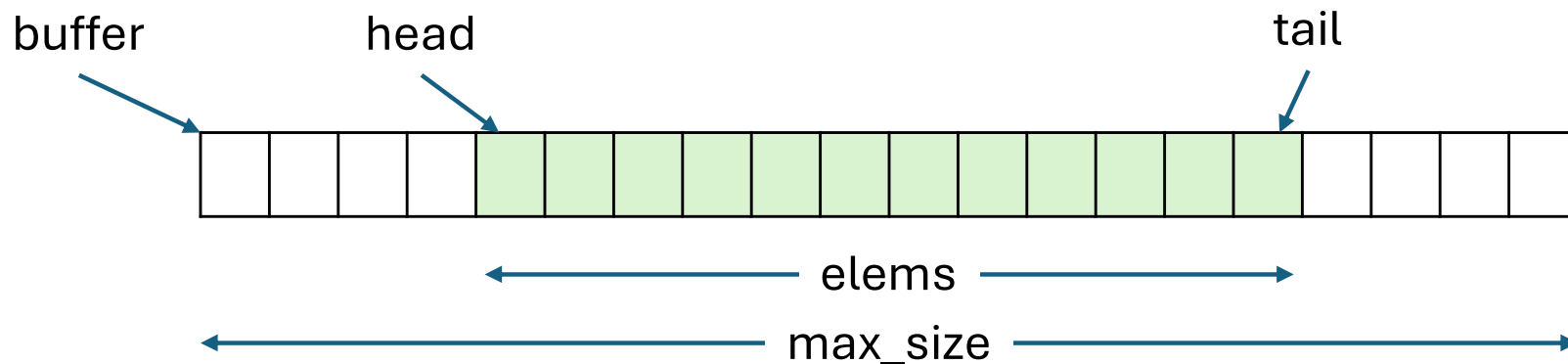
vfs_pipe

- Finally, `vfs_pipe` creates the references in the File Descriptor Table of the process to the created struct file



Struct pipe_inode_info

- Every pipe created in the system is backed by an internal kernel structure called struct pipe_inode_info allocated by the pipeFS driver
- A pipe is implemented as a ring buffer



- Buffer: pointer to the ring buffer
- Head: first position to read from
- Tail: first available position to write to
- Elems: available elements in the buffer
- Max_size: size of the buffer

Struct pipe_inode_info

- Remember also that:
- Reading from a pipe blocks if the pipe is empty and there are writers
- Writing to a pipe blocks if the pipe is full and there are readers
- This means that two queues are needed to block threads when needed, one for readers and another for writers
- The number of readers and writers must be tracked

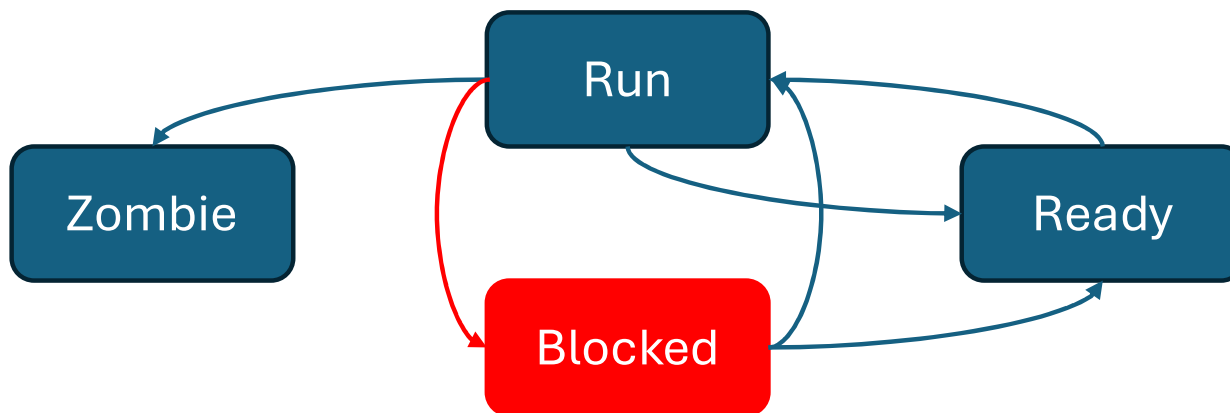
Struct pipe_inode_info

```
struct pipe_inode_info {  
    struct list_head read_wait;    // Blocked reading threads  
    struct list_head write_wait;   // Blocked writing threads  
    unsigned int head;             // Head of the ring buffer  
    unsigned int tail;             // Tail of the ring buffer  
    unsigned int max_usage;        // Size of the ring buffer  
    unsigned int elems;            // Elements in the ring buffer  
    unsigned int readers;          // Number of readers of the pipe  
    unsigned int writers;          // Number of writers of the pipe  
    struct pipe_buffer *buffer;    // Pointer to the buffer  
};
```

Blocking readers

- Reading from a pipe blocks if the pipe is empty and there are writers

```
if ((elems == 0) && (writers > 0))  
{  
    // Block the current thread  
    list_add_tail(&current()->anchor, &read_wait);  
    sched_next();  
}
```

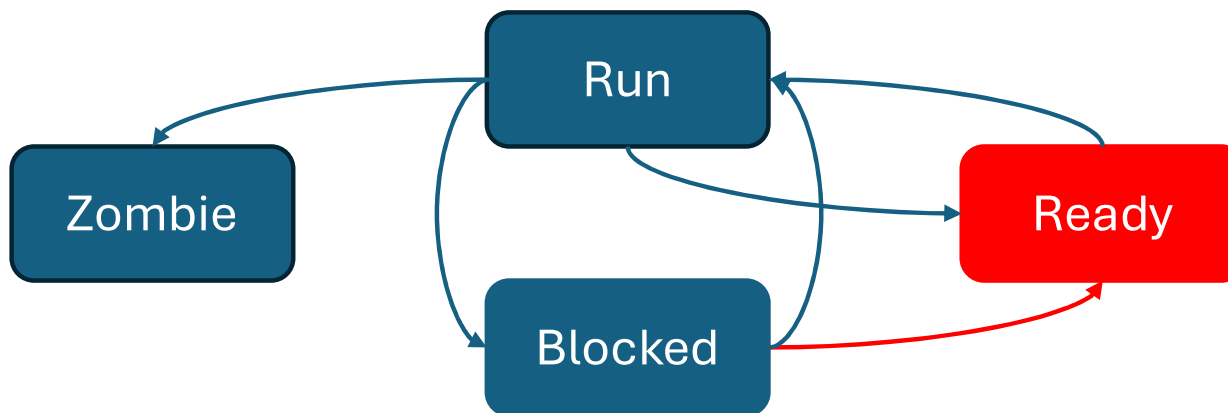


Blocks the thread

Unblocking readers

- When a writer writes in the pipe, it must unblock a blocked reader

```
if (!list_empty(&read_wait))
{
    // Unblock the current thread
    struct list_head *e = list_first(&read_wait);
    list_del(e);
    list_add_tail(e, &Readyqueue);
}
```



Unblocks the thread
queueing it in the
Ready queue

Blocking/Unblocking writers

- Writers must perform the same checks but having into account that a writer must be blocked if the pipe is full and there are readers
- In addition, readers must check if there is any blocked writer and unblock it after reading the pipe

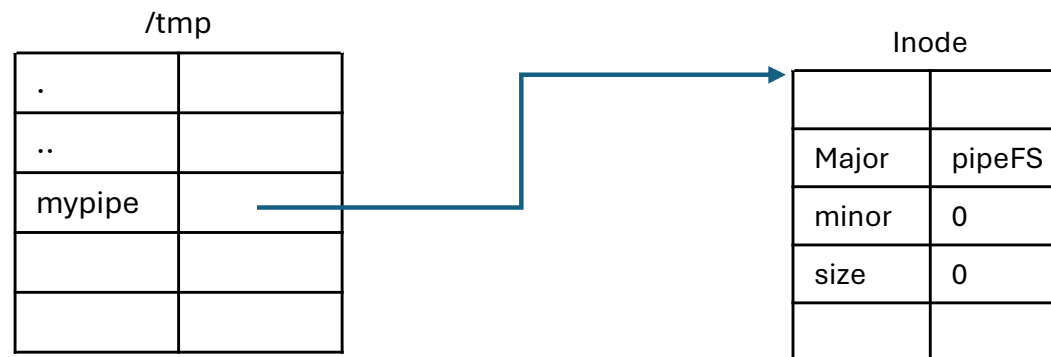
Named pipes

- Anonymous pipes can only be used by processes with a parent-child relationship since file descriptors do not point to an existing logical device in the file system
- To communicate unrelated processes, named pipes must be used
- Their behavior is similar to the anonymous pipes but named pipes are represented in the file system with a logical device that must be created before using them
- To communicate two processes with a named pipe, they must open it as an ordinary file
- The behavior of named pipes and anonymous pipes is the same when reading/writing

Creating a named pipe

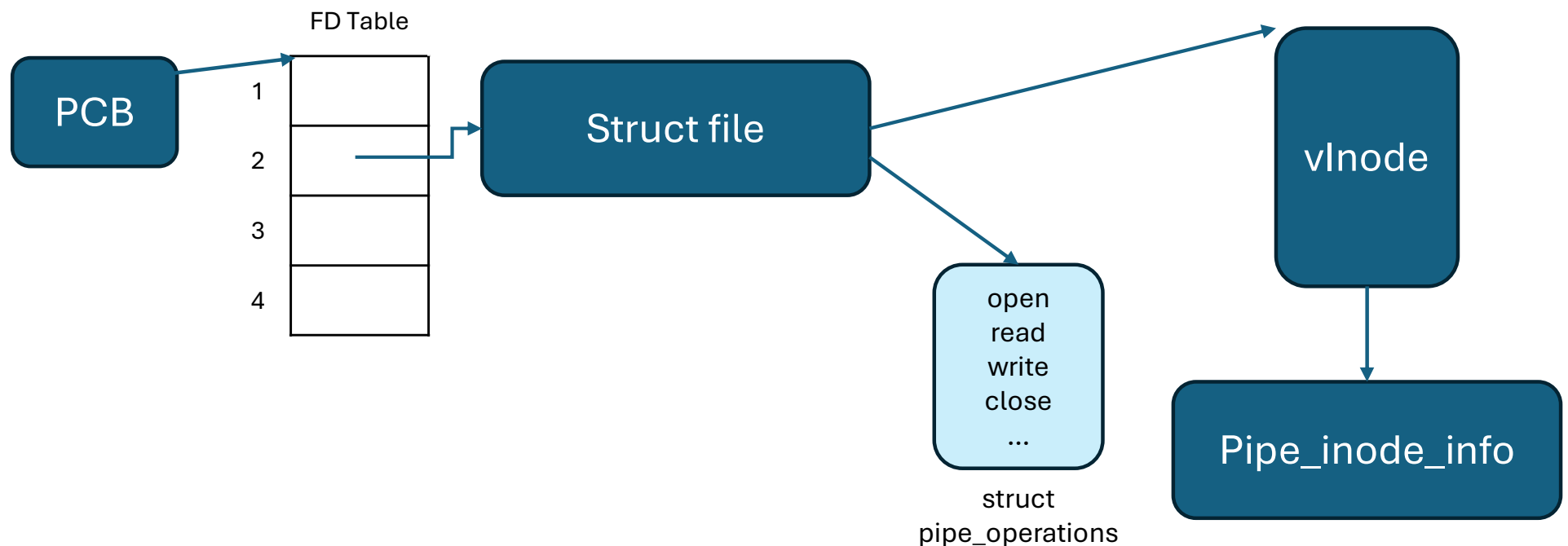
- To create a named pipe, the system call `mkfifo` must be used
- This system call creates a logical device which Inode has as Major the corresponding to the pipeFS file system device
- The named pipe will always have a size of 0 even if some data is written

```
> mkfifo /tmp/mypipe
```



Opening a named pipe

- To open a named pipe, the open system call must be used
- It works like pipe but only allocates one struct file and returns one file descriptor.



Opening a named pipe

- Remember that opening a FIFO for writing will block until another process opens it for reading, and vice versa. Data flows only when both ends are open.

```
int pipe_open(struct pipe_inode_info *p, int mode)
{
    if (mode & O_WRONLY)
    {
        p->nr_writers++;
        // If there are no readers, block
        if (p->nr_readers == 0)
        {
            list_add(&current(), &p->write_wait);
            sched_next();
        }
        // At least one readers, unblock all blocked readers if any
        while (!list_empty(&p->read_wait))
        {
            struct list_head *e = list_first(&p->read_wait);
            list_del(e);
            list_add(e, &ready_queue);
        }
    }
    ...
}
```

Sockets

- Logical bidirectional communication device that can be used to communicate processes that are on the same machine or processes on different machines across the network
- To create a socket, it is necessary to define
 - Type of communication
 - Namespace
 - Communication protocol

Type of communication

- Connection-oriented (stream)
 - A virtual circuit is established through which the information will be sent
 - When sending information, it is not necessary to specify the destination address
- Connectionless (datagram)
 - The circuit is not established: for each packet, a free link must be found
 - The recipient is specified in every transmission
 - Neither the delivery of packets nor the reception order is guaranteed

Namespace

- To specify source and destination address
- If sockets are used to communicate processes within a single machine: file system namespace
- If sockets are used to communicate processes across the network: namespace for internet addressing.
 - Identify host: IP address (32 bits)
 - Identify socket within the host: port number to which the socket is bound (16 bits). They can be well-known (associated with a service like ftp, web,...) or dynamically registered

Communication protocol

- Rules for transmitting information
- **TCP** (Transport Control Protocol)
 - Connection-oriented (stream)
 - 3 phases: establish connection / transfer data / close connection
- **UDP** (User Datagram Protocol)
 - Connectionless (datagram)
 - The application must implement reliability
- **Unix Local communication**
 - When sockets are used for local transmission

Client-server model

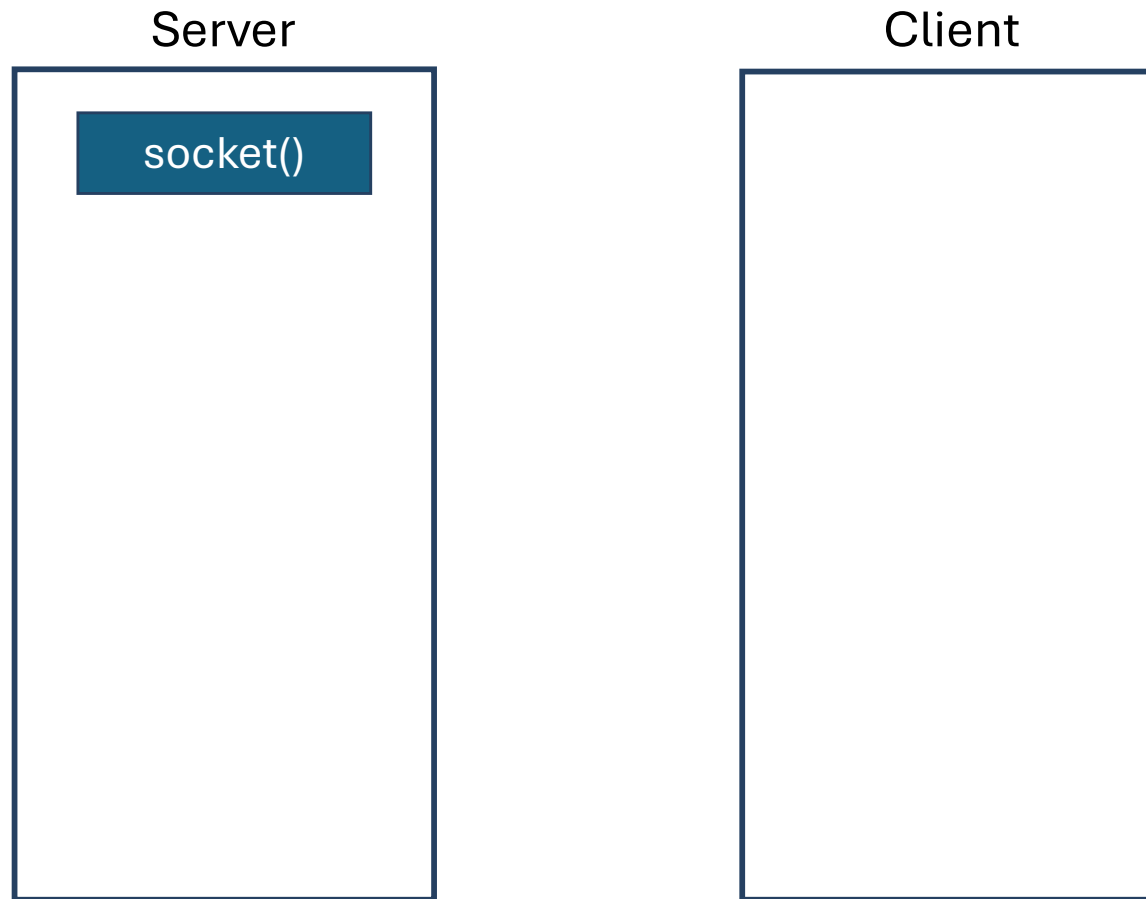
- **Server: manages access to a resource**
 - Sequential or iterative:
 - Receives request and serves it
 - Remaining clients have to wait
 - Concurrent:
 - Multiple threads or processes serving requests
 - Different thread creation schemes
- **Client: requests access to the resource**

Communication model

- One socket per process
 - Communication is full-duplex: both processes can read and write without needing to add synchronization to socket access
- When creating a socket, a channel is directly associated with it
- If you want the socket to be accessible by other processes, its address must be published
 - Stream communication: only the server needs to publish it
 - Datagram communication: both clients and the server must publish it
- Socket access
 - Stream communication: read and write
 - Datagram communication: sendto and recvfrom

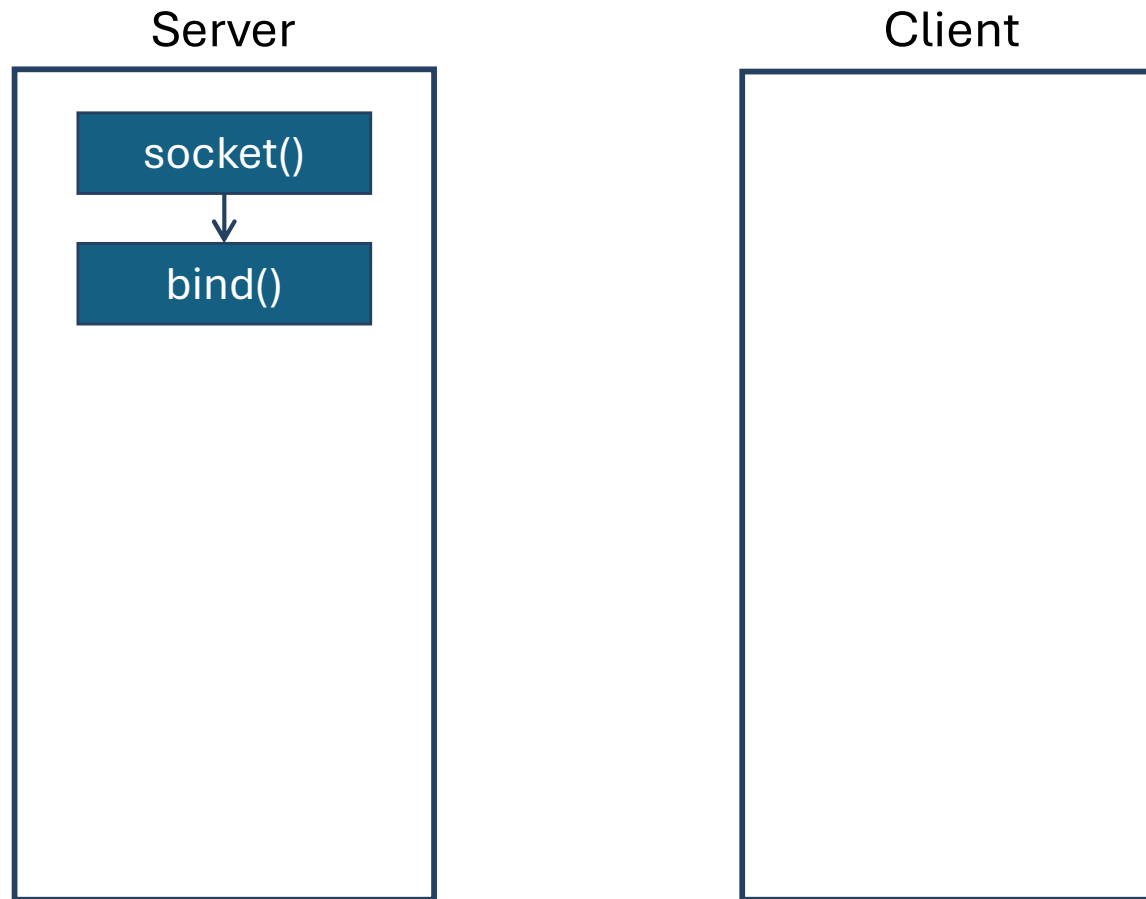
Creating a stream-based communication

- The server, first, creates a socket



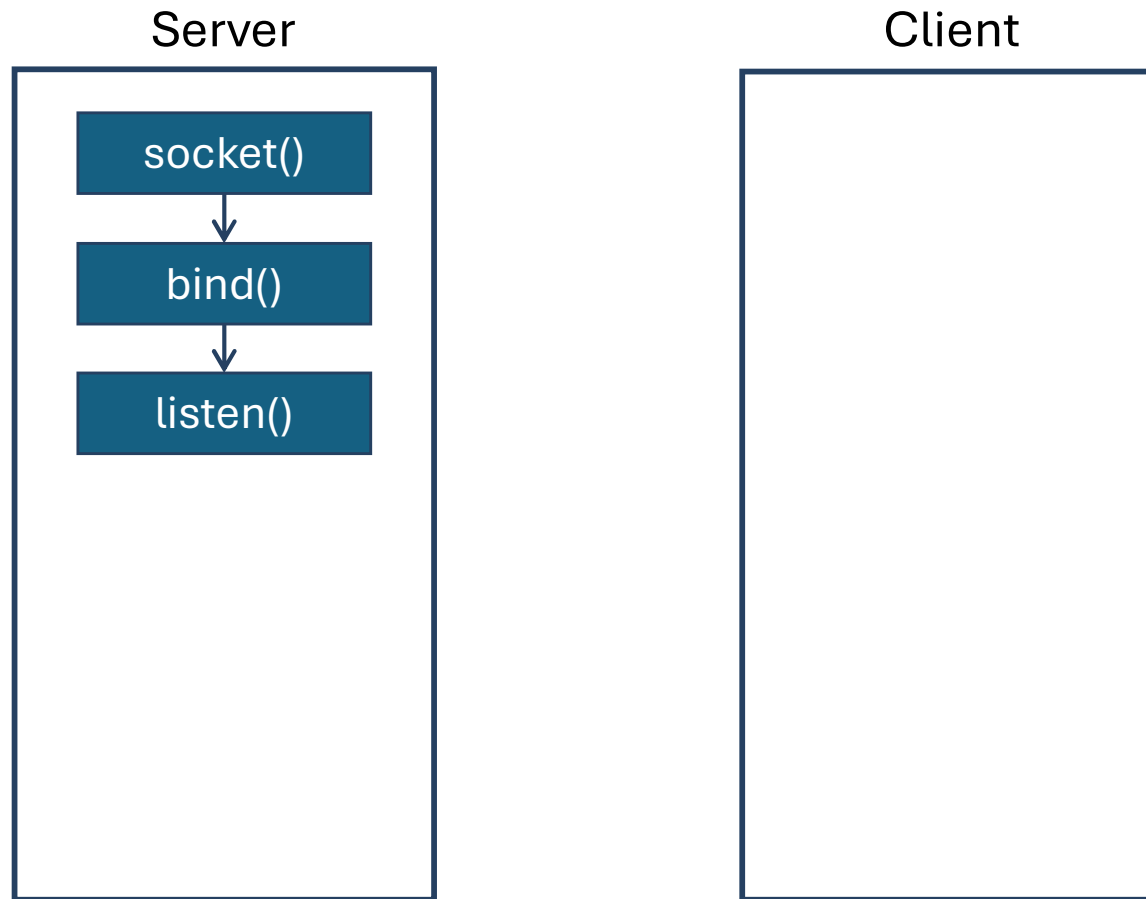
Creating a stream-based communication

- After that, the server publishes the socket. That means that assigns an IP+PORT to the socket



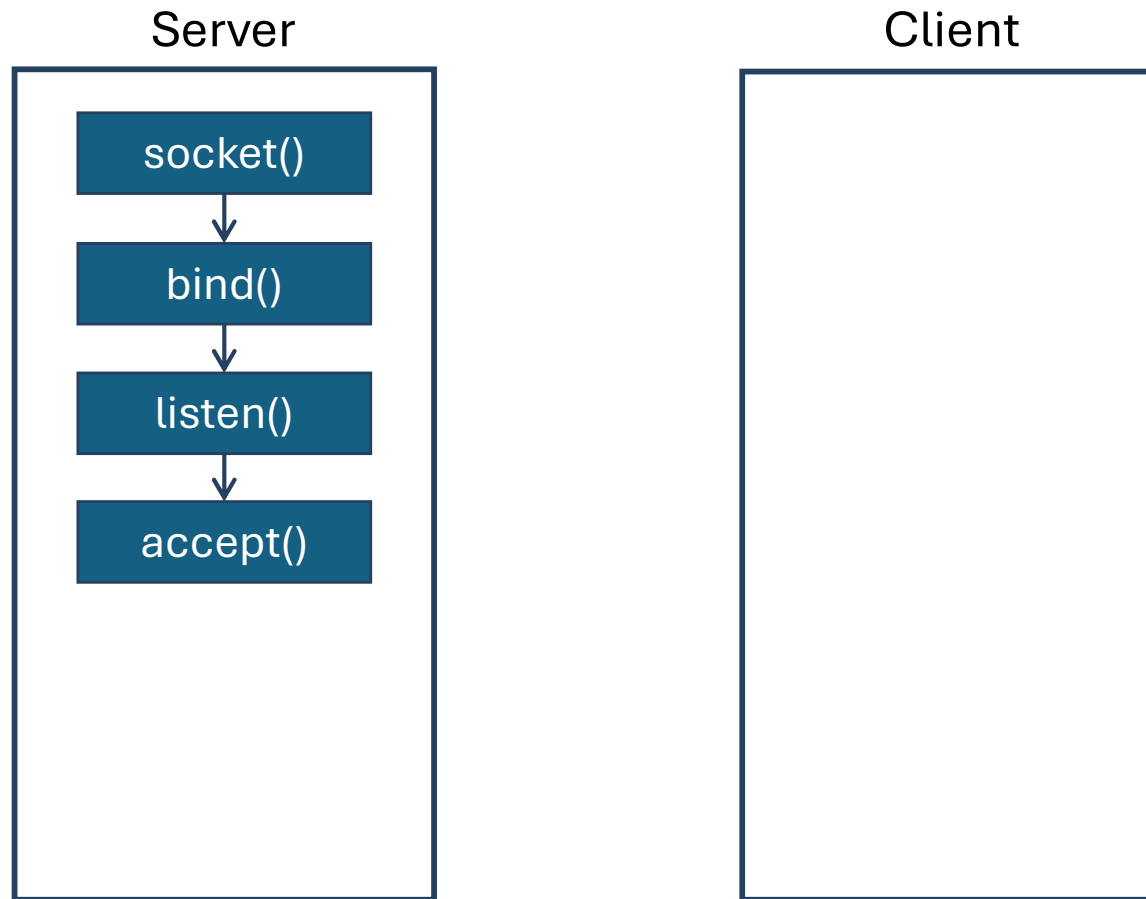
Creating a stream-based communication

- Once the socket is published, the server configures the socket



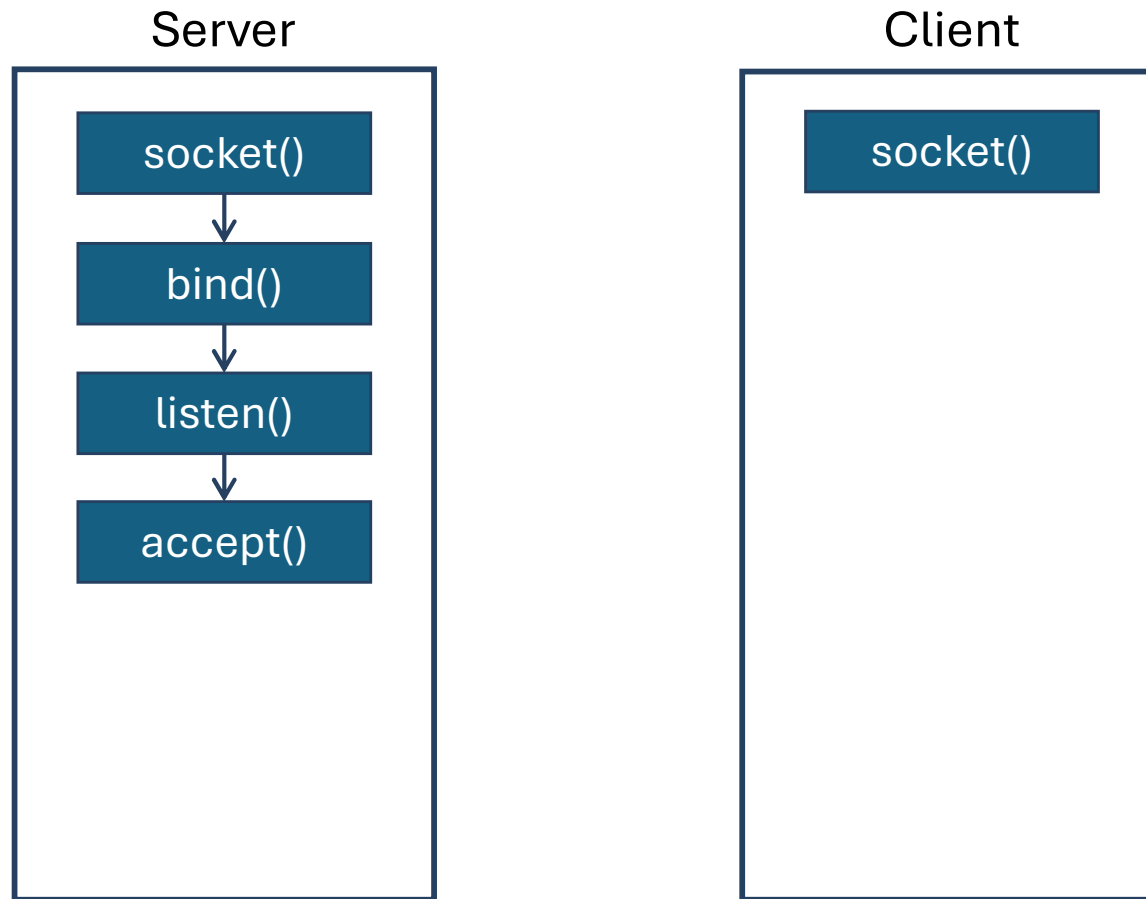
Creating a stream-based communication

- And the server waits for clients to connect to that socket using the IP+PORT address



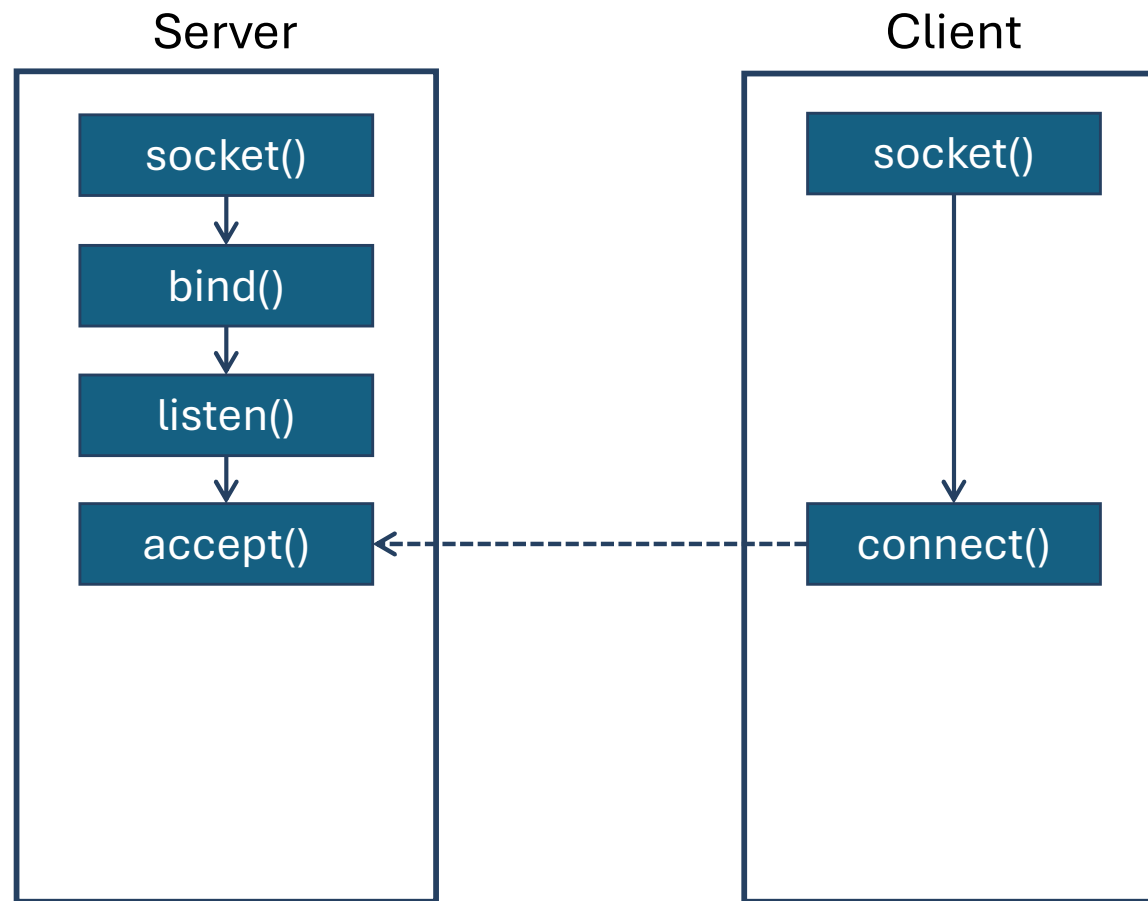
Creating a stream-based communication

- The client, as the server did, must create a socket to connect to the server



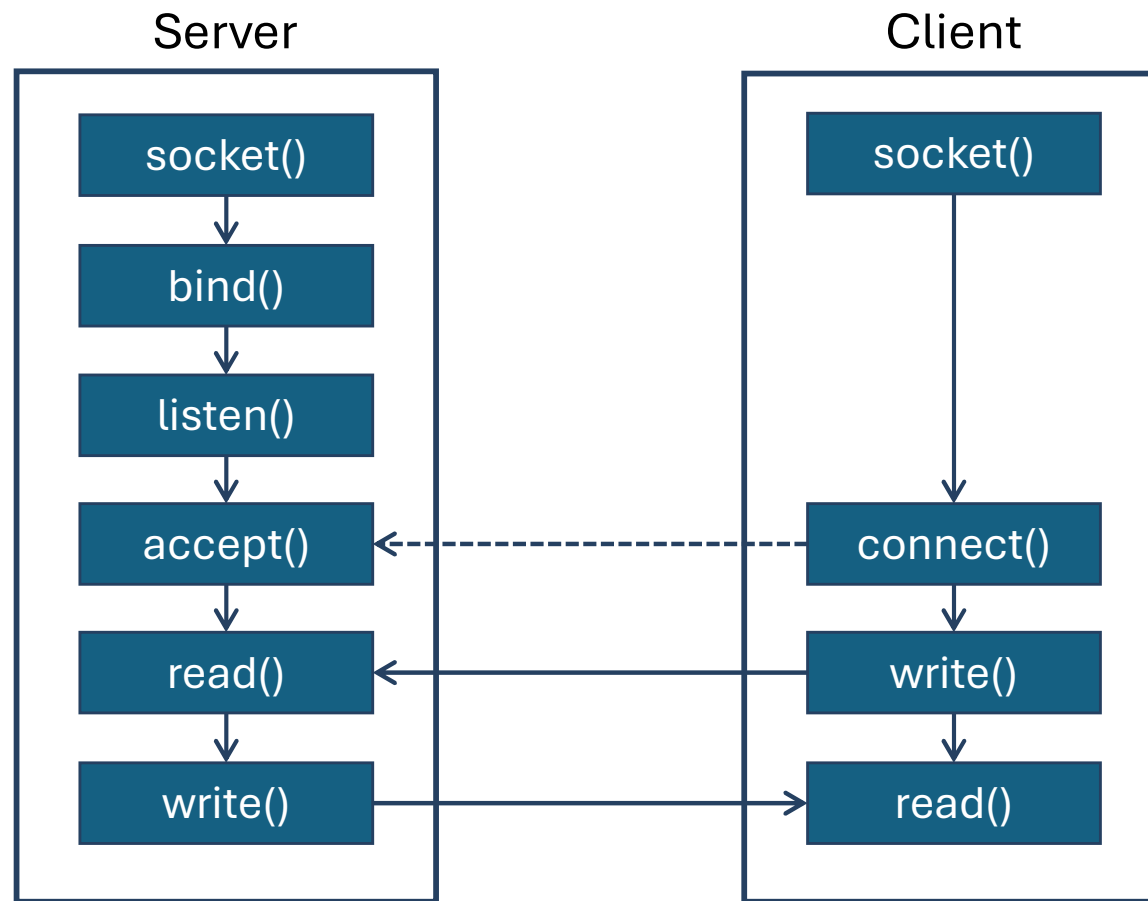
Creating a stream-based communication

- And the client connects to the IP+PORT the server published. The client must know that IP+PORT address.



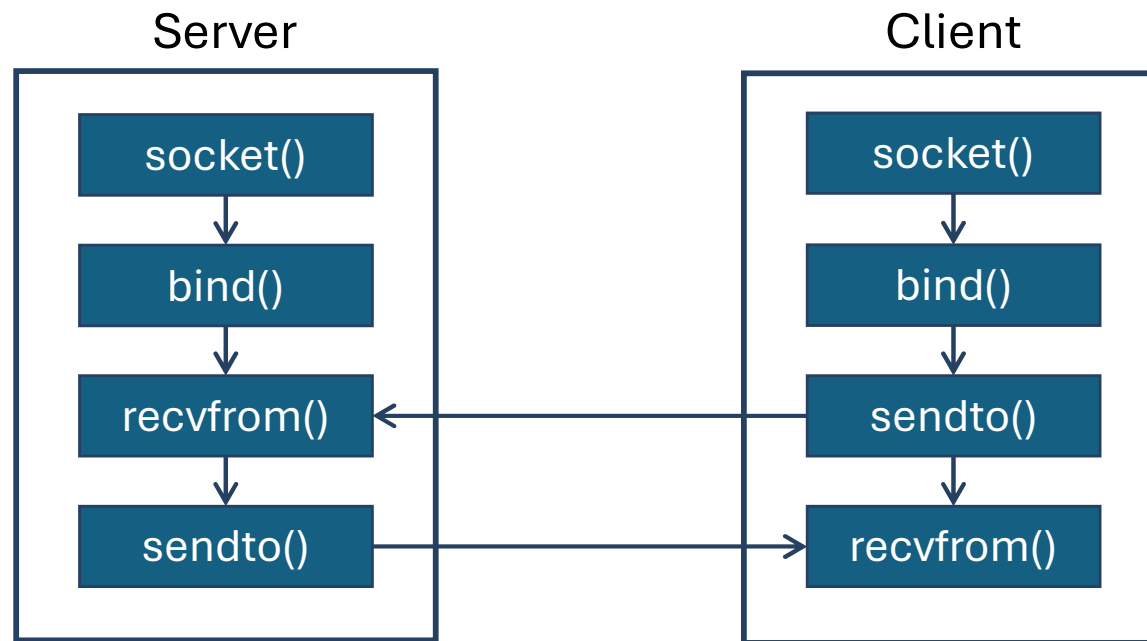
Creating a stream-based communication

- Once the connection is established, server and client can interchange information through the socket



Creating a datagram-based communication

- In datagram, both server and client must create and publish a socket. After that, they can interchange data.



socket syscall

- Creates a sockets and return the associated file descriptor

```
int socket(int af, int type, int protocol)
```

- af: family (namespace for the address)
 - PF_UNIX: unix socket
 - PF_INET: internet socket
- type: type of connection
 - SOCK_STREAM
 - SOCK_DGRAM
- protocol: if set to 0, the OS choses the correct one

Ports

- The same machine can offer several services to remote processes
- Every one of these services are identified with a public number: PORT
- For example: http uses port 80, telnet uses port 25...
- When a remote machine connects, it must indicate the IP of the machine to connect to and the connection PORT
- The OS must associate a port number with the connection socket
- This relationship <port, socket> is created with bind

Bind syscall

- Publishes the address of a socket

```
int bind(int fd, struct sockaddr* address, int address_size)
```

- fd: file descriptor of the socket
- address: address of the socket
 - AF_UNIX: a local file name
 - AF_INET: IP + PORT
- address_size: size, in bytes, of the address
- The OS stores, in an internal structure, the pair <address, socket>

Listen syscall

- Configure connection

```
int listen(int fd, int backlog)
```

- backlog: number of pending connection requests of a socket. If more than backlog requests arrive, the OS automatically rejects the connection.

Accept syscall

- Waits for a connection request

```
int accept(int fd, struct sockaddr* address, int address_size)
```

- fd: file descriptor of the socket
- address: address of the connected client
- address_size: size, in bytes, of the client address
- Returns a **new file descriptor** of the established connection
- Connections are accepted in FIFO order
- If no connections are pending, accept blocks the thread

Connect syscall

- Connection request

```
int connect(int fd, struct sockaddr* address, int address_size)
```

- fd: file descriptor of the socket
- address: address of the server to connect to
- address_size: size, in bytes, of the client address
- Blocks the thread until the server either rejects or accepts the connection

Reading/writing in a socket

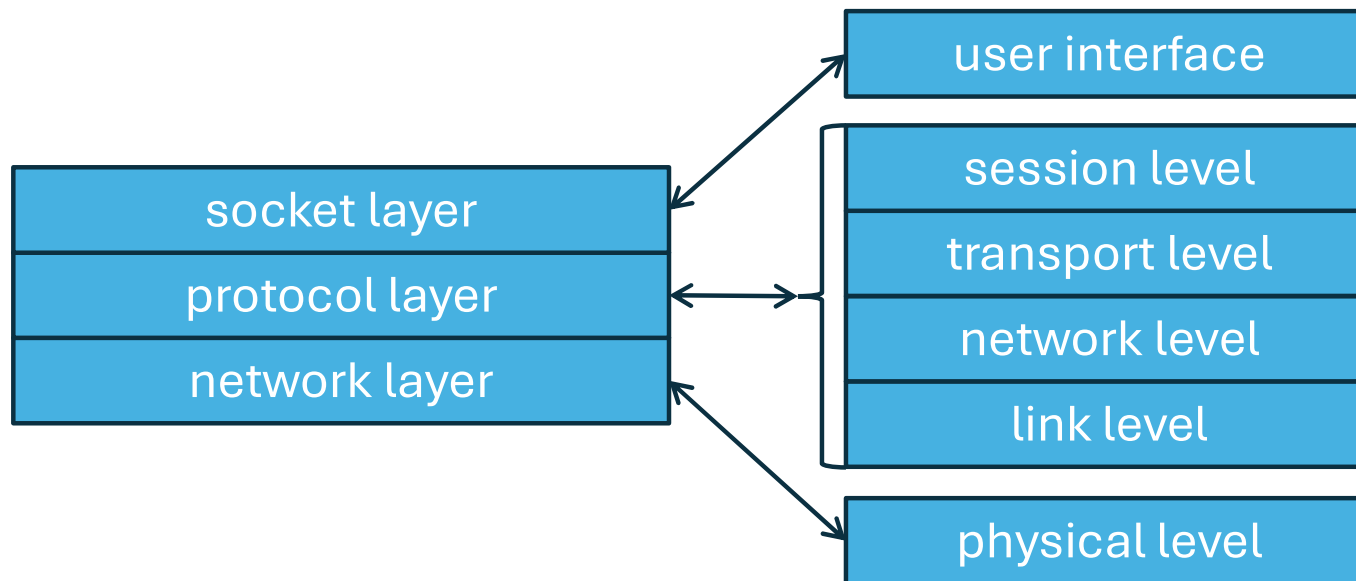
- For TCP:
 - Read/write
 - Remember to use the file descriptor returned by accept
- For UDP:
 - Recvfrom/sendto
 - Using directly the file descriptor returned by socket

```
int recvfrom(int fd, void *buffer, int length, int flags, void *from, int *fromlength);
```

```
int sendto(int fd, void *buffer, int length, int flags, void *to, int *tolength);
```

Socket: OS level

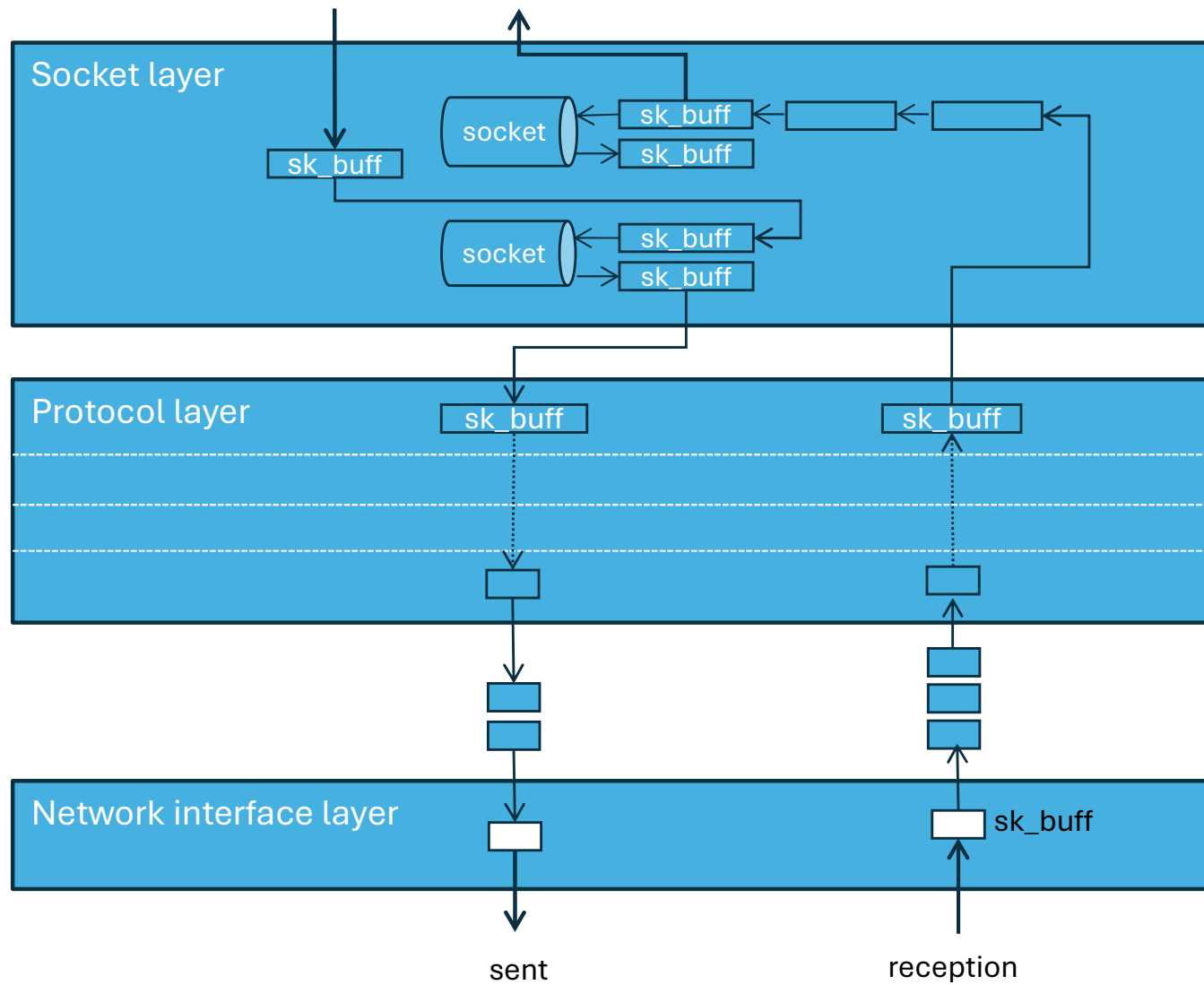
- Three layers responsible of the user interface, protocol management and network interface



Socket: OS level

- The communication between layers is performed using packet queues and software interrupts
- The struct `sk_buff` is the structure used to store a packet
- Every socket has two queues:
 - Received packets
 - Sent packets

Socket: OS level



Packet reception

- Interface management
 - Management of the network card
 - Interruption management: packet reception
 - Message decodification
 - Creates a struct `sk_buff` to store the received message
 - Packets are passed to the upper layer using message queues
 - Uses a software interrupt to notify the upper layer

Packet reception

- Protocol management:
 - There is one layer for supported protocol
 - Routing: select packets of the host and routes the others
 - For every protocol layer, a header is added
 - Determines the target socket
 - Passes packets to the upper layer: there is one queue per socket
 - Notifies the upper layer using a software interrupt

Packet reception

- Socket management:
 - When reading from the socket
 - Access the socket queue to select the message
 - Copies to user memory the received information
 - If any thread is blocked reading from the socket, it will be unblocked

Sending a packet

- Socket management
 - When writing the socket
 - A new struct `sk_buff` is created
 - Copies from user memory to the `sk_buff` the information
 - Passing packets to the lower layer: packet queue per socket
- Protocol management
 - There is one per supported protocol
 - For every protocol layer a header is added
 - Routing: determines the destination address
 - Uses a queue to pass the buffers to the lower layer
- Interface layer
 - Network card management
 - Codifies the message and programs the network card

Connection establishment

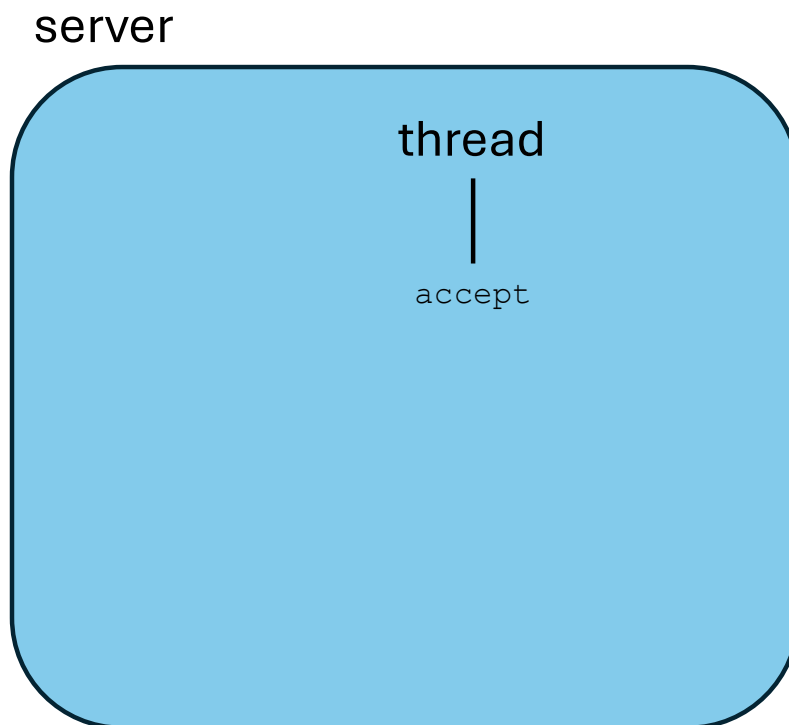
- Protocol management
 - When a connection request is received, enqueues the corresponding `sk_buff` to the target socket
 - If the thread is blocked in an `accept`, it is unblocked
 - If the thread is not blocked in an `accept`, a pre-`accept` is performed
 - Returns to the client a `syn-ack` packet
 - When the client responds with the `ack`, messages can be sent
- Socket management
 - When the server executes `accept`
 - If there is any enqueued connection requests, it is processed and the server continues execution
 - If there is not a connection request, the server is blocked

Concurrency models

- Server must handle thousands of concurrent requests
- As hardware scaling shifts from single-core speeds to massive parallelism, how concurrent execution is performed determines whether systems thrive or collapse under load.
- Orchestration, multithreading, memory and socket connections must be managed wisely to achieve high-performance
- Several concurrency models are used

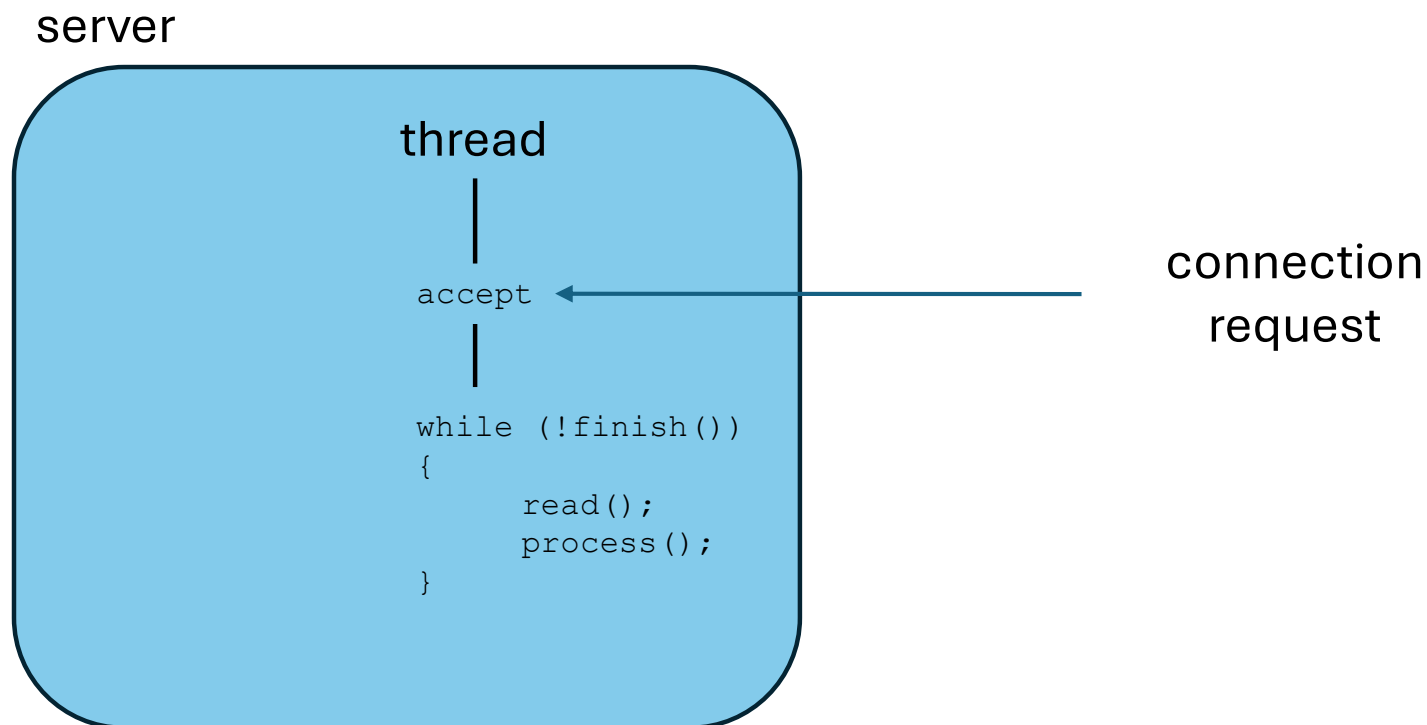
Basic connection processing

- The server has an open public socket and one thread is waiting for a connection



Basic connection processing

- When a new connection request is detected, the thread unblocks and keeps on executing in a loop processing the data of the connection



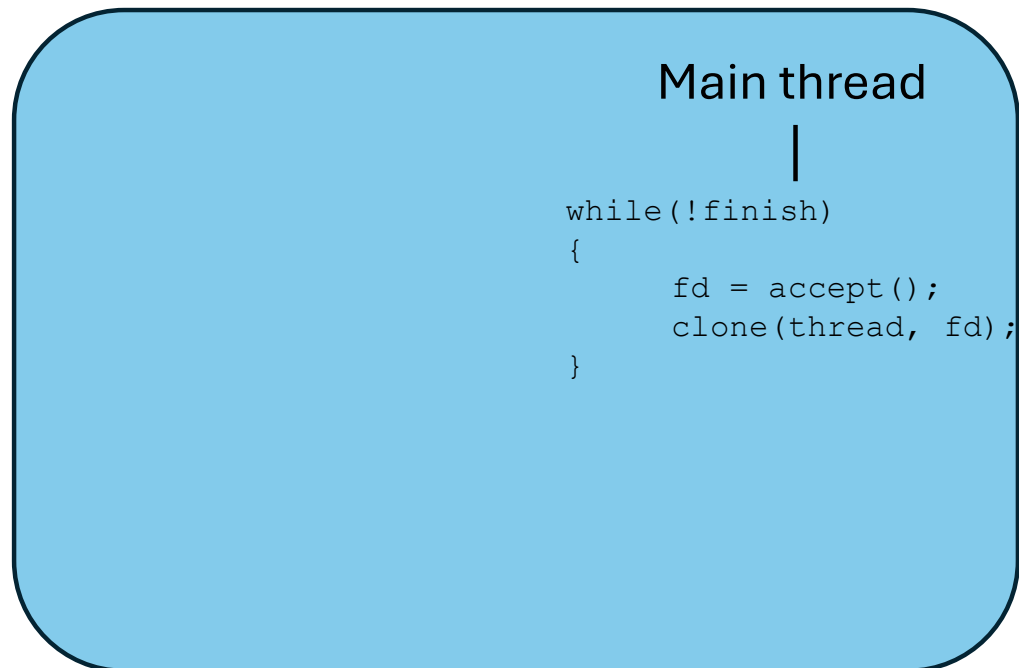
Basic connection processing

- Pros:
 - Very simple to implement
 - Few bytes of information per connection
- Cons:
 - The server can process just a single connection
 - Petitions per connection are processed sequentially
- Improvement: spawn one thread per connection

Spawn a thread per connection

- The server has one single thread that loops accepting connections

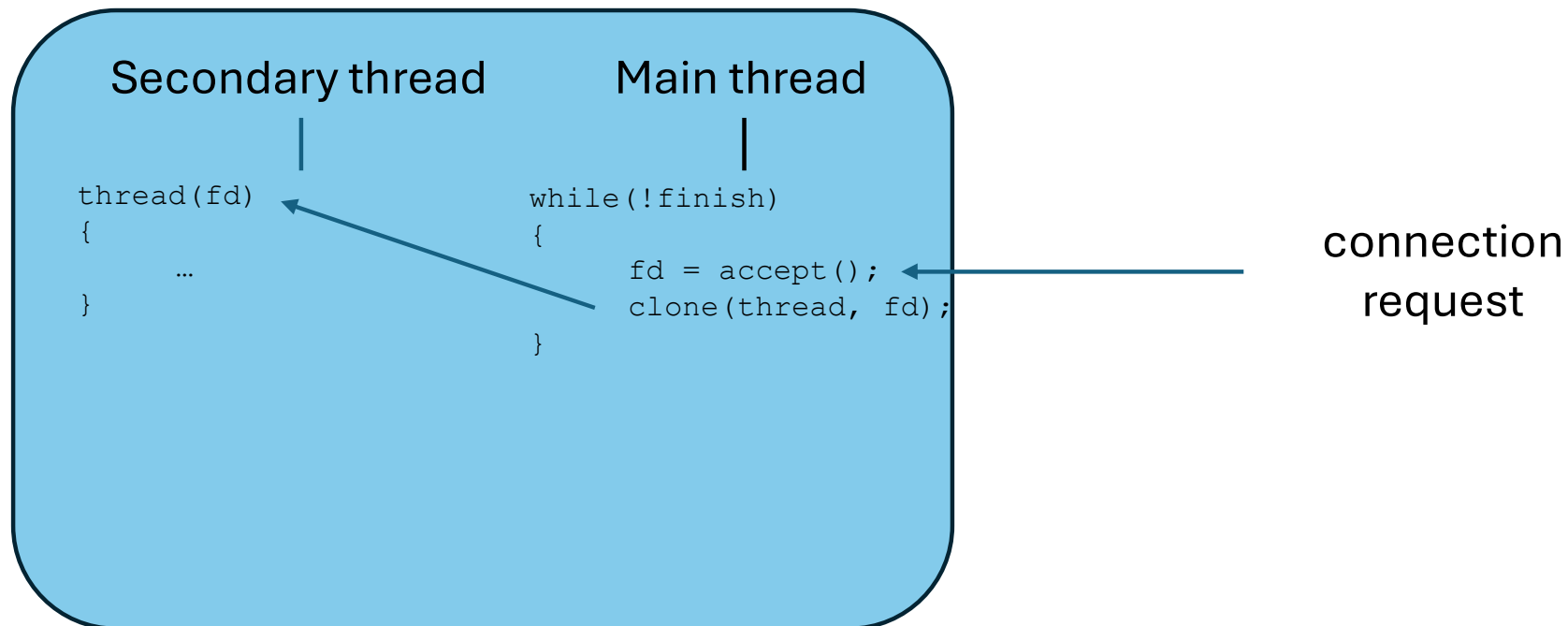
server



Spawn a thread per connection

- When a new connection request is accepted, a new thread is spawned having as argument, the file descriptor of the connection

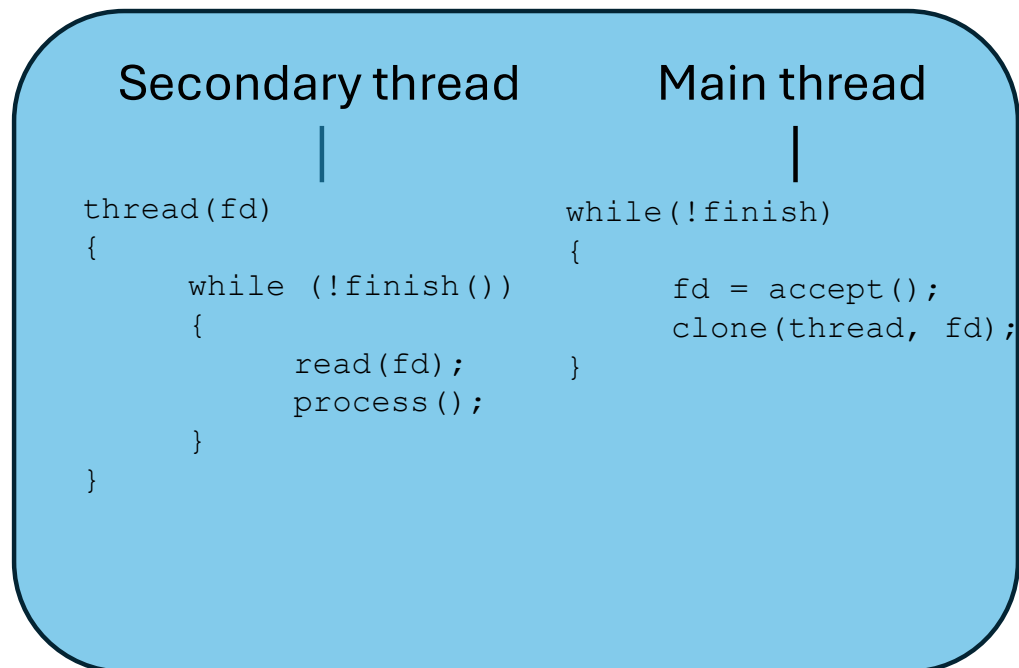
server



Spawn a thread per connection

- This second thread processes the connection in parallel with the first thread who is accepting more connections and spawning new threads

server



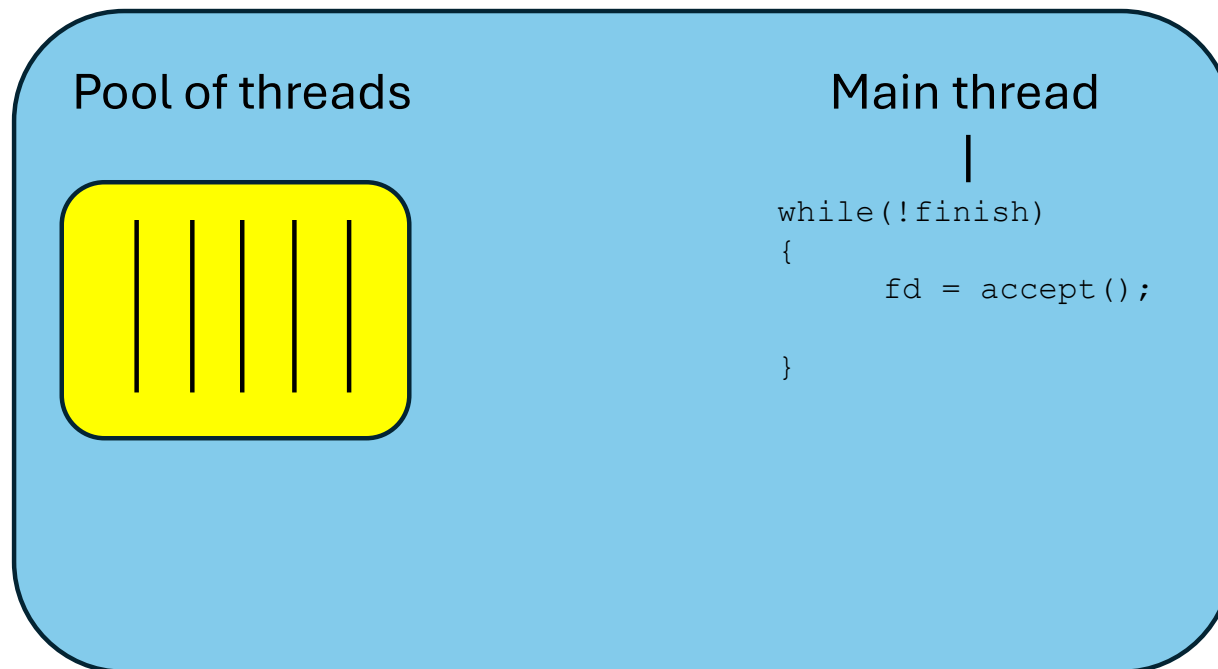
Spawn a thread per connection

- Pros:
 - Multiple connections can be processed in parallel
 - Simple to implement
 - Few bytes of information per connection
- Cons:
 - It is not easy to free (`pthread_join`) the resources allocated by the thread (TCB) when the processing finishes
 - When multiple connections are processed, a lot of context switches are provoked
 - Unbalanced work in connections leads to allocated unused resources in threads
- Improvement: static pool of threads

Pool of threads

- When the server starts, before creating the socket, spawns a pool of threads to process connections.

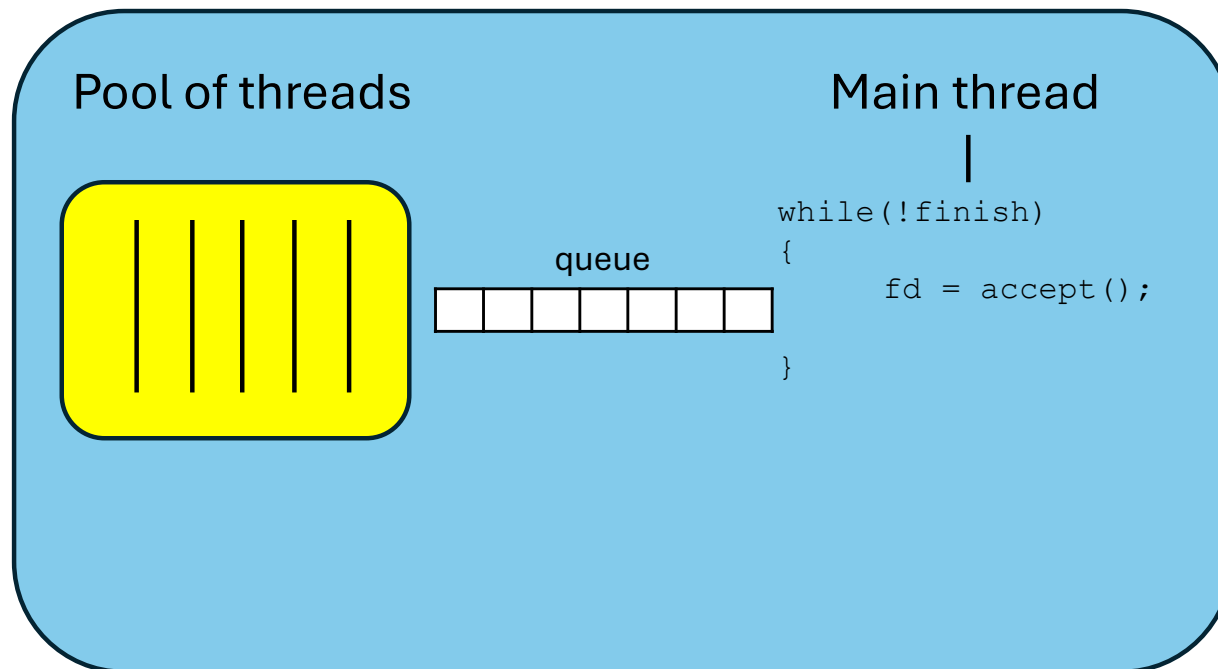
server



Pool of threads

- Also, a shared queue between the main thread and the pool of threads is allocated.

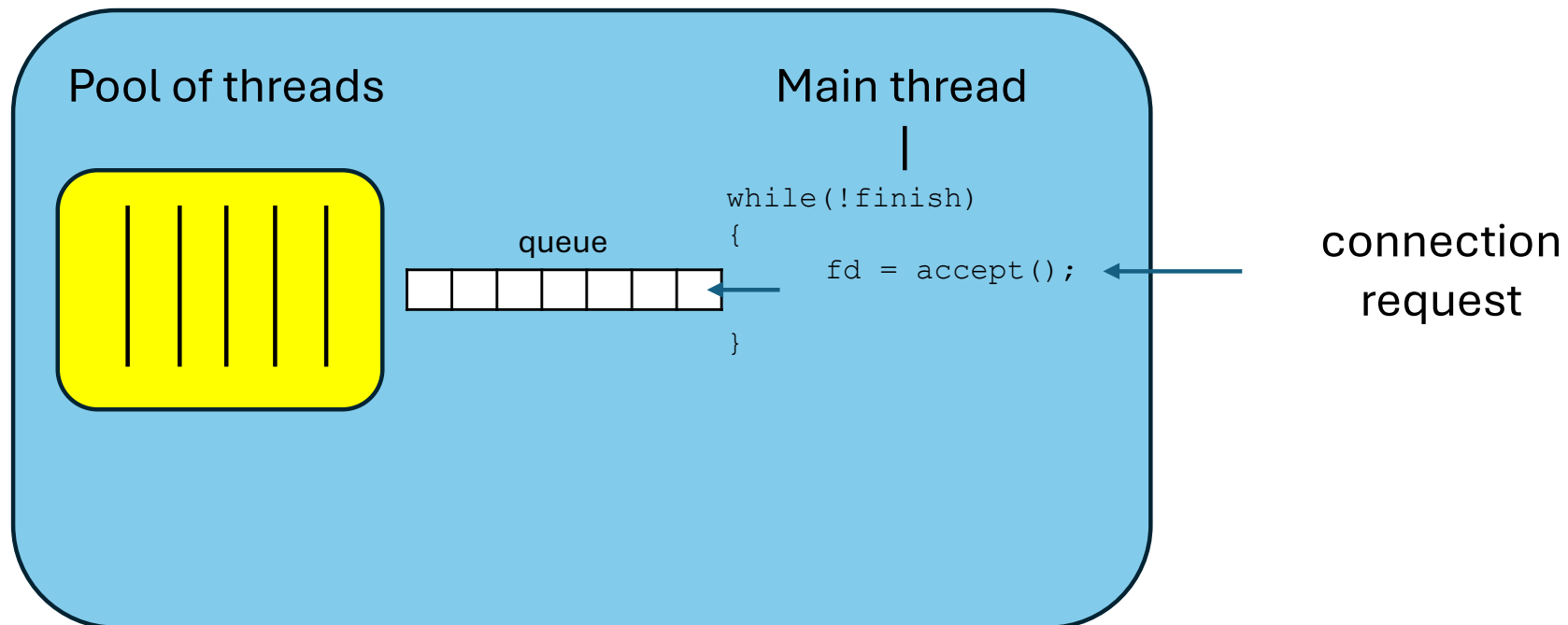
server



Pool of threads

- When a connection is accepted, the file descriptor is stored in the shared queue.

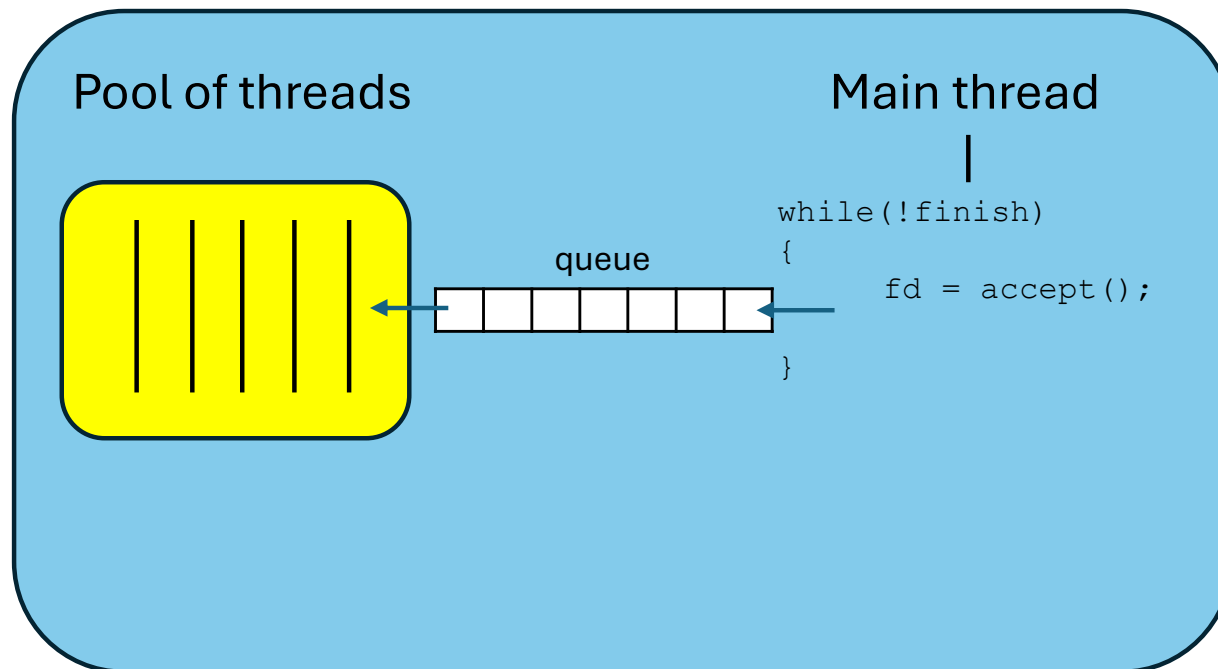
server



Pool of threads

- Threads in the pool of threads fight to get a file descriptor from this queue. When acquired, that thread will process the whole connection.

server



Pool of threads

- Pros:
 - Since threads do not exit (`pthread_exit`) resources must not be deallocated
 - The number of context switches are significantly reduced
- Cons:
 - The number of processed connections in parallel is limited by the number of threads in the pool
 - Even if the number of context switches have been reduced, it is still significant
 - Petitions in a connection are served by just one thread (the one that processes the connection) and not shared among threads
- Improvement: Single-threaded event loop

Single-threaded event loop

- Now, only one thread will process all connections
- For that, the thread will monitor the status of a set of file descriptors at the same time
- If the status of any of the file descriptors in the set changes, the thread will be notified
- So, the thread will remain blocked until some process of those file descriptors is needed
- Implementation: epoll

epoll (Event poll)

- **epoll** (Event Poll) is a Linux kernel system call designed for high-performance I/O event notification.
- It is the core building block used by high-throughput web servers and runtimes (like NGINX, Node.js, and Redis) to implement non-blocking event loops and handle tens of thousands of simultaneous connections

epoll kernel implementation

- The kernel maintains an in-memory data structure (a **red-black tree**) containing the active set of file descriptors to monitor.
- When a socket receives data, the network hardware interrupt places that ready socket into a **ready list** (a doubly-linked list).
- The server process checks this ready list in $O(1)$ **constant time**—no matter how many total sockets are currently idle.
- There are 3 system calls involved: `epoll_create`, `epoll_ctl` and `epoll_wait`

Epoll_create

- Initializes an epoll instance in the kernel and returns a file descriptor pointing to it

```
int epoll_create1(int flags)
```

- flags must be 0

Epoll_ctl

- Adds, modifies or remove file descriptors from the watched interest list

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
```

- epfd: epoll file descriptor
- op: can be one of
 - EPOLL_CTL_ADD: adds a file descriptor to the epoll
 - EPOLL_CTL_MOD: modified one file descriptor in the epoll
 - EPOLL_CTL_DEL: deletes one file descriptor from the poll
- fd: file descriptor to add/modify/remove
- event: event that forces the notification about the file descriptor

Epoll_ctl event parameter

- The event parameter of the `epoll_ctl` system call defines when the notification for a file descriptor must be raised

```
struct epoll_event ev;  
ev.events = EPOLLIN; // Monitor for incoming read data  
ev.data.fd = socket_fd;  
// Register socket_fd with the epoll instance  
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, socket_fd, &ev);
```

- Common events
 - EPOLLIN: read ready (level-triggered mode)
 - EPOLLOUT: write ready (level-triggered mode)
 - EPOLLERR: error
 - EPOLLET: force edge-triggered mode

epoll_ctl level-triggered vs. edge-triggered

- When configuring `epoll_ctl`, how the notifications are delivered must be configured

Trigger Mode	Behavior	Programming Pattern
Level-Triggered (Default)	<code>epoll_wait()</code> will repeatedly notify as long as unread data remains in the socket buffer.	Simple; forgiving if don't read all available bytes in one pass.
Edge-Triggered (EPOLLET)	<code>epoll_wait()</code> notifies only once when new data arrives (a state transition from no data to data available).	High performance; requires non-blocking sockets and reading in a while loop until <code>EAGAIN</code> or <code>EWOULDBLOCK</code> is returned.

Epoll_wait

- Blocks the calling thread until at least, one registered file descriptor becomes ready for I/O or until a timeout expires

```
int epoll_wait(int epfd, struct epoll_event events[n], int n, int timeout)
```

- epfd: epoll file descriptor
- events: ready list of file descriptors in the interest list
- n: maximum number of events to return
- timeout: number of milliseconds that epoll_wait will block

Standard event loop workflow

- To construct a server using epoll, follow this sequence:
- Create and bind a master listening socket, set it to non-blocking mode via `fcntl()`.
- Create the epoll instance using `epoll_create1()`.
- Add the listening socket to epoll using `epoll_ctl()` with `EPOLLIN`.
- Enter the Event Loop:
 - Call `epoll_wait()` to get ready sockets.
 - Iterate over returned events:
 - If event is on the listening socket: Call `accept()` to get the new client socket, set client socket to non-blocking, and register it with `epoll_ctl()`.
 - If event is on a client socket: Perform non-blocking `read()` or `write()` operations.
 - If client disconnects or errors: Unregister via `epoll_ctl(..., EPOLL_CTL_DEL, ...)` and `close()` the socket.

Standard event loop workflow

```
listen_fd = socket();
bind (listen_fd);
set_nonblocking(listen_fd);
listen(listen_fd);
epoll_fd = epoll_create(1);
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd);
while(1)
{
    int nfds = epoll_wait(epoll_fd, events);
    for (int i = 0; i < nfds; i++)
    {
        int fd = events[i].data.fd;
        if (fd == listen_fd)
        {
            new_fd=accept(listen_fd);
            set_nonblocking(new_fd);
            epoll_ctl(epoll_fd, EPOLL_CTL_ADD, new_fd);
        }
        else
            // process connection
    }
}
```

Single-threaded event loop

- Pros:
 - One single thread processes all connections (no context switches)
- Cons:
 - Difficult to implement
 - Not able to exploit all available hardware parallelism
- Improvement: hybrid model

Hybrid model

- If hardware can execute more than one thread in parallel (multithreaded/multicore processors) the single-threaded event loop is not able to exploit it
- Solution: execute a single-threaded event loop per core of the CPU
- For this:
 - The set of socket/connection file descriptor must be shared among threads
 - The state of every connection must be stored and shared among all threads

Model comparison matrix

Model	Memory per Connection	Context Switch	Scalability Limit	Ideal Use Case
Multi-Threaded	~Hundreds of KB	OS-level (Medium)	~Thousands	Enterprise Java apps, blocking DB workloads
Event-Driven	~Kilobytes	User-space (Light)	~Hundreds of Thousands	High-concurrency I/O (Nginx, Node.js)
Hybrid	~Kilobytes	Mixed	Scales across CPU cores	Modern web gateways, reverse proxies