Operating Systems & C

Lecture 8: I/O

Willard Rafnsson
IT University of Copenhagen

ITU CPH

Topics

- File Systems
  - implementation, conceptually
  - implementation, in Linux (VFS)
  - virtual file systems
- I/O APIs
  - File
  - Socket
  - Message-Passing Interface (MPI)

ITU CPH

File Systems

implementation, conceptually

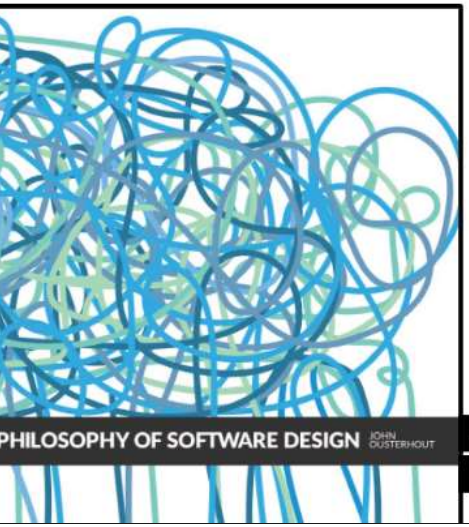
ITU CPH

Files

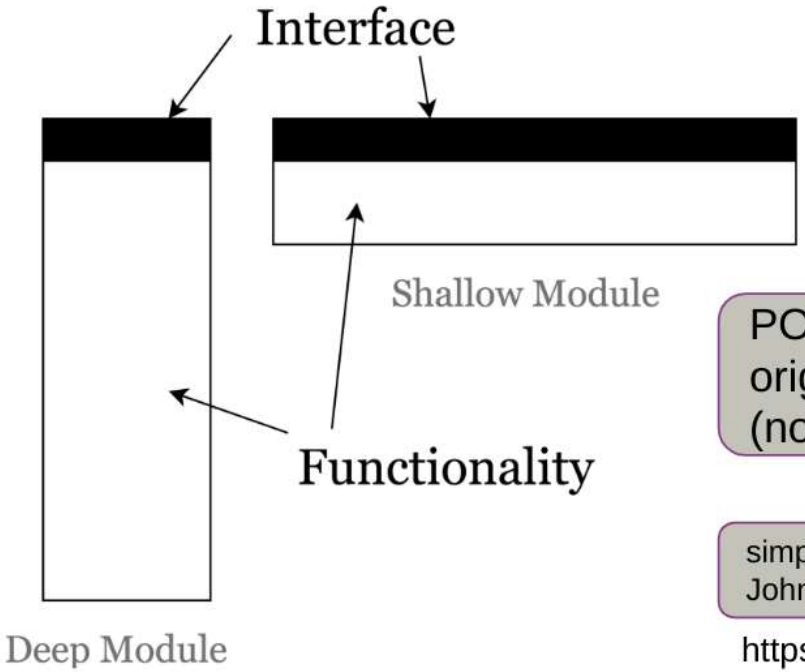
like memory!

A file is an array of bytes

File interface: create/delete, open/close, read/write



HAGEN



POSIX: unified unix specification.
original POSIX interface is beautiful.
(not part of C language, but std lib)

simple interface, hiding a lot of complexity.
John Ousterhout, professor at Stanford.

https://www.youtube.com/watch?v=bmSAYlu0N

11. ITU CPH

File System, Conceptual Overview

- path, hierarchy of
- directory, collection of
- filename, user-friendly name of a
- inode, identifies a (& holds metadata on)
- file, collection of
- block

let's start with blocks.

ITU CPH

Block Layer

we start with:

30s

A **block device** is an **array of blocks**.
To each block is associated a number,
a Logical Block Address (LBA)

sound familiar?
virtual memory! pages!
(quantized data)
block is a unit of transfer.
(associativity layer maps block number to actual block)

```
procedure BLOCK_NUMBER_TO_BLOCK (integer b) returns block
    return device[b]
```

hard disk drives today
(incl. SSDs) are block devices.

UNIVERSITY OF COPENHAGEN

ITU CPH

File Layer

How to represent files?
Each **file** is a **collection of disk blocks**
(more abstractly (haha), an **array of bytes**)

```
structure inode {
        integer block_numbers[N];   // the numbers of the blocks that constitute the file
        integer size;               // the size of the file in bytes
}
```

inode ("index node") is a collection o
block numbers (associated to the
file), and their collective size.
(we need this level of indirection)

```
procedure INDEX_TO_BLOCK_NUMBER (instance of inode i, integer index) returns integer{
        return i.block_numbers[index];
}

procedure INODE_TO_BLOCK ( integer offset, instance of inode i) returns instance of block
        o ← offset / BLOCKSIZE;
        b ← INDEX_TO_BLOCK_NUMBER(inode, o);
        return BLOCK_NUMBER_TO_BLOCK(b);
}
```

Inode Name Layer

File system state
inode_table

How to avoid carrying inodes around?

level of indirection

number the inodes!
inode_number to inode table (map),
carry this table around.

```
procedure INODE_NUMBER_TO_INODE(integer inode_number) returns instance of inode{
        return inode_table[inode_number];
}
```

```
procedure INODE_NUMBER_TO_BLOCK (integer offset, integer inode_number)
                                                    returns instance of block {
        structure inode i ← INODE_NUMBER_TO_INODE (inode_number);
        o ← offset / BLOCKSIZE;
        b ← INDEX_TO_BLOCK_NUMBER (i, o);
        return BLOCK_NUMBER_TO_BLOCK (b);
}
```

```
return INODE_TO_BLOCK(offset,i);
```

File Name Layer

File system state
inode_table

Representing directories

```
structure inode{
        integer block_numbers[N];   // the numbers of the blocks that constitute the file
        integer size;               // the size of the file in bytes
        integer type;               // type of file: regular file, directory,...
}
```

directory represented as inode.
(now have 2 types of inodes).

directory contents represented:
each block stores inode nums

User-friendly names

File name	Inode number
program	10
Paper	12

when you work with files, you don't work with inode numbers.
you work with filenames.
need mapping from filename to inode number.

in dir, we store,
alongside an inode number,
the *filename* of that inode

UNIVERSITY OF COPENHAGEN

ITU CPH

File Name Layer

File system state
inode_table

Directory lookup

```
procedure NAME_TO_INODE_NUMBER (character string filename, integer dir) returns integer {
        return LOOKUP (filename, dir);
}


procedure LOOKUP (character string filename, integer dir) returns integer{
        instance of block b;
         instance of inode i ← INODE_NUMBER_TO_INODE (dir);
        if i.type ≠ DIRECTORY then return FAILURE;
        for offset from 0 to i.size – 1 do {
                b ← INODE_NUMBER_TO_BLOCK (offset, dir);
                if STRING_MATCH (filename, b) then {
                        return INODE_NUMBER (filename, b);// return inode number for filename
                }
                offset ← offset + BLOCKSIZE;          // increase offset by block size
        }
        return FAILURE;
}
```

(inode number)

if filename occurs in b,

then return the inode num that's written next to the filename

`STRING_MATCH, INODE_NUMBER` implementation not shown

UNIVERSITY OF COPENHAGEN

ITU CPH

Path Name Layer

File system state
inode_table

Hierarchy of Directories

```
procedure PATH_TO_INODE_NUMBER (character string path, integer dir) returns integer{
        if (PLAIN_NAME (path)) return NAME_TO_INODE_NUMBER (path, dir);
        else {
                dir ← LOOKUP (FIRST (path), dir);
                path ← REST (path);
                return PATH_TO_INODE_NUMBER (path, dir);
        }
}
```

UNIVERSITY OF COPENHAGEN

ITU CPH

Absolute Path Name Layer

File system state
inode_table
Process state:
wd

Change working directory

```
procedure CHDIR (path character string) { wd ← PATH_TO_INODE_NUMBER (path, wd); }
```

How to name a file regardless of the current working directory?

```
procedure GENERALPATH_TO_INODE_NUMBER (character string path) returns integer{
        if (path[0] = "/") return PATH_TO_INODE_NUMBER(path, 1);
        else return PATH_TO_INODE_NUMBER(path, wd);
}
```

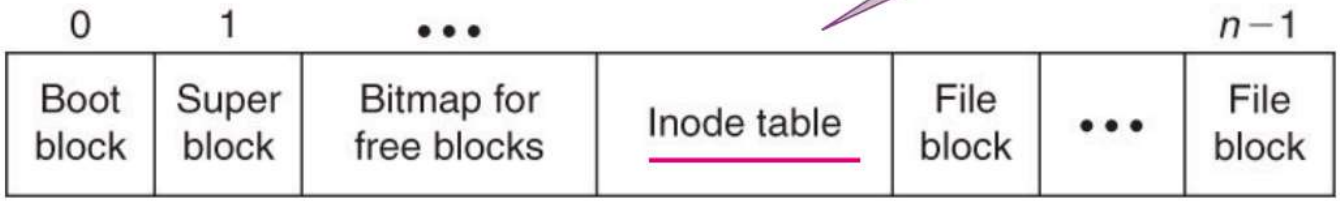
(root inode number)

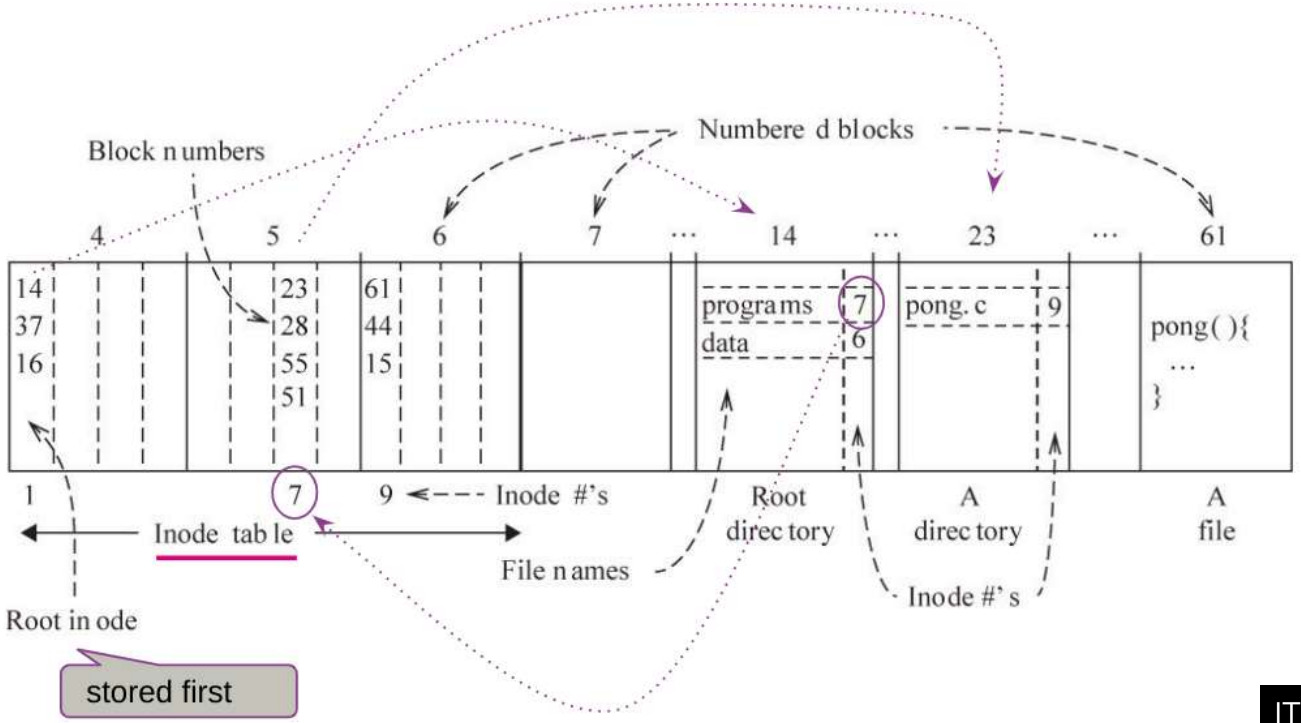
Unix File System Naming Scheme

File system state
inode_table
Process state:
wd

Disk Layout for a file system

must be persistent, else won't know which files are there





UNIVERSITY OF COPENHAGEN

ITU CPH

Symbolic Link Layer

we can have multiple paths to the same inode.

How about flexible management of files?

```
LINK (from_name, to_name);
UNLINK (from_name);
```

```
structure inode{
        integer block_numbers[N];
        integer size;
        integer type;
        integer refcnt;
}
```

once no path refers to inode it can be garbage collected.

Symbolic Link Layer

in Linux, you can mount a file system

10s

How to attach new disks to a file system?

```
MOUNT ("/dev/fd1", "/floppy")
```

(block) device
(should contain file system)

mount point
(where root of mounted file system is to be accessible)

"In UNIX, Everything is a File"
block devices (e.g. discovered during bootup, or e.g. when you plug in a USB) represented as files in the root file system on b

Naming Layers in Unix File System

10s

Layer	Names	Values	Context	Name-mapping algorithm	
Symbolic link	Path names	Path names	The directory hierarchy	PATHNAME_TO_GENERAL_PATH	↑ user-oriented names
Absolute path name	Absolute path names	Inode numbers	The root directory	GENERALPATH_TO_INODE_NUMBER	
Path name	Relative path names	Inode numbers	The working directory	PATH_TO_INODE_NUMBER	↓
File name	File names	Inode numbers	A directory	NAME_TO_INODE_NUMBER	machine-user interface
Inode number	Inode numbers	Inodes	The inode table	INODE_NUMBER_TO_INODE	↑ machine-oriented names
File	Index numbers	Block numbers	An inode	INDEX_TO_BLOCK_NUMBER	
Block	Block numbers	Blocks	The disk drive	BLOCK_NUMBER_TO_BLOCK	↓

UNIVERSITY OF COPENHAGEN

Source: Saltzer and K[illegible]e

ITU CPH

API: State

File system state
- inode_table
- file_table

Process state:
- fd_table
- wd

Which files are in use?
`file_table`

File name	Inode number	cursor
program	10	64
Paper	12	0

Cursor is the first byte that will be accessed by the next read or write operation.

Which files is each process using?
`fd_table`

Mapping from file descriptors into the **file_table**.
(file descriptors are per-process. natural numbers; 0 is stdin, 1 is stdout, 2 is stderr, …)

Multiple processes can have a file open with different cursors, and
Multiple processes can have a file open sharing a cursor (fork; **fd_table** shared)

API: inode

```
structure inode {
        integer block_numbers[N];   // the number of blocks that constitute the file
        integer size;               // the size of the file in bytes
        integer type;               // type of file: regular file, directory, symbolic link
        integer refcnt;             // count of the number of names for this inode
        integer userid;             // the user ID that owns this inode
        integer groupid;            // the group ID that owns this inode
        integer mode;               // inode's permissions
        integer atime;              // time of last access (READ, WRITE,...)
        integer mtime;              // time of last modification
        integer ctime;              // time of last change of inode
```

we did not talk abou
e.g. access control
(but we will!)

I/O API

File

ITU CPH

I/O API : File

recall: create/delete, open/close, read/write, link/unlink.

in Linux:

```c
int creat  ( const char *pathname, mode_t mode);
int remove /* calls `unlink` for files, `rmdir` for directories (refcnt...)

int link   ( const char *oldpath, const char *newpath);
int unlink ( const char *pathname);

int open   ( const char *pathname, int flags, ... /* mode_t mode */ );
int close  ( int fd);

ssize_t read  ( int fd, void buf[.count], size_t count);
ssize_t write ( int fd, const void buf[.count], size_t count);
```

with conceptual details above, you can imagine how these work; see man-pages for details.

will show open and read (conceptually) shortly.

ITU CPH

Linux I/O System Calls

- creat, open, read, write, close, lseek
- fsync
- link, unlink
- stat, lstat, fstat
- access, umask, chmod, chown, utime
- ioctl

libraries for this

there are also *async I/O* system calls. (e.g. aio_read)
and ways to batch system calls (io_submit, …)

API Calls: Open

(skippable

File system state
- inode_table
- file_table

Process state:
- fd_table
- wd

```
procedure OPEN (character string filename, flags, mode) {
        inode_number ← PATH_TO_INODE_NUMBER (filename, wd);
        if inode_number = FAILURE and flags = O_CREATE then { // Create the file?
                inode_number ← CREATE (filename, mode);          // Yes, create it.
        } else return FAILURE;
        inode ← INODE_NUMBER_TO_INODE (inode_number);
        if PERMITTED (inode, flags) then {    // Does this user have the required permissions?
                file_index ← INSERT (file_table, inode_number);
                fd ← FIND_UNUSED_ENTRY (fd_table);// Yes, find entry in file descriptor table
                fd_table[fd] ← file_index;              // Record file index for the file descriptor
                return fd;                        // Return fd
        } else return FAILURE;                    // No, return a failure
}
```

UNIVERSITY OF COPENHAGEN

ITU CPH

API Calls: Read

(skippable

File system state
inode_table
file_table
Process state:
fd_table
wd

File name	Inode number	cursor
program	10	64
Paper	12	0

```
procedure READ (fd, reference buf, n) {
        file_index ← fd_table[fd];
        cursor ← file_table[file_index].cursor;
        inode ← INODE_NUMBER_TO_INODE (file_table[file_index].inode_number);
        m = MINIMUM (inode.size – cursor, n);
        atime of inode ← NOW ();
        if m = 0 then return END_OF_FILE;
        for i from 0 to m – 1 do {
                b ← INODE_NUMBER_TO_BLOCK (i, inode_number);
                COPY (b, buf, MINIMUM (m – i, BLOCKSIZE));
                i ← i + MINIMUM (m – i, BLOCKSIZE);
        }
        file_table[file_index].cursor ← cursor + m;
        return m;
}
```

UNIVERSITY OF COPENHAGEN

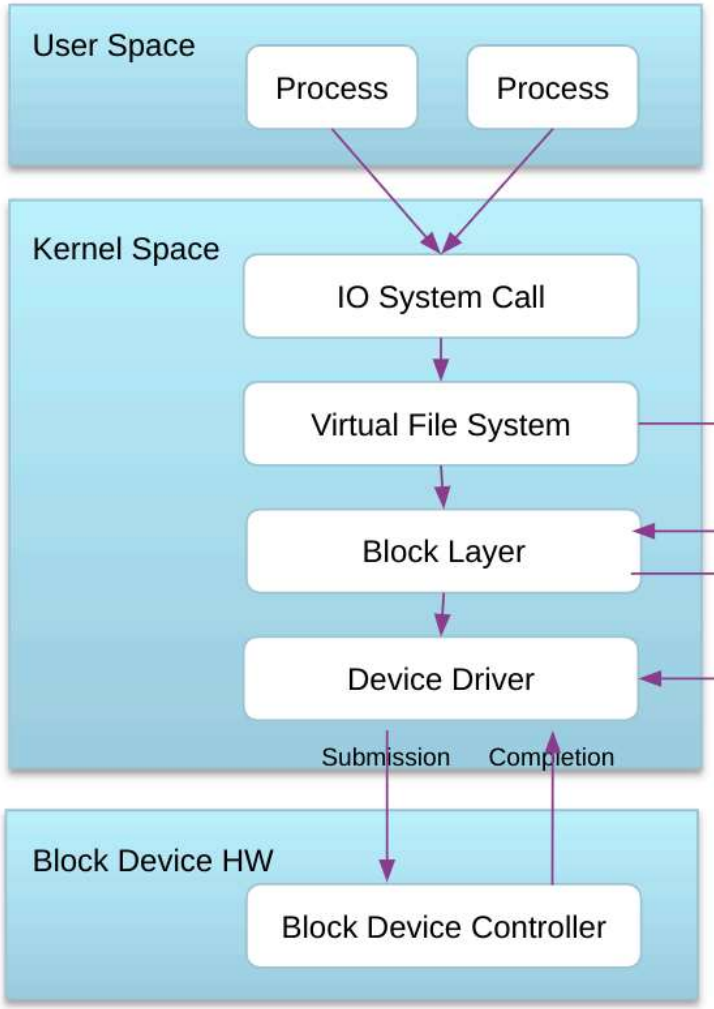
ITU CPH

File Systems

implementation, in Linux

ITU CPH

Linux File System



Linux virtual file system: software layer (in kernel), that provides the *file system interface* to *userspace programs*. (a uniform way for different file systems to hook up to Linux. (POSIX). including *virtual file systems* (aka. synthetic file systems))

Linux virtual file system: aka. **virtual filesystem switch** (I like this better; avoids concept overloading)

UNIVERSITY OF COPENHAGEN

11. ITU CPH

Linux Virtual File System

The virtual file system defines the generic file system interface and data structures:

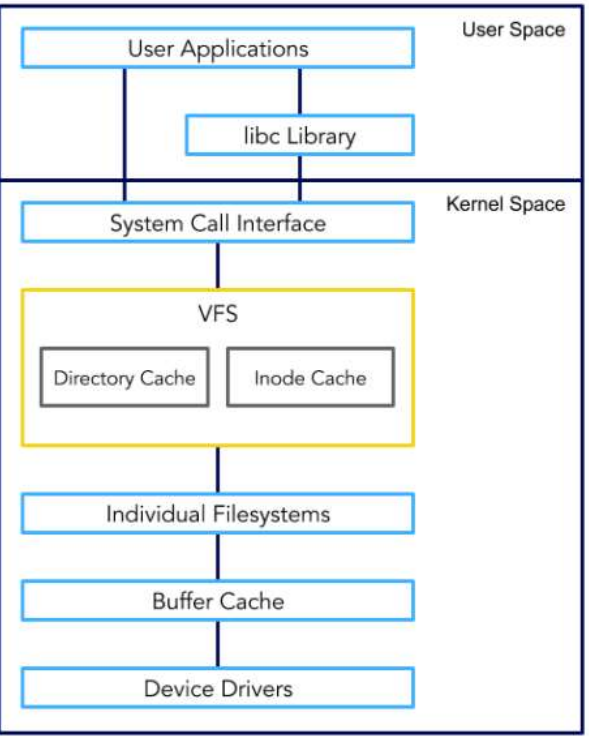
file, dentry, inode, vfsmount, super_block.

Each specific file system provides a specific implementation:

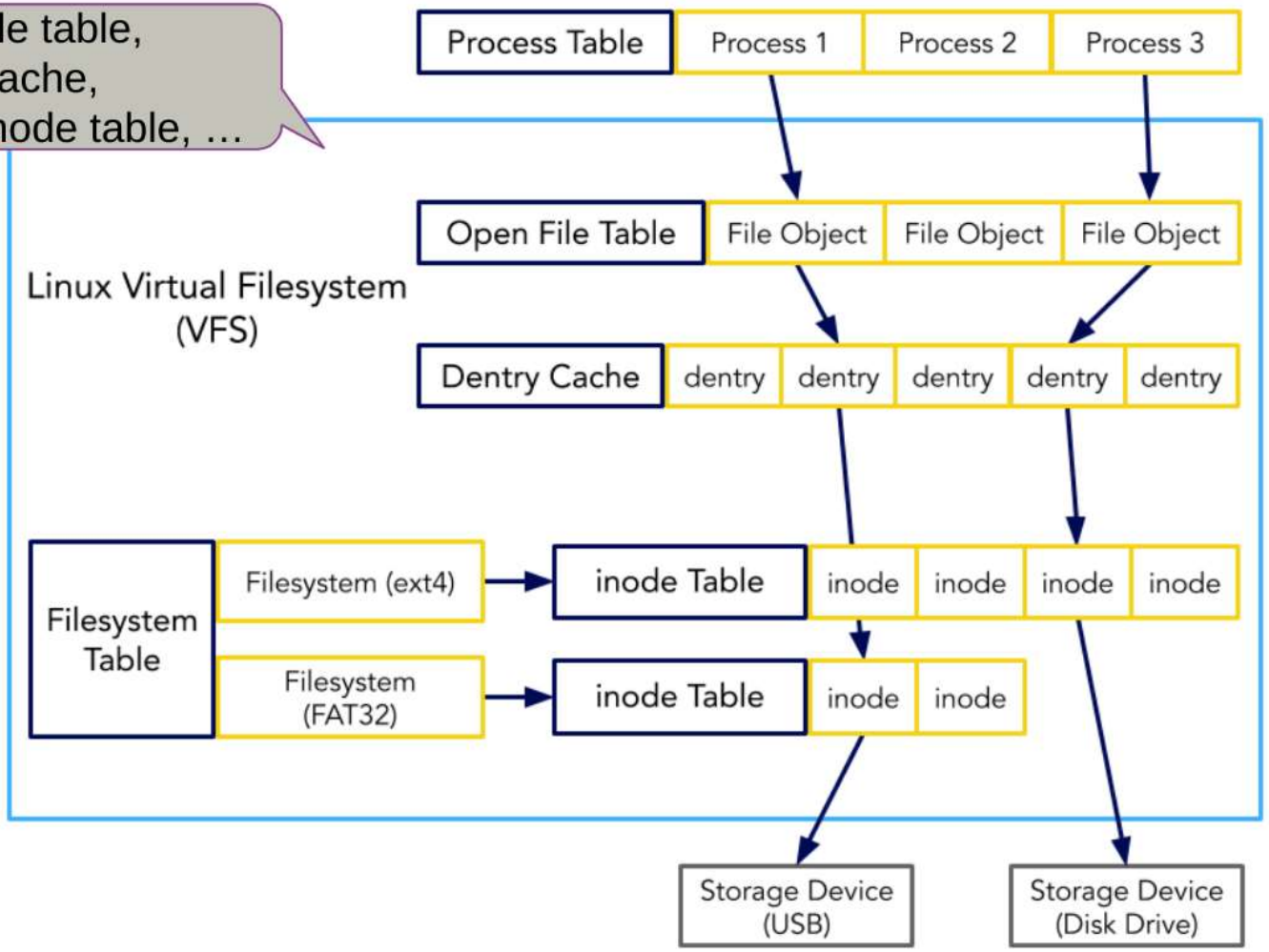
block-based FS (ext4, btrfs), network FS (NFS, ceph),
stackable FS, pseudo FS (sysfs),
special purpose FS (tmpfs)

Linux VFS

they all respect the VFS setup.



file table,
cache,
inode table, …



don't be afraid of VFS; it's just inodes, inode table, etc.

UNIVERSITY OF COPENHAGEN

https://www.starlab.io/blog/introduction-to-the-linux-virtual-filesystem-vfs-part-i-a-hi

ITU CPH

Aside

computational storage

ITU CPH

Computational Storage

Put Everything in Future (Disk) Controllers (it's not "if", it's "when?")

Jim Gray

http://www.research.Microsoft.com/~Gray

Acknowledgements:
Dave Patterson explained this to me a year ago
Kim Keeton, **Erik Riedel**, **Catharine Van Ingen** } **Helped me sharpen these arguments**

1

Basic Argument for x-Disks

- Future disk controller is a super-computer.
  - 1 bips processor
  - 128 MB dram
  - 100 GB disk plus one arm
- Connects to SAN via high-level protocols
  - RPC, HTTP, DCOM, Kerberos, Directory Services,….
  - Commands are RPCs
  - management, security,….
  - Services file/web/db/… requests
  - Managed by general-purpose OS with good dev environment
- Move apps to disk to save data movement
  - need programming environment in controller

Jim Gray, NASD Talk, 6/8/98
http://jimgray.azurewebsites.net/jimgraytalks.htm

Computational storage = Computation on the IO Path

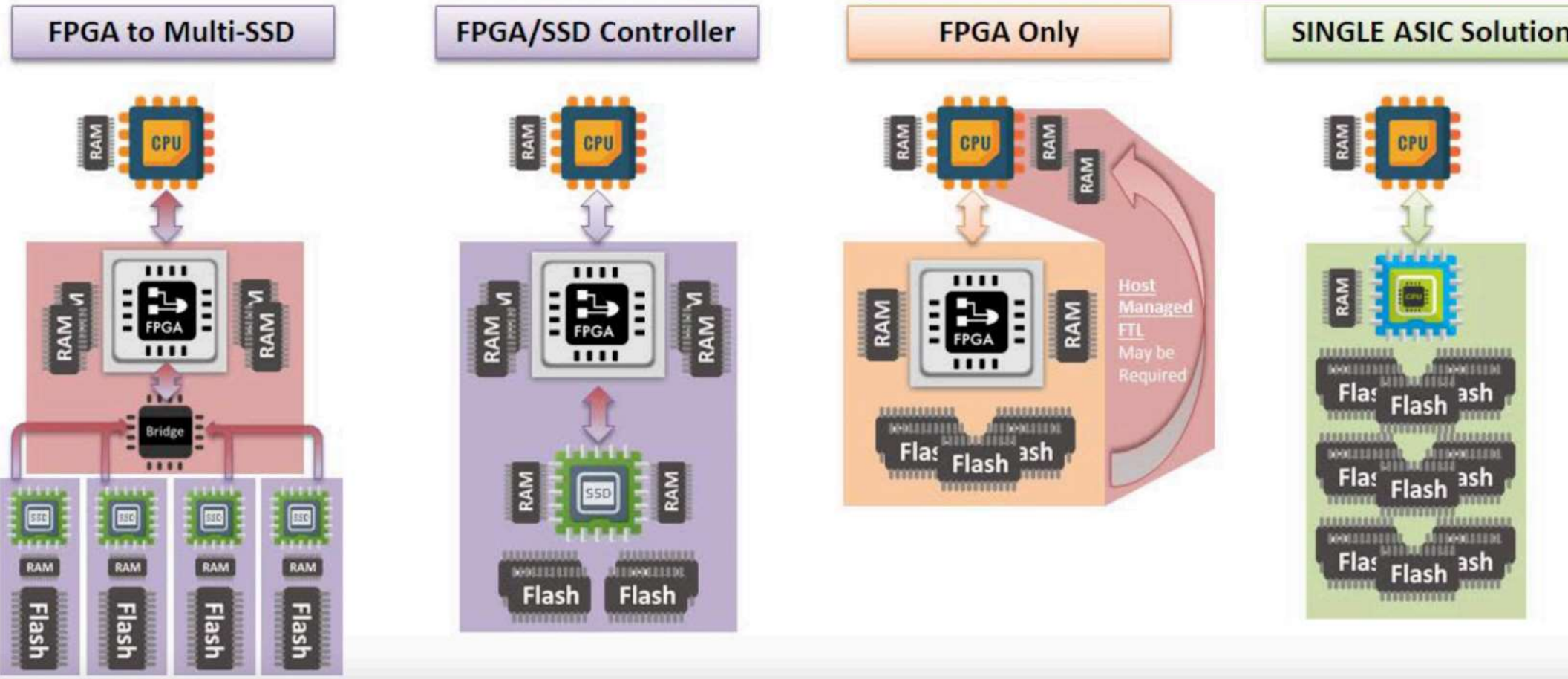
UNIVERSITY OF COPENHAGEN

ITU CPH

Computational storage

https://www.youtube.com/watch?v=3CeOIY1PO-Y

storage industry has defined an architecture for computational storage. NVMe will be standard.



Specialized storage interface + functionality offload

UNIVERSITY OF COPENHAGEN

ITU CPH

Communication Abstraction

what we get to… is a **communication** abstraction.
not just blocks, but e.g. 8MB object, transactions, …

Source: Saltzer and Kaashoe

File Systems

virtual file systems

ITU CPH

In Unix, Everything is a File

originated in Plan 9 mid-80s. concept is two-fold

- *in UNIX, everything is a stream of bytes*
- *in UNIX, the filesystem is used as a universal namespace*

global file system as unified namespace for heterogeneous resources. (that is very convenient).

in nutshell: have stream of bytes $\Rightarrow$ can implement the file I/O API, s.t. invocations of the file I/O API (through the global file system, on a mount-point for the byte-stream) will do *whatever your file I/O API implementation decides* to said byte stream.

ITU CPH

ple of pseudo filesystems are:

(`/proc`): The proc filesystem contains a hierarchy of special files which can be to query or control running processes or peek into the kernel internals through ard file entries (mostly text based).

(`/dev` or `/devices`): Devfs presents all devices on the system as a dynamic ystem namespace. Devfs also manages this namespace and interfaces directly with el device drivers to provide intelligent device management – including device registration/unregistration.

(`/tmp`): Temporary filesystem whose content disappear on reboot. Tmpfs is ned for speed and efficiency with features such as dynamic filesystem size as well emory storage with transparent fallback to swap space.

fs (`/p`): With the BSD portal filesystem you can attach a server process to the ystem global namespace. This can be used to provide transparent access to ork services through the filesystem. For instance an application could interact with MTP server hosted by `ph7spot.com` just by opening a regular file: `p/ph7spot.com/smtp`. The Portal filesystem is somewhat magical in that it provides et semantic in the filesystem which can be piped and leveraged by standard UNIX (e.g. `cat`, `grep`, `awk`, etc.) – even from the shell!

ITU CPH

2.1 High-Level Architecture

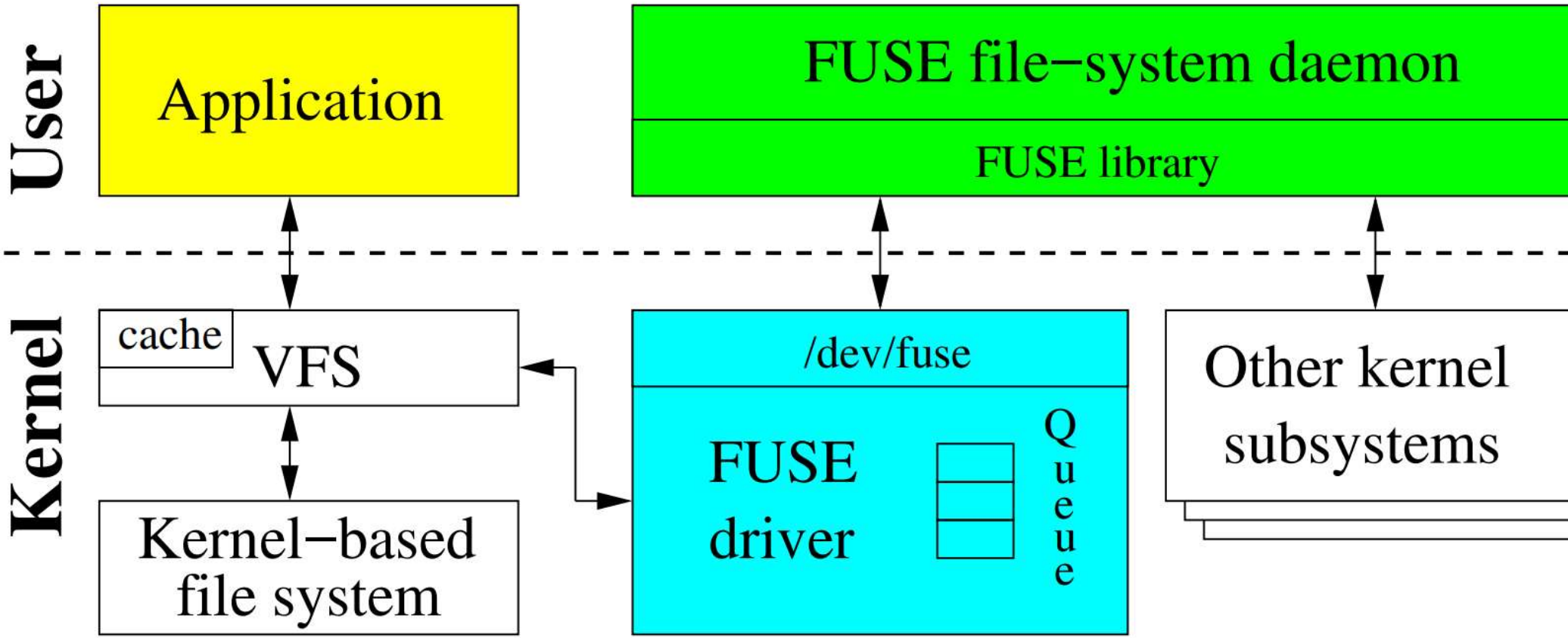


Figure 1: FUSE high-level architecture.

ITU CPH

FUSE consists of a kernel part and a user-level da
mon. The kernel part is implemented as a Linux kern
module that, when loaded, registers a *fuse* file-syste
driver with Linux's VFS. This *Fuse* driver acts as a prox
for various specific file systems implemented by diffe
ent user-level daemons.

In addition to registering a new file system, FUSE
kernel module also registers a `/dev/fuse` block d
vice. This device serves as an interface between use
space FUSE daemons and the kernel. In general, da
mon reads FUSE requests from `/dev/fuse`, processe
them, and then writes replies back to `/dev/fuse`.

ITU CPH

README

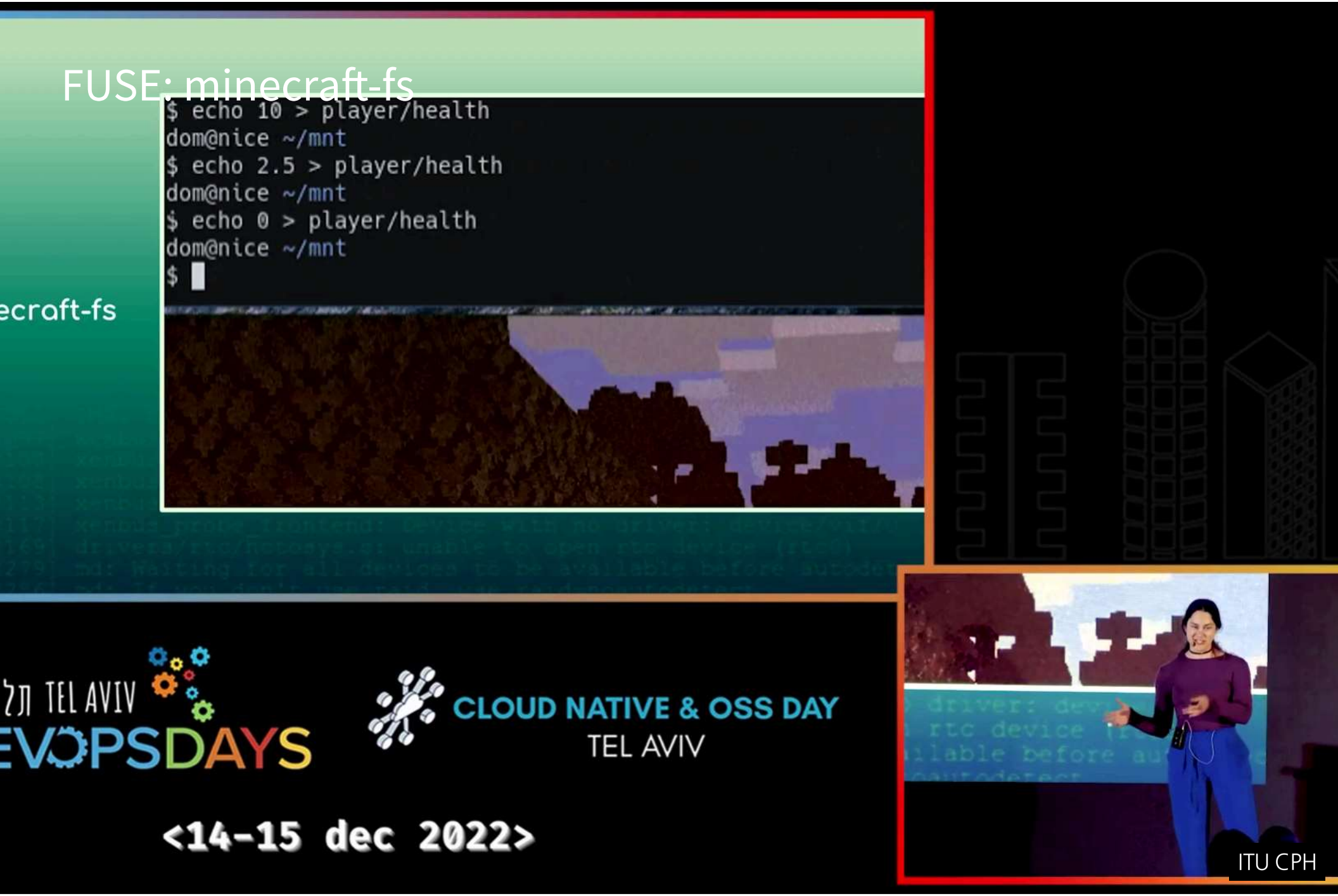
FUSE: Example Implementations

Built For

- gcsfuse - A user-space file system for interacting with Google Cloud Storage. Language: Golang.
- sshfs - File system based on the SSH File Transfer Protocol; same authors as osxfuse. Language: C.
- sshfs - A network filesystem client to connect to SSH servers; same authors as libfuse. Language: C.
- go-ipfs - IPFS implementation in go. Language: Golang.
- mirrorfs - Go filesystem project using bazil/fuse. Language: Golang.
- gocryptfs - Encrypted overlay filesystem written in Go. Language: Golang.
- tahoe-lafs - The Tahoe-LAFS decentralized secure filesystem. Language: Python.
- btfs - A bittorrent filesystem based on FUSE. Language: C++.
- google-drive-ocamlfuse - FUSE filesystem over Google Drive. Language: OCaml.
- mp3fs - FUSE-based transcoding filesystem from FLAC to MP3. Language: C++.
- encfs - An Encrypted Filesystem for FUSE. Language: C++.
- GDriveFS - An innovative FUSE wrapper for Google Drive; Language: Python.
- pachyderm - Containerized Data Analytics. Language: Golang.
- camlistore - Personal storage system for life: a way of storing, syncing, sharing, modelling and backing up content. Language: Golang.
- svfs - The Swift Virtual File System. Language: Golang.
- restic - restic backup program. Language: Golang.
- unionfs-fuse - union filesystem using fuse. Language: C.
- GlusterFS - Storage for your Cloud. Language: C.
- LoggedFS - Filesystem monitoring with Fuse. Language: C++.
- go-mtpfs - Mount MTP devices over FUSE. Language: Go.
- πfs - Filesystem that stores your data in π. Language: C.
- s3fs - FUSE-based file system backed by Amazon S3. Language: C++.

ITU CPH

FUSE: minecraft-fs



Takeaways: Files

ITU CPH

Take-Aways

File abstraction is one of Unix enduring contribution. Beautiful example of a deep module.

You should be able to describe the data structures and name mapping steps involved in file system operations.

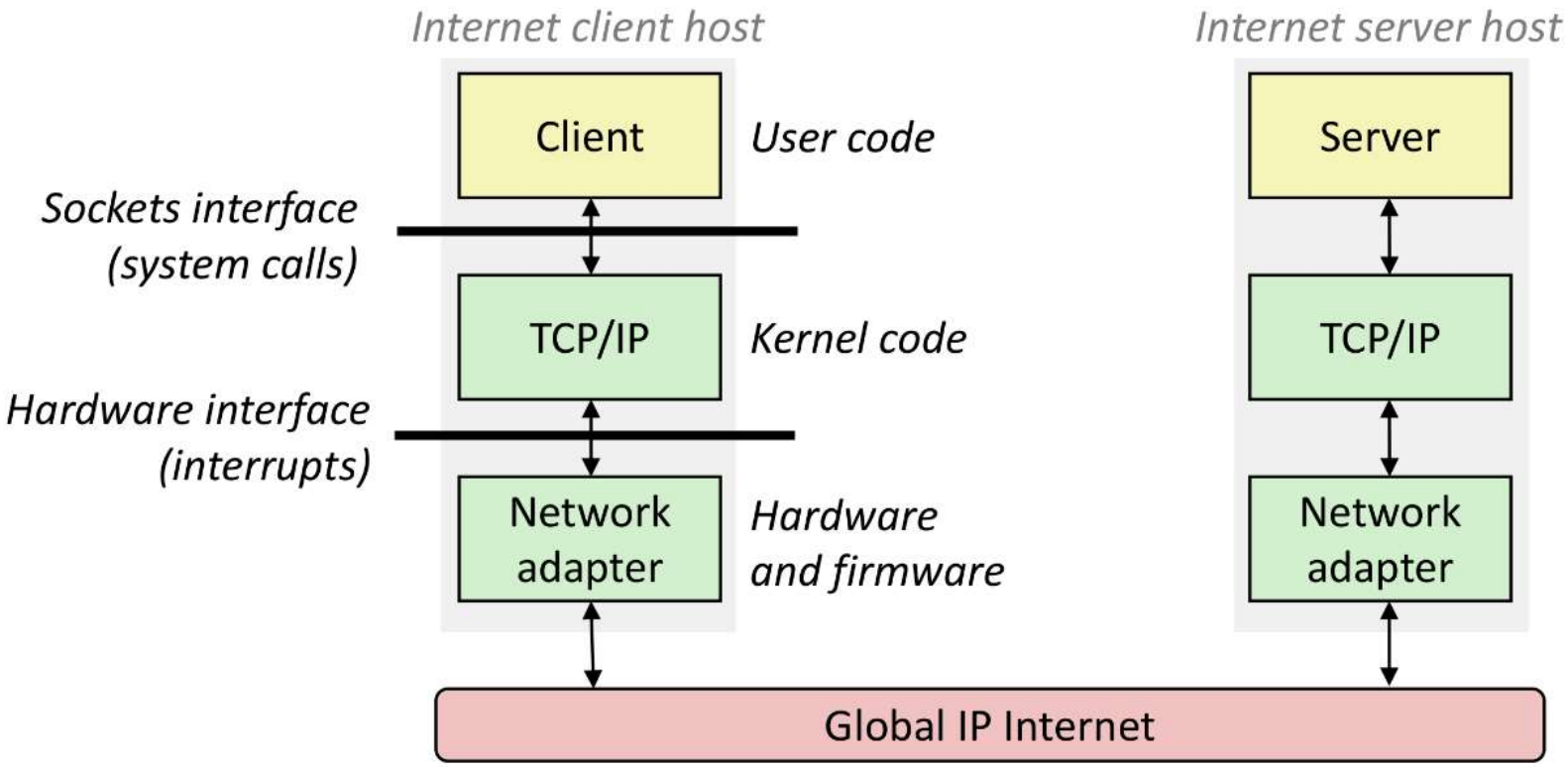
With computational storage, storage devices are moving from a memory to a communication abstraction.

I/O API

sockets

ITU CPH

Hardware and Software Organization of an Internet Application



UNIVERSITY OF COPENHAGEN

ITU CPH

Global IP Internet (upper case)

Most famous example of an internet

Based on the TCP/IP protocol family

- IP (Internet Protocol) :
  - Provides *basic naming scheme* and unreliable *delivery capability* of packets (datagrams) from *host-to-host*
- UDP (Unreliable Datagram Protocol)
  - Uses IP to provide *unreliable* datagram delivery from *process-to-process*
- TCP (Transmission Control Protocol)
  - Uses IP to provide *reliable* byte streams from *process-to-process* over *connections*

Accessed via a mix of Unix file I/O and functions from the *sockets interface*

A Programmer's View of the Internet

1. Hosts are mapped to a set of 32-bit *IP addresses*

   130.226.140.95 (2003)

   - We are trying to fully replace 32-bit addresses with 64-bits addresses. It has not happened yet (2022)

2. The set of IP addresses is mapped to a set of identifiers called Internet *domain names*

   130.226.140.95 is mapped to cos.itu.dk

3. A process on one Internet host can communicate with a process on another Internet host over a *connection*

(1) IP Addresses

32-bit IP addresses are stored in an *IP address struct*

IP addresses are always stored in memory in *network byte order* (big-endian byte order)

True in general for any integer transferred in a packet header from one machine to another.

E.g., the port number used to identify an Internet connection.

```
/* Internet address structure */
struct in_addr {
    uint32_t  s_addr; /* network byte order (big-endian) */
};
```

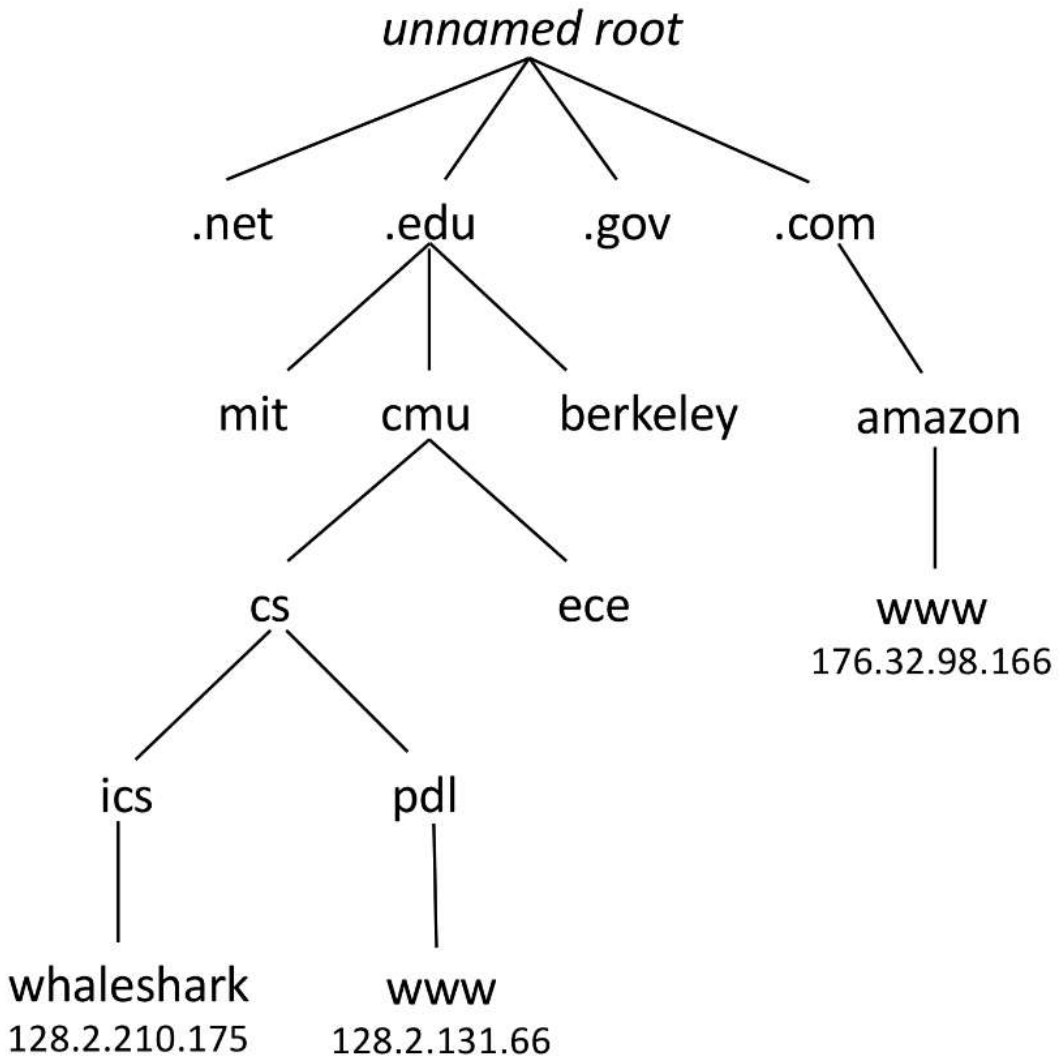
Dotted Decimal Notation

By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period

IP address: `0x8002C2F2 = 128.2.194.242`

Use `getaddrinfo` and `getnameinfo` functions (described later) to convert between IP addresses and dotted decimal format.

(2) Internet Domain Names



First-level domain names

Second-level domain names

Third-level domain names

UNIVERSITY OF COPENHAGEN

ITU CPH

Domain Naming System (DNS)

The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called *DNS*

Conceptually, programmers can view the DNS database as a collection of millions of *host entries*.

- Each host entry defines the mapping between a set of domain names and IP addresses.
- In a mathematical sense, a host entry is an equivalence class of domain names and IP addresses.

(3) Internet Connections

Clients and servers communicate by sending streams of bytes over *connections*. Each connection is:

Point-to-point: connects a pair of processes.

Full-duplex: data can flow in both directions at the same time,

Reliable: stream of bytes sent by the source is eventually received by the destination in the same order it was sent.

A *socket* is an endpoint of a connection

Socket address is an **`IPaddress:port`** pair

A *port* is a 16-bit integer that identifies a process:

***Ephemeral port*:** Assigned automatically by client kernel when client makes a connection request.

Well-known port: Associated with some *service* provided by a server (e.g., port 80 is associated with Web servers)

Well-known Ports and Service

Popular services have permanently assigned *well-known ports* *and* corresponding *well-known service names*:

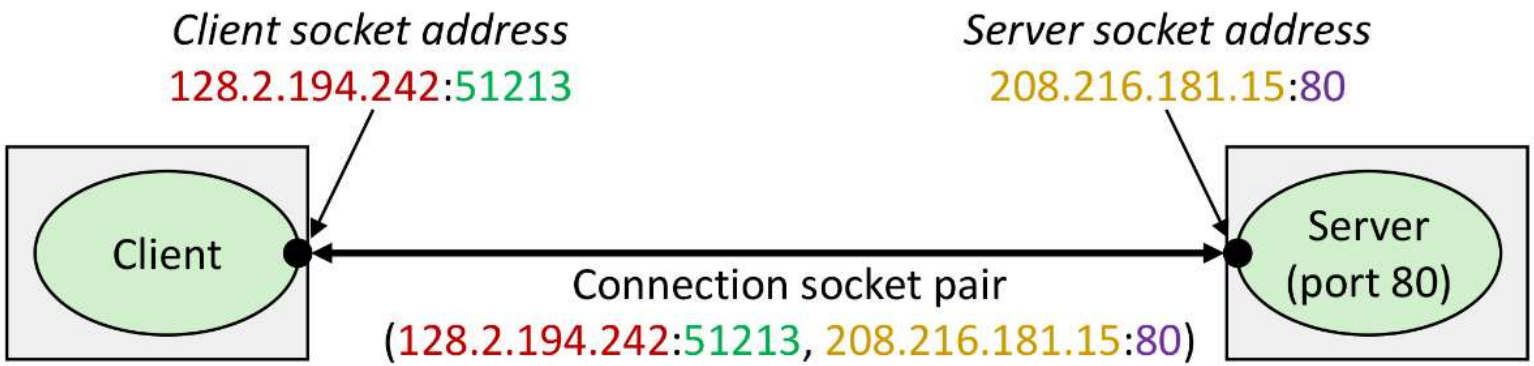
- echo server: 7/echo
- ssh servers: 22/ssh
- email server: 25/smtp
- Web servers: 80/http

Mappings between well-known ports and service names is contained in the file `/etc/services` on each Linux machine.

Anatomy of a Connection

A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)

```
(cliaddr:cliport, servaddr:servport)
```



Client host address
128.2.194.242

51213 is an ephemeral port allocated by the kernel

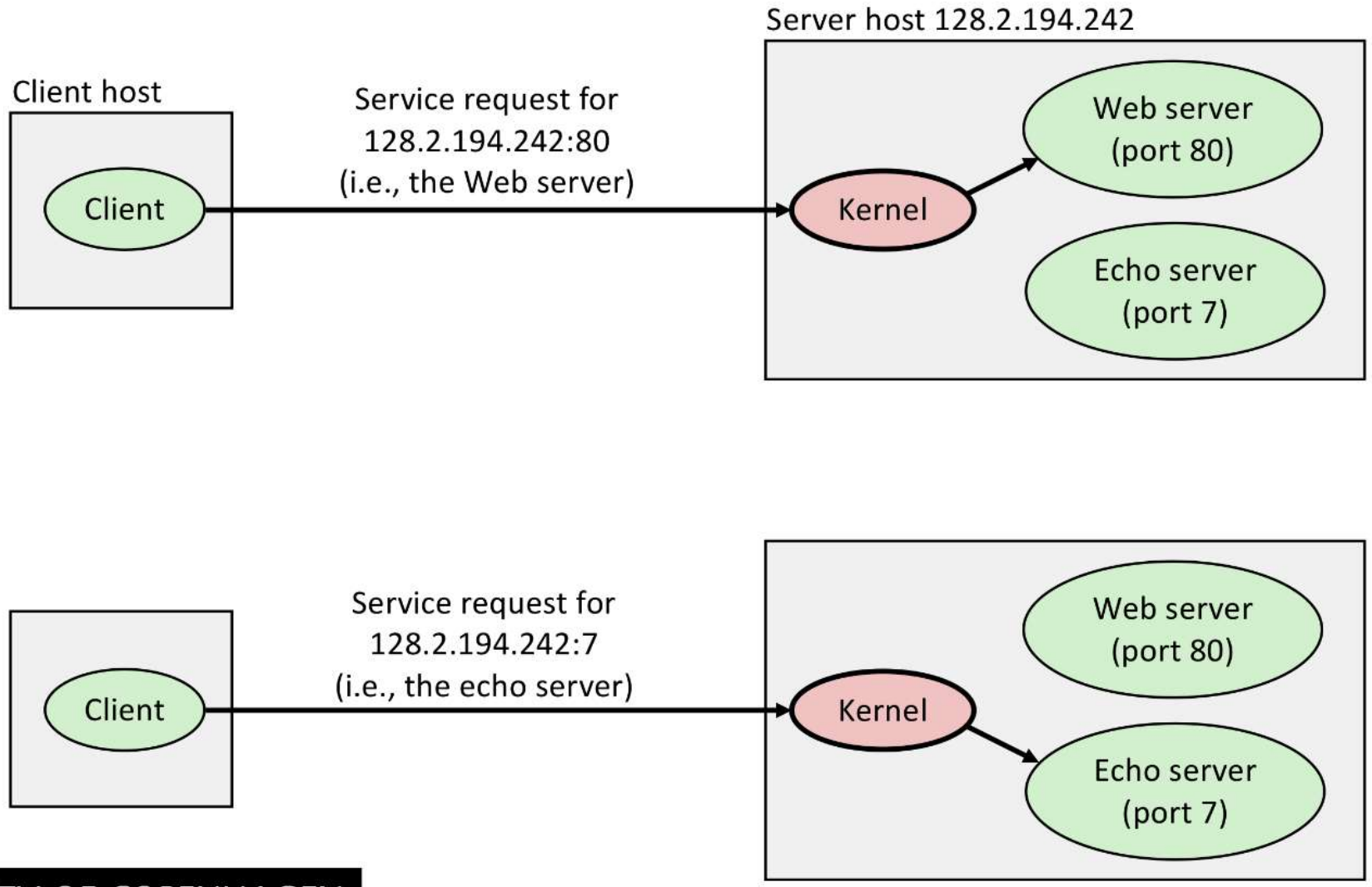
Server host address
208.216.181.15

80 is a well-known port associated with Web servers

UNIVERSITY OF COPENHAGEN

ITU CPH

Using Ports to Identify Services



UNIVERSITY OF COPENHAGEN

ITU CPH

Sockets Interface

Set of system-level functions used in conjunction with Unix I/O to build network applications.

Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.

Available on all modern systems

Unix variants, Windows, OS X, IOS, Android, ARM

Sockets

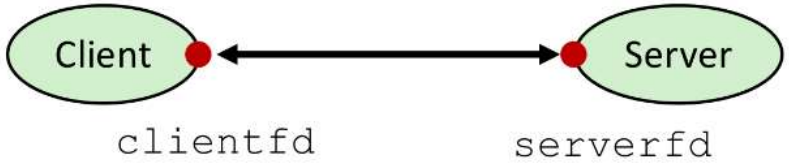
What is a socket?

To the kernel, a socket is an endpoint of communication

To an application, a socket is a file descriptor that lets the application read/write from/to the network

Remember: All Unix I/O devices, including networks, are modeled as files

Clients and servers communicate with each other by reading from and writing to socket descriptors



The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors

Socket Address Structures

Generic socket address:

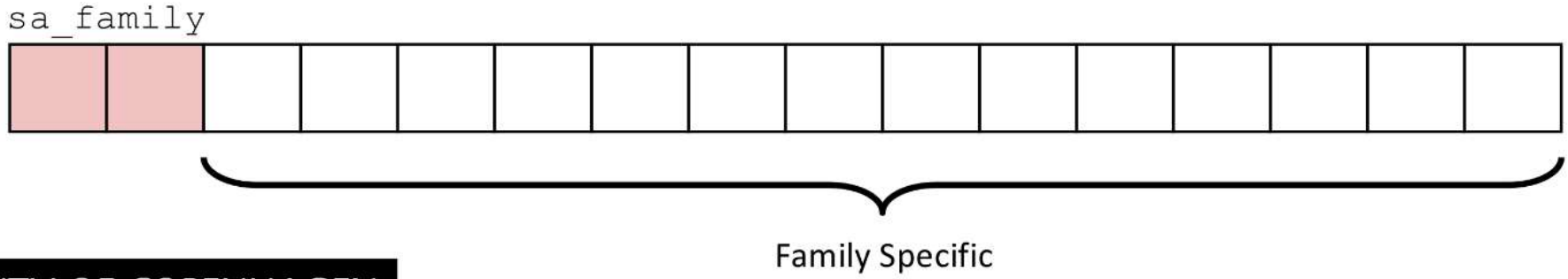
For address arguments to **`connect`**, **`bind`**, and **`accept`**

Necessary only because C did not have generic (**`void *`**) pointers when the sockets interface was designed

For casting convenience, we adopt the Stevens convention:

```
typedef struct sockaddr SA;
```

```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data.  */
};
```

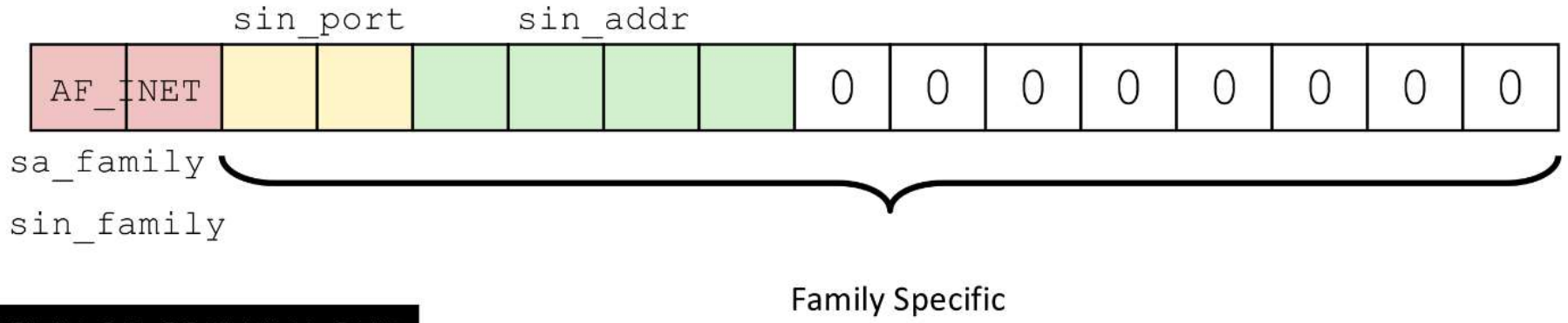


Socket Address Structures

Internet-specific socket address:

Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

```
struct sockaddr_in  {
  uint16_t        sin_family;  /* Protocol family (always AF_INET) */
  uint16_t        sin_port;    /* Port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```



Host and Service Conversion: `getaddrinfo`

```
int getaddrinfo(const char *host,              /* Hostname or address */
                const char *service,           /* Port or service name
*/
                const struct addrinfo *hints,/* Input parameters */
                struct addrinfo **result);   /* Output linked list */

void freeaddrinfo(struct addrinfo *result);  /* Free linked list */

const char *gai_strerror(int errcode);       /* Return error msg */
```

Given `host` and `service`, `getaddrinfo` returns `result` that points to a linked list of `addrinfo` structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.

Helper functions:

- `freeadderinfo` frees the entire linked list.
- `gai_strerror` converts error code to an error message.

UNIVERSITY OF COPENHAGEN

ITU CPH

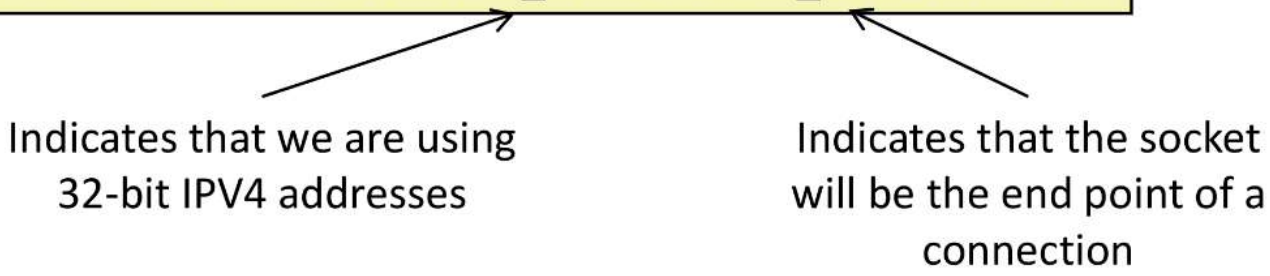
Sockets Interface: `socket`

Clients and servers use the `socket` function to create a *socket descriptor*:

Example:

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

```
int clientfd = socket(AF_INET, SOCK_STREAM, 0);
```



Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.

Sockets Interface: `bind`

A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`. Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`.

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

Sockets Interface: `listen`

By default, kernel assumes that descriptor from socket function is an *active socket* that will be on the client end of a connection. A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

```
int listen(int sockfd, int backlog);
```

Converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients.

`backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.

Sockets Interface: `accept`

Servers wait for connection requests from clients by calling `accept`:

```
int accept(int listenfd, SA *addr, int *addrlen);
```

Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client's socket address in `addr` and size of the socket address in `addrlen`.

Returns a *connected descriptor* that can be used to communicate with the client via Unix I/O routines.

Sockets Interface: `connect`

A client establishes a connection with a server by calling connect:

```
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

Attempts to establish a connection with server at socket address `addr`

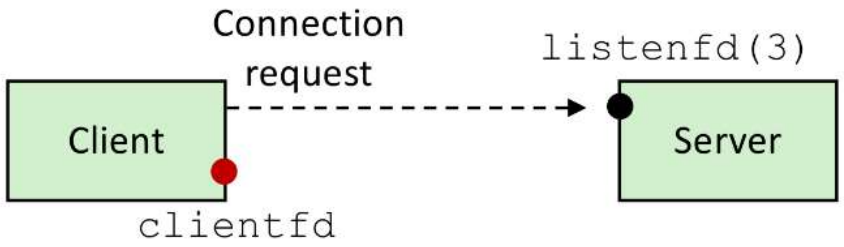
- If successful, then `clientfd` is now ready for reading and writing.
- Resulting connection is characterized by socket pair

  `(x:y, addr.sin_addr:addr.sin_port)`
  - `x` is client address
  - `y` is ephemeral port that uniquely identifies client process on client host

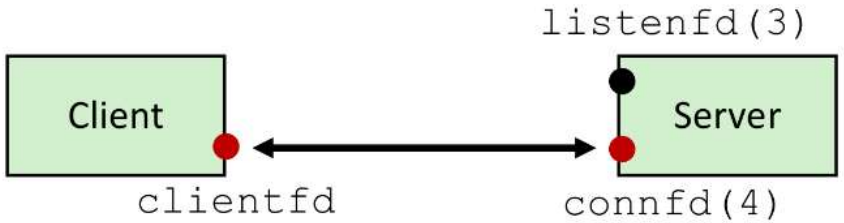
Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

accept Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`



2. Client makes connection request by calling and blocking in `connect`



3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

UNIVERSITY OF COPENHAGEN

ITU CPH

Connected vs. Listening Descriptors

Listening descriptor

- End point for client connection requests
- Created once and exists for lifetime of the server

Connected descriptor

- End point of the connection between client and server
- A new descriptor is created each time the server accepts a connection request from a client
- Exists only as long as it takes to service client

Why the distinction?

- Allows for concurrent servers that can communicate over many client connections simultaneously
  - E.g., Each time we receive a new request, we fork a child to handle the request

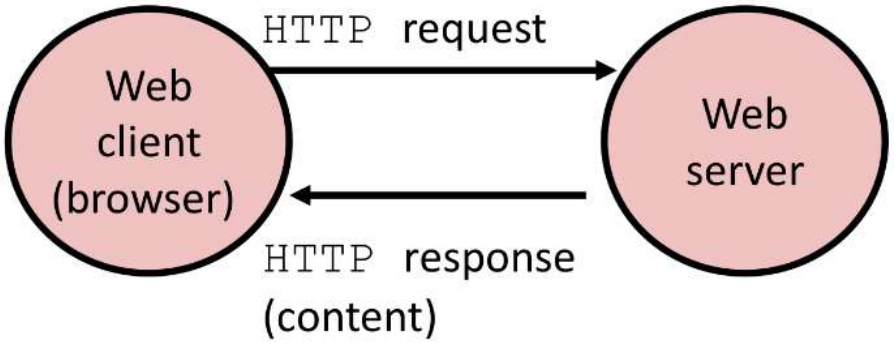
Web Server Basics

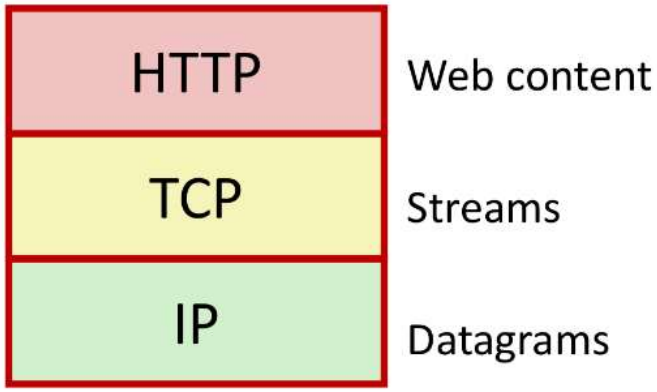
Clients and servers communicate using the HyperText Transfer Protocol (HTTP)

- Client and server establish TCP connection
- Client requests content
- Server responds with requested content
- Client and server close connection (eventually)

Current version is HTTP/1.1

- RFC 2616, June, 1999.





http://www.w3.org/Protocols/rfc2616/rfc2616.html

UNIVERSITY OF COPENHAGEN

ITU CPH

Takeaways: Sockets

ITU CPH

Take-Aways

Network as a strictly layered system: physical (ethernet), kernel (IP/TCP), applications

Protocol is a set of rules that governs how hosts and routers should cooperate when they transfer data from network to network via naming scheme and delivery mechanism.

Socket as communication abstraction. To the kernel, a socket is an endpoint of communication. To an application, a socket is a file descriptor that lets the application read/write from/to the network. Client connects to a server via a socket. Servers bind sockets to address:port, listen and accept incoming connections from clients. Thereafter, clients and servers can read and write.

I/O API: MPI

ITU CPH

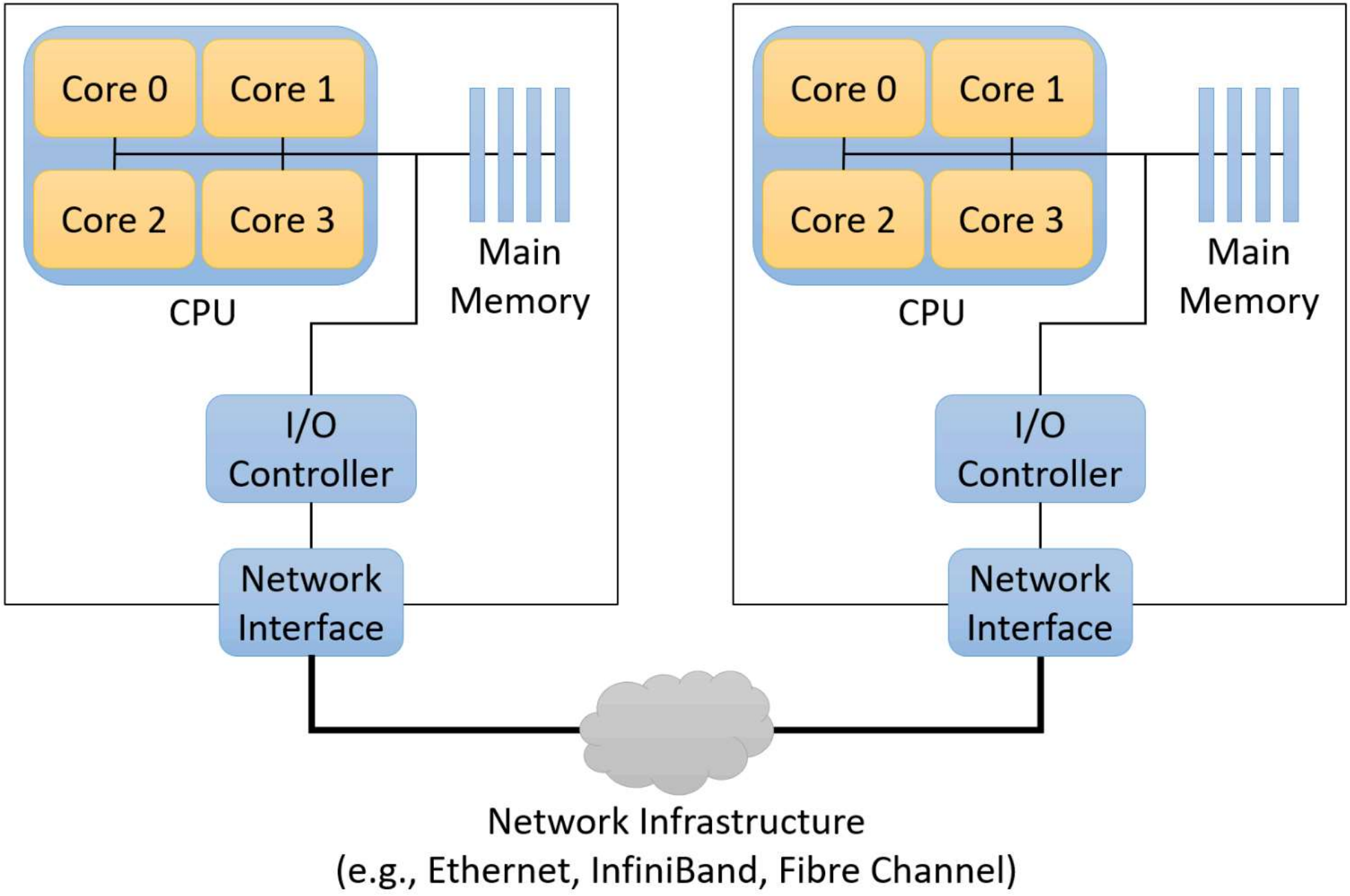


Figure 218. The major components of a shared-nothing distributed memory architecture built from two compute nodes

ITU CPH

Parallel & Distributed Processing

models:

- client/server
- pipeline (each worker processes data independently)
- boss/worker (one worker process orchestrates)
- peer-to-peer (coordination/consensus algorithm needed)

communication: via. message-passing, following a protocol.

example paradigm: scatter/gather (example in book); boss scatters array across workers, gathers result from them.

ITU CPH

15.2.4. MPI Hello World

As an introduction to MPI programming, consider the "hello world" program (hello_world_mpi.c) presented here:

```c
#include <stdio.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc, char **argv) {
    int rank, process_count;
    char hostname[1024];

    /* Initialize MPI. */
    MPI_Init(&argc, &argv);

    /* Determine how many processes there are and which one this is. */
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* Determine the name of the machine this process is running on. */
    gethostname(hostname, 1024);

    /* Print a message, identifying the process and machine it comes from. */
    printf("Hello from %s process %d of %d\n", hostname, rank, process_count);

    /* Clean up. */
    MPI_Finalize();

    return 0;
}
```

When starting this program, MPI simultaneously executes multiple copies of it as independent processes across one or more computers. Each process makes calls to MPI to determine how many total processes are executing (with

ITU CPH

MPI Hello World

compiling (requires framework, e.g. OpenMPI or MPICH)

```
1 $ mpicc -o hello_world_mpi hello_world_mpi.c
```

running:

```
1 $ mpirun -np 8 --hostfile hosts.txt ./hello_world_mpi
2 Hello from lemon process 4 of 8
3 Hello from lemon process 5 of 8
4 Hello from orange process 2 of 8
5 Hello from lemon process 6 of 8
6 Hello from orange process 0 of 8
7 Hello from lemon process 7 of 8
8 Hello from orange process 3 of 8
9 Hello from orange process 1 of 8
```

ITU CPH

Takeaways: MPI

ITU CPH

Takeaways

- distributed processing is challenging
  - sharing
  - failure
  - races
  - consensus
- fortunately, frameworks (MPI) exist (based on decades of research) that make this easier.

ITU CPH