Week 1: Bits, Bytes, and Data Representation

- **Binary and Hexadecimal Numbers**: Modern computers use the binary system (base-2) internally. A single **bit** is 0 or 1, and 8 bits make a **byte**. We often group bits in hex (base-16) for readability. For example, the hex number **AA** in base-16 equals decimal **170** *(Exam Q1)*. Converting between bases is a key skill – e.g., binary `1010` = hex `A` = decimal 10.
- **Number Representation**: Because a computer's bit-length is finite, integers have limited range. An 8-bit unsigned integer ranges 0–255. Negative integers are commonly stored in **two's complement** form so that subtraction and addition use the same circuitry. For instance, in 8-bit two's complement, 11111111_2 represents –1. Overflow occurs if the result exceeds the bit-width.
- **Bitwise Operations in Python**: Python supports binary literals (prefixed with `0b`) and bitwise ops. The **shift left** `<<` and **shift right** `>>` operators move bits, and `&` is bitwise AND (mask). For example:

  ```
  x = 0b01001011  # 0x4B in binary (75 decimal)
  result = (x >> 2) & 0b111
  print(bin(result))  # Output: 0b100
  ```

  Here we shifted `0b01001011` right by 2 (dropping two least significant bits) and then masked with `0b111` to get the lowest 3 bits of the shifted value, resulting in `0b100` (binary 4)[1]. This illustrates extracting specific bits *(Exam Q5)*. Similarly, `(0b11001011 << 4) >> 4` in Python zeroes out the upper 4 bits of an 8-bit number, useful for masking *(Exam Q6)*.
- **Character Encoding**: Characters are represented by numeric codes. **ASCII** uses 7 bits for basic English letters (e.g., `'A'` = 65)[2][3]. This yields 128 possible characters, which is insufficient for other languages. **Unicode** extends this, using up to 32 bits per character to cover over 140,000 characters across languages[4][5]. (Unicode characters are often stored in UTF-8 encoding, which uses multiple bytes per character as needed.)
- **Stored Program Concept & Machine Instructions**: In computers, **programs and data both reside in memory as binary**[6][7]. A CPU fetches machine instructions (bit patterns) from memory and executes them. For example, a hypothetical instruction $0110\ 0100\ 1000\ 1100_2$ could mean *"add value at address 4 to value at address 8, store result at address 12."* This idea of storing instructions as data is the basis of the **Von Neumann architecture** (stored-program computer)[8]. High-level code (like C or Python) gets compiled or interpreted down to these machine instructions.
- **Operating System Overview**: The OS is the master program managing hardware and software resources. It provides services like **process management**, **memory management**, **file systems**, **I/O control**, and a user interface[9][10]. For example,

modern OSes enable **concurrency** (running multiple programs at once) and security protections. The OS loads programs into memory and schedules the CPU to run them. Essentially, it acts as an intermediary between user applications and the hardware. Key components introduced include the **kernel** (core OS code running with high privileges) and system calls (interfaces for programs to request OS services).

- **File System Basics**: Data is persistently stored in a **file system**, which organizes files into directories (folders) in a hierarchical namespace. Filenames are typically in a tree structure (e.g., `/home/user/doc.txt` in Unix). The OS provides system calls to create, read, write, and delete files. Common file systems include **ext4** on Linux and **NTFS** on Windows. (Notably, **ext4** is a native Linux file system *(Exam Q14)*.) File systems manage metadata like file permissions and keep track of which blocks on disk belong to each file.

Key Exam Connections: Binary/hex conversions and bit operations are frequently tested (e.g., converting AA_{16} to decimal *(Exam Q1)*, understanding bit shifts/masks *(Exam Q5, Q6)*). Fundamental concepts of the stored program and OS roles provide background for many topics but are not directly quizzed with a specific question. However, understanding these underpins process management questions in later weeks.

Week 2: Command Line, Shell, and Processes

- **Command Line & Shell Basics**: The *shell* is a command-line interpreter that lets users type commands to perform tasks[11]. In Unix, common shells include **bash** (Bourne Again SHell). The shell prints a prompt (e.g., `$`) and waits for user input. When a command is entered, the shell parses it and executes the corresponding program. For example, typing `ls` runs the program that lists directory contents.
- *Common file commands*: `pwd` (print current directory), `cd` (change directory), `ls` (list files), `cp` (copy files), `mv` (move/rename files), `rm` (remove files), `mkdir` (make directory), `rmdir` (remove directory)[12][13]. These allow navigation and manipulation of the file system. Special directory names: `.` refers to the current directory, `..` to the parent directory, `~` to the home directory, and `/` to the root of the filesystem[14][15]. For example, `cd ..` moves up one level, and `cd ~/docs` goes to the "docs" folder in your home.
- **Shell Variables and Environment**: The shell can hold variables. Assign with `NAME=value` (no spaces around `=`)[16], and retrieve with `$NAME`[17]. For example:

  ```
  $ NAME=Hazem
  $ echo $NAME
  Hazem
  ```

  Here **`NAME`** is a shell variable containing "Hazem", and `echo` prints it[16][18]. *(Exam Q7:)* Setting a variable and then using `$NAME` will output its value, not the literal

name[19]. Shell variables are strings by default. Arithmetic can be done via the `expr` command or newer shell arithmetic syntax. For instance,

```
$ A=1
$ expr $A + 1
2
```

would output 2[20][21]. To update `A`, command substitution with backticks or `$()` is used:

```
$ A=$(expr $A + 1)  # now A is 2
```

- **Shell Scripts & Control Structures**: A shell script is a file containing shell commands that can be executed like a program. For example, a simple script `myname` could contain:

  ```
  echo "$1 is your name"
  ```

  Running `$ bash myname Bob` prints "Bob is your name" (here `$1` is the first argument)[22][23]. The shell provides programming constructs:

- **If statements**:

  ```
  if [ "$NAME" == "Bob" ]; then
      echo "Name is ok"
  elif [ "$NAME" == "Alice" ]; then
      echo "Name is good"
  else
      echo "Name not recognized"
  fi
  ```

  The condition `[ ]` uses the `test` command. Note the spaces and the use of `==` for string comparison in `bash`[24][25].

- **Loops**: Shell supports `for`, `while`, etc. E.g.,

  ```
  for f in *.txt; do
      echo "File: $f"
  done
  ```

  iterates over all `.txt` files in the current directory.

- **Command substitution**: Using backticks `` `...` `` or `$( ... )` replaces the command with its output. For example, `echo `echo hello`` will first execute the inner `echo hello` (producing "hello") and then the outer echo prints "hello". So, `echo `echo hello`` ultimately outputs "hello" *(Exam Q12)*[26].

- **Pipes and Redirection**: The shell allows chaining commands with **pipes** (`|`) and redirecting I/O to/from files or devices.

- **Redirection**: `>` redirects standard output to a file (overwriting), and `>>` appends to a file. For example, `echo "Hello" > output.txt` writes "Hello" into *output.txt*, creating it if it doesn't exist[27]. `2>` redirects standard error (file descriptor 2). For

instance, `grep "foo" file 2>errors.log` sends any error messages to *errors.log*. *(Exam Q33:)* `echo "Hazem" > 2` will create a file named "2" containing "Hazem", because without an `&`, the shell interprets `2` as a filename[28]. To redirect output to stderr, use `>&2` (for example, `echo "Error" >&2`).

- **Pipes**: `cmd1 | cmd2` connects the output of cmd1 to the input of cmd2. For example, `ls | grep ".txt"` passes the list of files to grep, which filters lines containing ".txt". A common idiom is `command > file 2>&1` to redirect both stdout and stderr to the file (here `2>&1` means "send stderr (2) to the same place as stdout (1)").
- **Here-doc and here-string**: Using `<<` or `<<<` can feed multiline strings to commands, and `cat >> file` allows interactive or programmatic appending. For example, the sequence:

```
$ cat > words   # using > will create/overwrite 'words'
one<Ctrl-D>
$ cat words
one
```

writes "one" into *words*. Using `>>` would append instead. *(Exam Q8)* demonstrated using `cat >> words` and typing "one" to append to the file[29].

- **Common Shell Commands**: In addition to file management, other useful commands include:
- `cat file` – print file content to screen (concatenate).
- `head`/`tail` – show beginning or end of a file.
- `grep "pattern" file` – search for lines matching a regex pattern. For instance, `grep "^a.*e$" list` finds lines starting with `a` and ending with `e` in *list (Exam Q13)*[30][31].
- `wc` – count lines/words/characters.
- `man command` – show manual page for a command (usage help).
- **Processes and the Kernel**: A **process** is a running instance of a program (with its own memory space). In Unix, when you launch a program, the shell creates a new process. The Unix kernel provides system calls like `fork`, `exec`, and `wait` to manage processes[32][33].
- **fork()**: The system call to create a new process by duplicating the current process. The new process (child) is almost an exact copy of the parent, except it gets a unique Process ID (PID)[34]. After a successful `fork`, **both processes continue execution from the next instruction**, but `fork` returns 0 in the child and returns the child's PID in the parent[35].
- **exec()**: The system call to replace the current process's code with a new program. The child often calls an `exec` variant (e.g., `execl`) to run a different program after fork[36][37]. After exec, the new program starts executing (in the same PID), and the original code is gone.

- **wait()**: A call for a parent to wait for its child process to finish execution[38]. This prevents zombie processes and allows the parent to get the child's exit status.
- **Standard I/O and File Descriptors**: By convention, each process has **file descriptor 0** = stdin, **1** = stdout, **2** = stderr. The child inherits open files from the parent, so after fork, parent and child initially share file descriptors[39]. This is why, for example, output redirections set up by the shell apply to the child's output. *(Exam Q35:)* The fd number for stderr is 2[40].
- **Running Processes in Background/Foreground**: Normally, the shell waits for a command to finish. Appending & runs the process in the **background**, letting the shell prompt return immediately[41][42]. Example: `sleep 30 &` runs a 30-second sleep in background. The shell prints a job ID like `[1]` and PID. You can continue using the shell while the background job runs. To see background jobs, use `jobs`. To bring a job to foreground, use **`fg %jobnumber`** (e.g., `fg %1` for job 1). *(Exam Q29:)* `fg 1` brings job #1 (or process with ID 1 in jobs list) to the foreground, resuming its execution on the terminal. To suspend a foreground job, press `<Ctrl-Z>` (sends SIGTSTP). To terminate a job, press `<Ctrl-C>` (SIGINT). *(Exam Q11)*: `<Ctrl-Z>` *suspends* (pauses) a process, whereas `<Ctrl-C>` *terminates* it[43].
- **Process States**: Processes can be in states like **running**, **waiting (blocked)** for I/O, **ready** (runnable but waiting for CPU), or **terminated**. For instance, when a process calls `wait()`, it may sleep until a child exits. These details become important in scheduling (Week 5), but understanding that a process isn't always running helps explain concurrency.
- **Process Hierarchy**: In Unix, every process (except the initial `init` process) has a parent. The shell's spawned processes have the shell's PID as their PPID (parent PID). You can visualize the **fork-exec** workflow as follows:

```
Parent shell (bash, PID 7)
    │
    ├─ fork() → Child process (PID 22, still bash code)
    │              ├─ exec("ls") → now running `ls` program (PID 22)
    │              └─ (child prints directory listing and exits)
    │
    └─ wait(pid 22) (shell waits for child to finish)
```

In the diagram above, the shell forks a child and then waits; the child execs `ls` and eventually exits, notifying the parent to continue[44][45]. After the `ls` completes, the shell prompt returns. *(Exam Q9)*: An instance of a program in execution (like that `ls`) is called a **process**[46].

- **User IDs and Permissions**: Each process has an **owner user ID (UID)** and group ID. This ties into file permissions (a process can only access files it's permitted to). The shell runs under your user ID, and any forked processes inherit that by default. Some commands (like `sudo`) can launch processes with elevated privileges.

- **Viewing Processes**: Use the `ps` command to list processes. `ps` with options shows more details (e.g., `ps -al` shows a detailed list including PID, PPID, state, etc.)[47].

Another command, `top`, provides a live updating list of running processes and their CPU/memory usage. *(Exam Q32:)* To find what processes are running on the system, one can use **ps** (or its alternatives like `top` or `ps aux`)[47]. For example, `ps ax` lists all processes; `ps u` adds user info.

Key Exam Connections: Many shell concepts were tested. For example, using environment variables *(Q7)*, command substitution *(Q12)*, I/O redirection *(Q8, Q33)*, counting script arguments (`$#` in **Q10**, and `$1` in **Q28** for first arg), job control *(Q11, Q29)*, and pipes/grep patterns *(Q13, Q31)*. Knowing that a process is a running program (not just code on disk) is critical *(Q9)*. Understanding `fork/exec` underlies questions like the output of forking code *(see Week 4/5 for Q15, Q34)*. Mastery of shell syntax and process management is crucial for writing shell scripts (see Week 5) and for system operation questions.

Week 3: Emulators, Virtual Machines, and Containers

- **Machine Language & Emulation**: In Week 1 we saw machine instructions as binary patterns executed by the CPU. An **emulator** is a program that **simulates a CPU**, reading those binary instructions and performing their effects in software[48][49]. Essentially, it "pretends" to be a particular computer architecture. For example, the QEMU program can emulate an ARM CPU on an x86 host by reading ARM instructions and executing equivalent behavior. Emulators enabled running software for one machine on a different machine. Early examples include the Java Virtual Machine (JVM), which emulates a hypothetical CPU for Java bytecode. Emulation is typically slower than native execution, but very flexible (it can even emulate older consoles or different endianness CPUs).
- **Assembly vs Machine Code**: To make writing machine instructions easier, programmers use **assembly language** – human-readable mnemonics for binary opcodes[50]. An *assembler* translates assembly (e.g., `ADD 4,8,12`) into the actual machine code bits. This was mentioned to illustrate what an emulator interprets. Emulators might operate at the machine code level, but we write examples in assembly for clarity.
- **Virtual Machines (VMs)**: A **Virtual Machine** uses **virtualization** to run multiple operating systems on one physical hardware. Unlike an emulator that might imitate a different CPU, a VM often uses the same CPU instruction set as the host for efficiency (though it *can* emulate other hardware aspects). A **hypervisor** (virtual machine monitor) like VirtualBox, VMware, or KVM manages VMs. There are two types:
- *Type 1 (bare metal)*: Hypervisor runs directly on hardware (e.g., VMware ESXi, Xen), and hosts multiple OS instances.
- *Type 2 (hosted)*: Hypervisor is a software on top of an OS (e.g., VirtualBox on Windows), and it creates VMs as regular processes on the host.

Each VM has virtualized hardware – CPU, memory, disk, network – and can boot its own OS (called the guest OS). The hypervisor multiplexes the real CPU among VMs and provides each a portion of RAM and disk. Crucially, VMs are **isolated** from each other (memory and process spaces are separate). You can run different OSes (Linux, Windows) concurrently on one physical machine using VMs. The overhead is moderate (some CPU time to manage virtualization, but modern CPUs have virtualization extensions to assist).

- **Containers**: Containers (e.g., Docker, LXC) are a lighter form of virtualization. Instead of emulating an entire hardware machine, containers **share the host OS kernel** but isolate the user-space environment. Key technologies are **cgroups** (to limit resources) and **namespaces** (to isolate processes, networks, file systems) in Linux. A container packages an application with all its dependencies in a lightweight unit, ensuring it runs the same everywhere. Unlike a VM, which might gigabytes in size (including a full OS), a container image is usually tens of megabytes, containing just the app and necessary libraries. Containers start much faster than VMs (since no OS boot is needed; they use the running kernel). They are ideal for deploying microservices. However, because the kernel is shared, containers must use the host OS's kernel (e.g., you cannot run a Windows kernel container on a Linux host, though you can simulate via a VM).
- **Differences – Emulators vs VMs vs Containers**:
- *Emulator*: Can imitate a completely different system (e.g., old game console or different CPU architecture). Slowest, but most flexible. Provides full hardware *simulation*.
- *Virtual Machine*: Runs *multiple OS instances* on the same physical hardware. Each VM feels like a full computer. Requires hypervisor, with some performance overhead (though near-native with modern support). Great for OS-level isolation – different kernels possible.
- *Container*: *Shares OS kernel* with host. Isolates at application level (processes, file system, network separated per container). Minimal overhead, but all containers must be compatible with the host OS kernel. Less secure isolation than VMs (since a kernel bug could potentially let a container escape), but usually sufficient and much more efficient.
- **Use Cases**:
- Emulators are used for running software for one architecture on another (e.g., old console games on PC, or Android emulator on x86) and for debugging (simulate hardware devices).
- VMs are used to consolidate servers (run many server OS instances on one machine), for testing different OS environments, and for cloud computing (cloud providers run your instances in VMs).
- Containers are used to deploy applications in a consistent environment; popular in DevOps for packaging microservices (Docker containers). They allow high density (many containers on one host, where each would be too small to justify a full VM).

- **Overlap**: Modern container platforms (like Kubernetes) might run containers inside VMs for extra isolation in cloud environments. Also, some hypervisors use CPU virtualization plus paravirtual drivers to improve performance.

There was no specific exam question on emulators/VMs/containers in the 2024 final, but understanding these concepts provides context for cloud computing (Week 10). In summary: an emulator imitates hardware in software; a VM provides a virtual hardware stack for full OS instances; a container isolates applications using the host OS. Each technology balances performance, isolation, and flexibility.

Week 4: Processes and File Systems

- **Processes vs Programs**: A *program* is the static code (on disk), while a *process* is that code in execution (in memory with resources)[51][52]. When you run a program, the OS loads the program's code and data into memory, sets up a process control block (with info like PID, owner, open files), and starts the program's execution. Thus, one program can be run by multiple processes simultaneously (each with its own state). *(Exam True/False Q22)*: "A program is stored on permanent storage (disk file)." – True[53], whereas a process lives in volatile memory when running.
- **Command-line Arguments**: When you run a program, you can pass arguments (e.g., `grep foo file1 file2`). In C or Python these appear in `argv`. In shell scripts, special variables hold them: `$0` is the script name, `$1` the first argument, `$2` second, and `$#` the number of arguments. *(Exam Q10)* asks how to get the count of arguments in a shell script – **`$#`** is the answer[54]. And *(Exam Q28)* asks for the first argument – **`$1`**. In Python, `len(sys.argv) - 1` would also give number of arguments (excluding script name).
- **Starting Processes in Shell**: Normally, the shell runs one command at a time (foreground). You can start a process in *parallel* (background) with & as discussed in Week 2[41]. The shell maintains a **job table** for background processes. Each background job has an ID ([%1], [%2], ...). You can use `jobs` to list them and `fg`/`bg` to move them to foreground/background.
- **System Calls for Process Management**: As noted, **fork**, **exec**, and **wait** are the primary calls. A typical pattern in the shell: the parent (shell) calls `fork()`. The child process then uses `exec()` to load the new program, while the parent uses `wait()` to wait for child to finish[33][36]. This creates a parent-child relationship. The child inherits things like file descriptors. For example, if you do `ls > out.txt`, the shell sets up the redirection (opens out.txt as fd1), then forks: the child inherits that fd for stdout and execs `ls`, so `ls` writes to fd1 (out.txt). Meanwhile, the parent waits. After `ls` finishes and exits, the parent resumes. This model underlies *every command execution* in Unix shells.
- **Python `os.fork()` and `os.exec()`**: In Python, you can directly use these system calls via the `os` module. The final exam had a short code (Exam Q34) with `os.fork()`

and `os.execl()`. Key behaviors: the code after fork runs in both processes, but after `os.execl()` (in the child), the new program replaces the child's code, and anything after the `execl` call in that branch will not execute. In Q34, the sequence leads to output "1 2 4" (child prints 1, exec's echo which prints 2, then parent after wait prints 4; the child's code after exec (print 3) never runs)[55][56]. This demonstrates the **fork-exec-wait** pattern and how exec does not return if successful.

- **Inter-process Communication (IPC)**: While not deeply covered in lecture, processes often need to communicate. One basic form is using pipes (`|`) in shell (which we saw). Another is via special files or sockets. Week 4 mentions *devices in the namespace* and *pipes* as special "files". In Unix, **everything is a file**: hardware devices (like `/dev/tty`), pipes, etc., appear in the file system. For instance, **named pipes** (FIFOs) are created with `mkfifo` and can be written to and read from by separate processes. Also, `/dev/null` is a special file that discards data written to it. The file abstraction is powerful – the shell used it to implement I/O redirection (writing to `>` a file or `2>` to a file uses the same mechanism whether the "file" is a regular file or a device).
- **File System Hierarchy**: Unix-like systems have a single unified directory tree. The **root** `/` is the top. Devices are found under `/dev`, user home directories under `/home`, system config in `/etc`, temporary files in `/tmp`, etc. The lecture highlighted **mounting**: attaching a filesystem (like a USB drive or another disk partition) at a directory mount point[57][58]. For example, if you plug in a USB drive, the OS might mount it at `/mnt/usb` or `/media/username/USB`. The `mount` command associates a device with a location in the namespace. Conversely, `umount` detaches it. This allows multiple file system types to coexist in one hierarchy.
- **File System Implementation**: A file system like ext4 or NTFS organizes how files are stored on disk. It keeps tables of inodes (metadata about files) and data blocks. Concepts include:
- **Inode**: data structure holding file metadata (permissions, size, pointers to data blocks, etc.). Filenames are stored in directories linking name -> inode. Thus, multiple names (hard links) can point to one inode.
- **Path resolution**: The OS parses paths (`/dir/subdir/file`) by traversing directories (which are special files listing name-inode mappings).
- **Permissions**: Each file has permission bits (rwx for owner, group, others) and an owner UID and GID. The OS checks these when opening files. For example, if a file is `rw-r--r--`, the owner can read/write, others can only read.
- **ext4 and other FS**: ext4 is a widely used Linux file system (successor of ext3/ext2), supporting journaling (for reliability). Windows uses NTFS, FAT32/exFAT on flash drives. macOS uses APFS. The exam identified ext4 specifically as a Linux FS *(Exam Q14: ext4 is a Linux filesystem)*[59]. Knowing ext4 belongs to Linux and not, say, Windows, was expected.
- **Symbolic Links**: A **symbolic link** (symlink) is a special file that points to another file or directory by name. It's like a shortcut. When you open a symlink, the OS follows it

to the real target. Symlinks can even point to directories or across file systems. They are created with `ln -s target linkname`. (In contrast, a *hard link* is another name for the same inode; symlinks have their own inode and just refer to a path.)

- **Special Files**: Unix represents many things as files: devices (e.g., `/dev/sda` for a disk, `/dev/tty` for terminal), the proc filesystem (`/proc` has pseudo-files representing processes and system info), pipes, etc. For instance, reading from `/dev/urandom` gives random bytes; writing to `/dev/tty` writes to your terminal. This unification simplifies I/O – programs don't need to know if an input is from a file, device, or pipe; they just read bytes.

Key Exam Connections: This week's material underlies several exam questions. The concept "instance of a program running is a process" (Exam Q9) we already covered. The idea that command-line args in shell are accessed via `$#`, `$1` was tested (Q10, Q28). Understanding of background jobs led to Q29 about `fg`. On file systems, Q14 tested recognition of ext4 as a Linux file system. Also, knowledge of the file descriptor numbers (Q35) and usage of `cat` for file concatenation (Q30: e.g., `cat file1 file2 > file3`)[60] was needed. While the details of inode or mounting weren't explicitly in the final, they deepen understanding of how commands like `ls -l` or `mount` work. It's also useful for systems programming tasks (like the assignment on building a shell, perhaps). Overall, mastering process creation and file system structure is crucial for systems operations.

Week 5: Memory Management, Scheduling, Bootstrap, and Shell Scripting

Part I: Memory Management

- **Process Memory Layout**: When a program becomes a process, the OS gives it memory divided into sections[61][62]:
- **Text (Code)**: contains the program's compiled instructions.
- **Data**: for global and static variables. (Often split into initialized data and BSS for uninitialized.)
- **Heap**: dynamic memory region where malloc/new (or Python objects) are allocated. Grows upward.
- **Stack**: for function call frames and local variables. Grows downward.
  Each process has its own address space structured like this. For example, the stack starts at a high address and extends down, while heap starts near the data segment and grows up towards the stack[61][63]. This is why an out-of-control recursion (stack overflow) or unchecked heap growth can run into each other.

- **Simple (Base/Limit) Memory Mapping**: Early multiprogramming systems used a **base register** and **limit register** for each process[64][65]. The idea: the OS/CPU

maps a process's **virtual address 0** to a **physical address = base value**, and disallows addresses >= base+limit. So each process "sees" a memory starting at 0, but actually it's offset in physical RAM. The **base register** is added to every memory reference (on the fly by hardware), and the **limit** is the max offset allowed. This provides isolation: a process cannot address memory outside its allocated range. For example, if base=1000 and limit=5000, then the process's address 0 is physical 1000, and it can access up to physical 5999. An address reference to 5000 would trap (since 5000 >= limit).

- **Virtual Memory and Paging**: Modern OSes use **paging** to manage memory more flexibly[66][67]. Physical memory is divided into frames (e.g., 4KB each), and process memory into pages of the same size. A **page table** is maintained per process, mapping virtual page numbers to physical frame numbers (or to a disk location if not in RAM)[68]. This provides:
- **Isolation**: each process has its own mapping, so they can use the same virtual addresses without conflict.
- **Efficient allocation**: non-contiguous physical memory can appear contiguous to the process.
- **Demand paging**: a page not in physical memory causes a **page fault**; the OS can load the page from disk (swap space) into a free frame, update the page table, then resume the process[69][70]. This allows programs to use more memory than physically available, storing rarely-used pages on disk ("virtual memory"). *(Exam Q26)* was about extracting bits from an instruction, not directly VM, but understanding bit fields relates: e.g., page number vs offset bits in an address.
- **Page Replacement & Swapping**: If memory is full and a new page is needed, the OS may **swap out** (write to disk) some page to free a frame[71][72]. If this happens too frequently, it leads to **thrashing** (performance collapse due to constant disk I/O). Thus, OSes use algorithms (LRU, etc.) to decide which pages to evict and try to keep frequently used pages in RAM.
- **Shared Memory**: Some pages can be marked as shared across processes[73]. For instance, multiple processes running the same program might share the code pages (mapped into each process's memory but using one physical copy marked read-only). This saves memory. Also, processes can use shared memory segments for IPC to communicate by reading/writing the same memory (explicitly set up via OS APIs).
- **Memory Protection**: Virtual memory inherently provides protection – a process cannot access memory that isn't mapped in its page table (and it can't arbitrarily modify the page table as that's privileged). Additionally, page table entries have permission bits (read/write/execute). For example, the OS can mark code pages as non-writable (so if a process tries to modify its own code, it faults) and data pages as non-executable (to prevent injecting code into data, mitigating some security attacks).

Part II: Process Scheduling

- **The Scheduler**: When multiple processes are runnable, the OS **scheduler** decides which one to run on the CPU next[74]. It employs a scheduling algorithm (Round Robin, etc.) to share CPU time fairly or by priority. In a simple system, processes are kept in a **ready queue**. The scheduler selects the next process when the current one blocks (e.g., waiting for I/O) or its time slice expires (on a timer interrupt). This context switch involves saving the CPU state of the current process and loading the state of the next.
- **Process Priority and Niceness**: Some OSes assign priorities to processes. Higher priority processes get more CPU time. In Unix, **niceness** is a user-influenced priority value (range -20 to +19, where *lower* nice means higher priority). By default, processes start with nice=0. You can lower the value (make it "more negative") to ask for more CPU share or increase niceness to be polite (less CPU).
- The `nice` command can start a process with a given niceness, e.g., `nice -n 10 myprog` starts it with nice 10 (lower priority).
- The `renice` command changes the niceness of an existing process, e.g., `renice -n -5 -p 1234` makes PID 1234 higher priority (niceness -5).
  (Exam Q20) asked: which command increases priority of PID 54923? Since lower niceness = higher priority, you need to **decrease** the nice value. `nice -n -10 -p 54923` would start a new process with -10 (if it accepted -p, but typically `nice` doesn't target existing PIDs), whereas `renice -n -10 -p 54923` is correct to adjust an existing process to -10 niceness[75][76]. The answer was likely **`renice -n -10 -p 54923`** *(Exam Q20)*. Also, *(Exam Q24)*: "A larger niceness value causes a higher priority." – This is **False**, since higher nice means *lower* priority[77][78].
- **Foreground vs Background Processes**: We covered job control earlier. From a scheduling perspective, background jobs are simply processes not interacting with the terminal (stdin/out). The OS doesn't inherently schedule them differently, but the shell by default gives them lower priority by increasing their niceness when you use &. This ensures your interactive use is smooth. (The lecture specifically mentions using `nice` for background jobs).
- **Process States Revisited**: A running process may become **blocked** waiting for I/O, in which case the scheduler picks another process. Many systems employ a preemptive round-robin: each process runs for a time quantum (say 100ms) then is preempted so another can run. This gives the illusion of parallelism on one CPU.

Part III: Bootstrapping (System Boot Sequence)

- **Bootloader and Boot Sequence**: When a computer is powered on or reset, it goes through a series of steps to load the OS, because the OS itself is stored on disk. The process is:
- **BIOS/UEFI**: The Basic Input/Output System (on older PCs; or UEFI on modern ones) is firmware that runs first. It initializes hardware and selects a boot device (disk,

USB, etc.). It then loads the first stage bootloader from a fixed location (like the Master Boot Record of a disk).

- **MBR and Bootloader**: The Master Boot Record (MBR) is the first 512 bytes of the boot drive. It contains a small bootloader program. This bootloader may load a more sophisticated second-stage loader. Commonly, **GRUB** (Grand Unified Bootloader) is used on Linux systems[79]. *(Exam True/False Q23)*: "GRUB is involved in the boot sequence" – True. GRUB resides either in MBR or in a partition, and it presents a menu of OS choices.
- **GRUB/Bootloader stage 2**: The bootloader's job is to load the OS kernel from disk into memory. GRUB knows filesystem formats, so it can load `/boot/vmlinuz...` (the Linux kernel file) and an optional initial RAM disk (`initrd`). It then hands off execution to the kernel.
- **Kernel Initialization**: The OS kernel (e.g., Linux) starts running. It initializes core subsystems (memory manager, scheduler, device drivers). It will then spawn the first process (on Linux, traditionally this is PID 1, `init` or systemd in modern distros).
- **OS Startup**: The `init` process runs startup scripts/services to bring the system to an operational state (starting daemons, configuring network, etc.). Eventually, a login prompt or GUI appears.
- **Bootstrap Summary**: *BIOS -> Bootloader (MBR) -> GRUB -> Kernel -> init/system processes*. It's a chain of trust and functionality: each stage loads the next. This was only briefly covered, but the key terms: BIOS (or UEFI), MBR, GRUB, kernel, OS are in sequence.

Part IV: Shell Scripting (Advanced)

By week 5, students are expected to write more complex shell scripts, so more advanced techniques and best practices were highlighted:
- **Handling Command-line Arguments**: As mentioned, `$#` gives count, `$0` the script name, `$1`…`$N` the args. A script should check for the right number of args. Example pattern:

```
if [ $# -lt 2 ]; then
    echo "Error: too few arguments" >&2
    exit 1
fi
```

Here we send the error to stderr using `>&2` and exit with code 1 (non-zero exit statuses indicate error). *(Exam Q36)* expected students to validate argument count and print an error to stderr with exactly that format[80][81].
- **Loops in Scripts**: For tasks like printing a range of numbers (Exam Q36 scenario) or scanning files (Exam Q37 scenario), you'd use loops. - *Example*: Print all numbers between X and Y:

```
if [ $1 -le $2 ]; then
    for ((i=$1; i<=$2; i++)); do echo $i; done > output.txt
else
    for ((i=$1; i>=$2; i--)); do echo $i; done > output.txt
fi
```

This uses a C-style for loop (Bash extension) and redirects output to a file. The logic chooses increasing or decreasing order based on the argument values.
- Another example is iterating over files in directories (for Q37, finding the largest numbered jpg in subdirectories). One might use a loop like:

```
for dir in "$1"/*/; do
    max=0
    maxfile=""
    for img in "$dir"*.jpg; do
        # extract number from filename
        name=$(basename "$img" .jpg)
        if [ "$name" -gt "$max" ] 2>/dev/null; then
            max=$name; maxfile="$img"
        fi
    done
    [ -n "$maxfile" ] && cp "$maxfile" "$1"
done
```

This script loops through each subdirectory of the given directory, finds the largest numeric filename, and copies it to the parent directory. It uses `basename` to strip the directory and extension, then numeric comparison. (This is one approach; other solutions might use `sort -n` or similar to avoid inner loops.) The exam expected an understanding of loop and numeric comparison (and possibly hint usage of parameter expansion to strip extensions, as given in the hint about `${filepath##*/}` or so).
- **Conditions and File Tests**: Shell `if` can use `[ ... ]` or `[[ ... ]]`. Some useful file test operators: `-e file` (exists), `-d file` (is directory), `-f file` (is regular file). In Q37, one might check that exactly one argument was passed (`if [ $# -ne 1 ]; then ... error ...; fi`) *(Exam Q37 requires checking argument count)*. Or ensure the argument is a directory: `if [ ! -d "$1" ]; then echo "Error: not a directory" >&2; exit 1; fi`.
- **Arithmetic in Shell**: Within double-parentheses `(( ))`, you can do arithmetic without needing `expr`. Also, `(( var++ ))`, `(( i += 1 ))` are allowed. In test brackets, you use `-lt`, `-gt`, `-eq` for numeric compare (or within `(( ))` you can use standard `>`, `<`, `==`).
- **Error Handling**: As shown, sending messages to stderr and exiting with a non-zero code is important for scripts (so that other programs know your script failed). The exam tasks (Q36, Q37) explicitly required printing errors like “Error: too few arguments” to stderr (hence `>&2`) and exiting. This demonstrates good practice in script writing.
- **Shebang and Execution**: Most scripts start with a shebang `#!/bin/bash` to indicate which interpreter to use. And you must make the script executable (`chmod +x script.sh`)

or run it via `bash script.sh`. The content of week 5 assumes familiarity with this from earlier or labs.

Key Exam Connections: This week tied together many earlier concepts into practical tasks. Q20 and Q24 on scheduling we covered (nice vs priority). The boot sequence understanding was tested by Q23 (GRUB in boot). The shell scripting skills were tested heavily in written coding questions Q36 and Q37. For instance, in Q36 (writing a script to output a range of numbers to a file), one needed argument count checks, numeric comparisons, loops, redirection (> vs >> usage as hinted)[81]. In Q37 (finding max numbered jpg in subdirectories), one had to iterate directories, parse filenames (hint given about removing path and extension)[82], and copy files – demonstrating understanding of loops, string operations, and file handling in shell. Those who understood the lecture and practiced shell scripting would recognize patterns to solve these. Overall, Week 5 combined OS theory (memory, scheduling, boot) with practical shell scripting, making it a comprehensive and crucial week for the exam and assignments.

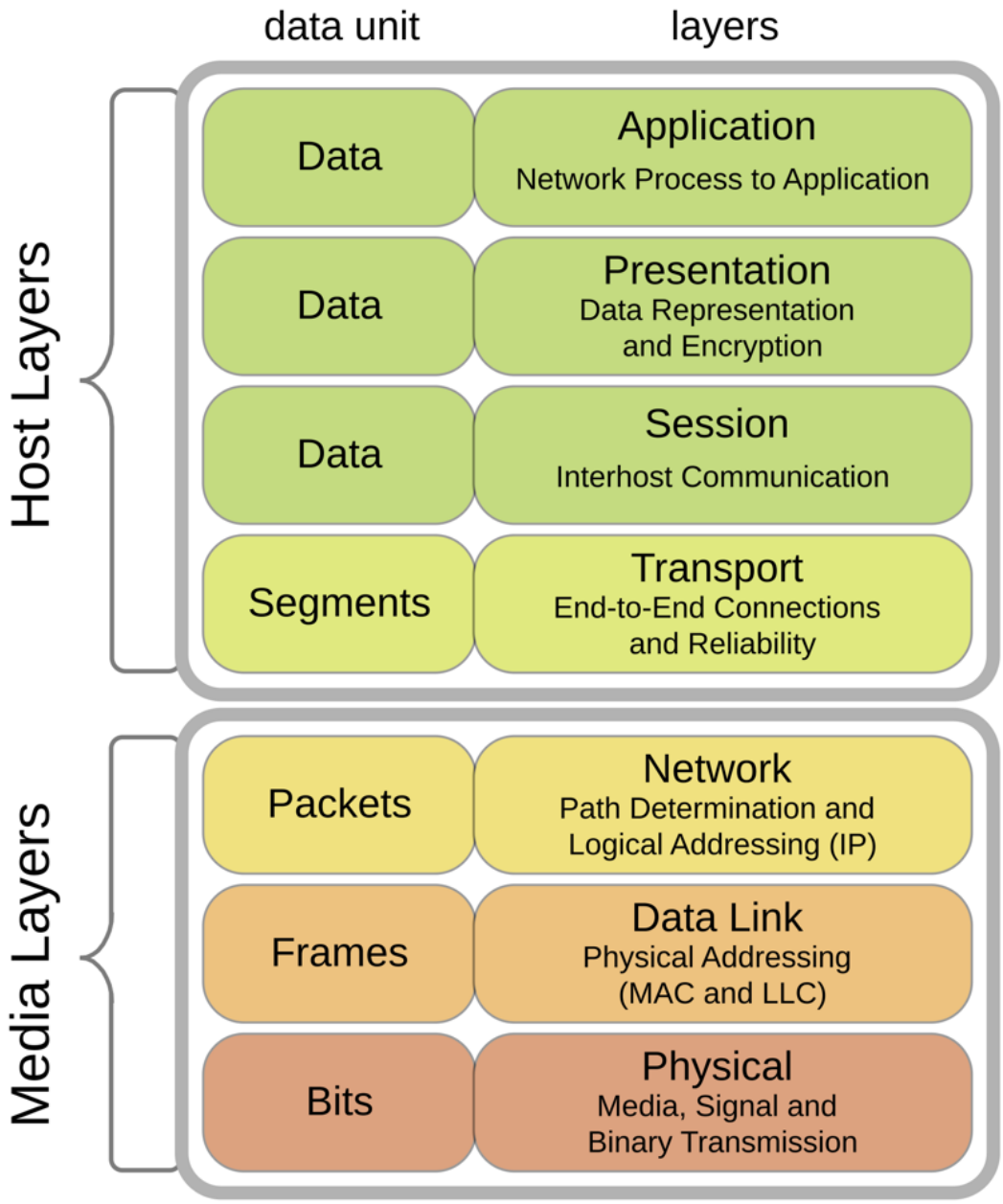
Week 6: Introduction to Networks & Switching

- **Networking Basics**: A computer network is a collection of interconnected nodes (computers, routers, etc.) that communicate over some medium. Fundamental questions: How do we structure this communication? Early network designs borrowed from the telephone system (circuit switching) but evolved to the Internet's packet switching for efficiency.
- **Circuit Switching vs Packet Switching**:
- **Circuit Switching**: Establishes a dedicated communication path (circuit) between parties for the duration of the session[83][84]. Classic telephone networks did this – if you call someone, a route with reserved bandwidth is maintained until you hang up. Advantage: guaranteed bandwidth and consistent delay once set up[85]. Disadvantages: setup time, and if you're not using the channel fully, the reserved bandwidth is wasted[86]. If a circuit fails, call is dropped (no automatic rerouting)[87].
- **Packet Switching**: Messages are broken into **packets** (small chunks, e.g., around 1500 bytes) which are sent independently through the network[88]. Each packet carries destination info and is routed individually, possibly via different paths[89]. At the destination, packets are reassembled into the original message[89]. This approach is used by the Internet. It's more efficient as many "conversations" can share the same links, and if one path is congested or broken, packets can be dynamically routed around it[90][91]. However, packets can experience variable delay, may arrive out of order, or get lost, so higher-level protocols must handle that. The Internet sacrificed the guaranteed timing of circuit switching for robustness and efficiency. *Store-and-forward*: each router receives the whole packet before forwarding[92], which introduces per-hop delay proportional to packet size and link

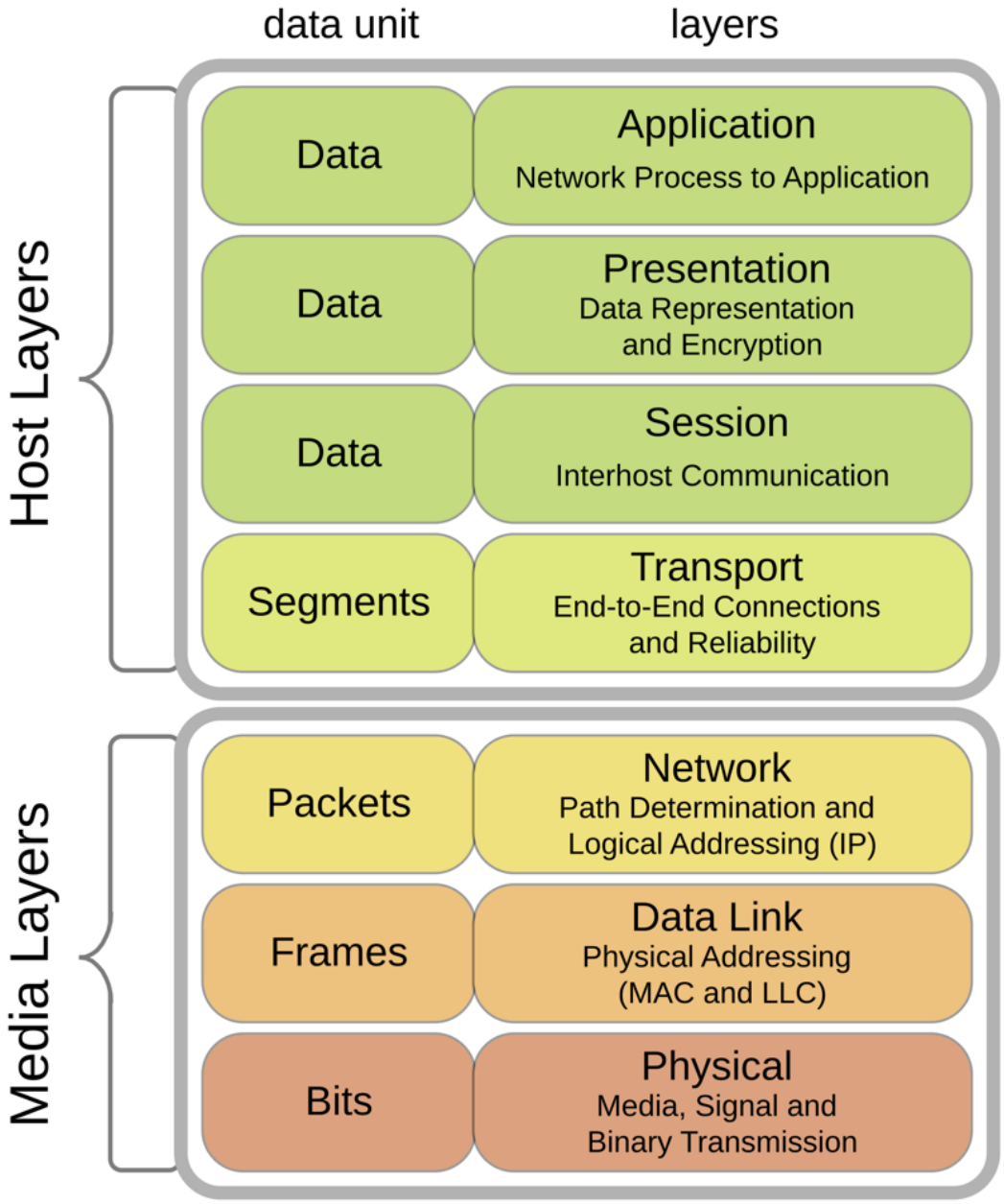
speed[93]. Example: 7.5 Mbit packet on 1.5 Mbps link takes 5 seconds to transmit one hop[94].

- **Network Structure – LANs and WANs**:
- **LAN (Local Area Network)**: a network covering a small area (home, office, campus). High speed, technologies like Ethernet or Wi-Fi are common. Typically owned by one entity.
- **WAN (Wide Area Network)**: spans large distances (cities, countries). Lower speed (generally) and often composed of interconnected LANs via routers. The Internet is a global WAN composed of many networks.
- **Ethernet (Link Layer)**: Ethernet is the dominant LAN technology. It's a **packet-switched** system at the link layer (Layer 2). Each Ethernet interface has a **MAC address** (48-bit) that uniquely identifies it[95]. Ethernet frames have source and destination MAC addresses. On wired Ethernet, devices share a medium; classic Ethernet used CSMA/CD (an algorithm to handle collisions on a shared coax). Modern Ethernet usually uses switches so collisions are rare. Wi-Fi is similar at link layer but uses radio (with its own collision avoidance mechanism).
- **Frames and Packets**: Lecture indicates that an "Internet packet" (IP packet) can be carried inside an Ethernet frame[96]. So a packet goes from host to switch to router, etc., and at each hop it's encapsulated in that link's frame. For instance, on a Wi-Fi hop, the IP packet is inside a Wi-Fi frame; on a fiber link, inside maybe PPP or other framing.
- **Addressing**: MAC addresses are used on a LAN so that a frame reaches the correct host. But they are not used for global delivery – that's where IP addresses come in (network layer).
- **Network Layer & IP**: The **Internet Protocol (IP)** (Layer 3) handles routing packets across networks[97][98]. Key aspects:
- **IP Packets**: They have an IP header (with source and destination IP addresses, etc.) and payload (often a TCP/UDP segment). IPv4 addresses are 32-bit (written as four octets, e.g., 129.78.8.1)[99]. IPv6 addresses are 128-bit (written in hexadecimal colon-separated format). An IP address identifies a host (or more specifically, a network interface on a machine) on the global network *(Exam Q18: An IP address identifies a machine (or host) in a network)*[100].
- **Routing**: Routers forward IP packets towards their destination. Each router has a **routing table** that tells it which next-hop to use for a given destination network[101][97]. If a packet is destined for the local network, it is delivered directly; if not, it goes to a router (gateway).
- **Error Detection**: IP and lower layers have mechanisms to detect errors (like checksum in IP header, CRC in Ethernet frame)[102]. If a packet is corrupted, it can be detected and dropped (higher layers might retransmit if using TCP).
- **ICMP**: Internet Control Message Protocol, used for network diagnostics (e.g., ping uses ICMP Echo Request/Reply).

- **Transport Layer (TCP/UDP)**: Sits above IP to provide process-to-process communication.
- **UDP (User Datagram Protocol)**: A simple protocol on top of IP. It provides multiplexing via **ports** (16-bit numbers identifying source/dest application endpoints), but it's unreliable and connectionless (no guarantees of delivery or order). Useful for simple query/response and streaming where speed matters more than reliability (e.g., DNS, video streaming). *(Exam Q17)*: DNS queries typically use UDP[103][104], because it's simple and one packet each way usually. If the response is too large or if reliability is needed (zone transfers), DNS can also use TCP, but primary DNS lookup is UDP on port 53.
- **TCP (Transmission Control Protocol)**: A reliable, connection-oriented protocol. It establishes a connection (3-way handshake), ensures ordered delivery of bytes, retransmits lost packets, controls flow to not overwhelm receiver, and avoids congestion. It's used for most applications where data integrity is crucial (HTTP, SMTP email, etc.). TCP and UDP use **ports** to allow multiple connections on one host (e.g., port 80 for web, 22 for SSH, etc.). There are 65,536 possible port numbers (16-bit) – *(Exam Q4)*: "Port is 16-bit, how many possible?" Answer: **65536**[105].
- **Network Layering Model**: We have seen layers: **Application (layer 7)** – e.g., HTTP, DNS; **Transport (layer 4)** – TCP/UDP; **Network (layer 3)** – IP; **Data Link (layer 2)** – Ethernet/WiFi; **Physical (layer 1)** – bits on wire. The Internet (TCP/IP) model combines OSI layers 5-7 into one Application layer, so it's often described as 5-layer or 4-layer model[106][107]. Below is a diagram of the OSI 7-layer model and its data units


data unit
layers
Host Layers
Media Layers
Data
Application
Network Process to Application
Data
Presentation
Data Representation
and Encryption
Data
Session
Interhost Communication
Segments
Transport
End-to-End Connections
and Reliability
Packets
Network
Path Determination and
Logical Addressing (IP)
Frames
Data Link
Physical Addressing
(MAC and LLC)
Bits
Physical
Media, Signal and
Binary Transmission

:



OSI 7-Layer Model: Application (data), Presentation (data), Session (data), Transport (segments), Network (packets), Data Link (frames), Physical (bits). Internet's TCP/IP model merges Presentation and Session into Application, and often treats Data Link+Physical together as Network Access. The "Host Layers" (Application to Transport) deal with end-to-end data; "Media Layers" (Network to Physical) deal with delivering packets over the network infrastructure.

At each layer, information (headers) is added (encapsulation). For example, an HTTP request (Application data) gets a TCP header (becoming a segment), then an IP header

(packet), then an Ethernet header (frame) before going out on the wire. Each layer's header is stripped off at the corresponding layer on the receiver.

- **Local Networking**: On a LAN, how do packets get to the right host? IP relies on ARP (Address Resolution Protocol) to map IP addresses to MAC addresses. If Host A (IP 10.0.0.5) wants to send to 10.0.0.7 on the same LAN, it sends an ARP query "Who has 10.0.0.7? Tell 10.0.0.5". Host B with 10.0.0.7 replies with its MAC. Then A sends the IP packet encapsulated in an Ethernet frame to B's MAC. ARP is vital for local delivery on Ethernet.
- **Switches vs Routers**: A **switch** operates at Layer 2 (Ethernet). It learns MAC addresses and forwards frames only to the port where the destination is connected (unlike a hub which broadcasts to all ports). Switches make LANs more efficient and segmented. A **router** operates at Layer 3 (IP), connecting different networks and routing packets. A home router typically connects your LAN (say 192.168.1.0/24) to the ISP's network, performing Network Address Translation (NAT) as well (more on that in Week 7).

Key Exam Connections: From this introductory week, some straightforward questions appeared: e.g., conversion of an IPv6 address length (Exam Q2: IPv6 uses 128 bits = 16 bytes)[108], which one should know from the fact IPv6 addresses are 128-bit. Q18 about what an IP address identifies – a host on the network, not an application or process *(correct answer: machine in a network)*[100]. Also, Q25 (T/F): "Ethernet frame format varies with physical layer (wired vs wireless)" – The statement is a bit tricky; the basic Ethernet frame structure is consistent, but wireless 802.11 frames have some differences in header fields. Likely the expected answer was **False**, because Ethernet (802.3) and Wi-Fi (802.11) are different link protocols, not just a variation of one frame format[109]. The specifics of that question aside, understanding that Ethernet and Wi-Fi are distinct link layers is important. No direct exam question on circuit vs packet switching, but the concept underpins the design of the Internet (reliability and redundancy vs guaranteed QoS). This week's content sets the stage for deeper network topics in Weeks 7 and 8.

Week 7: The Internet Protocol and Routing

- **Internet Packets and IP**: We focus on the Network layer (IPv4/IPv6) and how packets get from source to destination across multiple networks. Each IP packet has: source IP, destination IP, TTL (time-to-live to avoid infinite loops), protocol field (to indicate if it's TCP, UDP, etc inside), header checksum (for error detection), and other info[97][102]. Data is encapsulated after the header. The maximum size of an IP packet (IPv4) is 65,535 bytes (16-bit length field)[110], but networks have smaller MTUs (Maximum Transmission Units), so large packets are fragmented into smaller ones.

- **TCP vs UDP vs ICMP**: The exam asked a question about identifying protocols: e.g., "What does an IP address identify?" (a host interface) and a question about which protocol DNS uses (UDP) we saw in Week 6. It also had a multi-choice possibly about what some protocol is (maybe asking something like "RSA is an example of: a) symmetric algorithm b) asymmetric algorithm c) Ethernet protocol d) …" which was Q19 – that's from Week 11 security: RSA is asymmetric crypto). But relevant here, know that **TCP (connection-oriented, reliable)**, **UDP (connectionless, unreliable)**, **ICMP (control messages)** are key Internet layer or just above.
- **Transport Ports**: Ports differentiate multiple services on one host. E.g., HTTP is typically port 80, HTTPS 443, SSH 22, DNS 53, etc. Ports are 16-bit, so as mentioned 65536 possibilities (0–65535). Ports < 1024 are "well-known" (reserved for system/root on many OS).
- **Routing**: Each router examines the destination IP of a packet and forwards it according to its routing table. The routing table has entries like "For network X use next-hop Y via interface Z". A router has interfaces (each with an IP in the respective subnet). When a packet arrives, it checks for the longest prefix match in the table (i.e., most specific network containing the destination). E.g., a router might have:
- 10.0.2.0/24 via 10.0.5.1 (means any IP 10.0.2.* send to router 10.0.5.1 next)
- 0.0.0.0/0 via 10.0.5.254 (default route for everything else)
- **LAN, WAN, and Routing**: Often within a LAN, hosts have an IP and a netmask defining their network range. If dest IP is in same range, host uses ARP and delivers directly. If not, it sends to its **gateway** (router). The router then forwards further. This hop-by-hop continues until a router finds the destination on one of its directly connected networks or one hop away.
- **Address Resolution Protocol (ARP)**: We touched on ARP last week; specifically: an IP packet on an Ethernet needs a MAC destination. If the next hop IP is known (either the final dest if local or the gateway), the sender must find the MAC for that IP. ARP query: "Who has IP X? Tell IP Y." and ARP reply: "IP X is at MAC M". Entries are cached in an ARP table to avoid excessive broadcasts. ARP is only for IPv4. IPv6 uses a similar concept called Neighbor Discovery (ND) which uses ICMPv6.
- **Routing Example**: **Traceroute** is a tool that shows the path (sequence of router hops) to a destination. It works by sending packets (UDP or ICMP) with increasing TTL values and listening to ICMP "Time Exceeded" messages from routers, thereby identifying routers on the path.
- **NAT (Network Address Translation)**: Because IPv4 addresses are limited, many networks use private IP ranges (like 192.168.x.x, 10.x.x.x). A **NAT router** translates private IPs to a shared public IP for internet access. For example, your home router (with one public IP from ISP) will rewrite outgoing packets: source address from your private IP (say 192.168.0.2) to the router's public IP, and track the connection (source port too) so that when a reply comes back, it maps it to the right internal IP/port. NAT is prevalent – almost every home uses it. One downside is it breaks direct incoming connections (unless you set port forwarding). It's a sort of "hack"

that extended IPv4's life. IPv6 largely removes the need for NAT, since address space is huge.

- **IPv6**: Next-gen IP with 128-bit addresses. Represented in hex like `2001:0db8:85a3::8a2e:0370:7334`. It provides vastly more addresses and simplifies some aspects (no ARP – uses ND, no fragmentation by routers – hosts should use Path MTU discovery, mandatory support for IPsec encryption, etc.). **IPv6 address example**: given something like `2001:8003:c41c:d000:84d4:8756:171f:7add`, how many bytes? It's 128 bits / 8 = **16 bytes** *(Exam Q2)*[108].
- **ICMP and Utilities**:
- **ping** uses ICMP Echo request to check if a host is reachable and measure latency.
- **traceroute** (Unix) or `tracert` (Windows) uses increasing TTL to map routes.
- **ipconfig/ifconfig** shows network config, `route -n` or `ip route` shows routing tables.
- **UDP/TCP and Applications**: Some specifics:
- **DNS** as noted uses UDP (and sometimes TCP for large responses like zone transfers). *(Exam Q17)*: DNS lookup uses UDP[111] – answer: UDP (with IP of course as underlying network layer).
- **HTTP** uses TCP (usually port 80).
- **SMTP email** uses TCP (port 25).
- **SSH** uses TCP (22).
- **Streaming protocols** or real-time like VoIP often run over UDP (with additional protocols to handle loss).
  The exam's multiple-choice might include something like "Which of these is an Internet layer protocol? a) IP, b) TCP, c) UDP, d) HTTP" – correct is IP (TCP/UDP are transport, HTTP is application). Or "Which protocol provides reliable transport? a) IP, b) UDP, c) TCP, d) Ethernet" – answer TCP. These weren't explicitly in the provided exam sections but are typical.

Key Exam Connections: Q2 (IPv6 length) and Q18 (IP address identifies a machine) directly come from this content[108][100]. Q17 (DNS uses UDP) was explicitly answered in lecture slides[104]. Q4 (number of ports) was a general knowledge: 2^16 = 65536[105]. Q25 (Ethernet frame format differs on wired/wireless) – likely intended false, because Ethernet is defined for wired; wireless has a different frame structure (802.11), but if interpreting "Ethernet frame" strictly, there is only one format used on Ethernet links regardless of cable or fiber, so the statement is false that it varies by physical medium[109]. Understanding NAT and private vs public IP wasn't directly quizzed in 2024 but is crucial for real-world networking.

In summary, Week 7 teaches how data finds its way through a complex network: from ARP on local link to global routing with IP, plus the roles of TCP/UDP. It sets the stage for Week 8's focus on higher-level protocols like DNS and HTTP.

Week 8: DNS and Application Layer Protocols

- **Domain Name System (DNS)**: DNS is a distributed database that maps **domain names** (like `www.sydney.edu.au`) to IP addresses[112][113], and vice versa (for reverse DNS). Key points:
- **Hierarchy**: DNS names are hierarchical, read right-to-left: host.subdomain...domain.TLD. For example, `www.sydney.edu.au`: "au" is the country Top-Level Domain, under that "edu" (second-level domain for education in Australia), under that "sydney" (institution's domain), and "www" is a host within that domain[114][115]. The DNS namespace is a tree with the root "." at top[116][117]. The root has name servers that delegate to TLD servers (for .au, .com, etc.), which delegate to second-level, and so on.
- **Distributed and Redundant**: No single server has the entire database; it's split among millions of DNS servers. The *root servers* (13 sets) know the addresses of all TLD servers. Each domain (e.g., `sydney.edu.au`) has authoritative name servers responsible for answering queries about names in that domain[118][119]. This distribution avoids the centralized hosts.txt problem[120][121].
- **Name Resolution Process**: When a client needs to resolve a name, it typically asks a **recursive resolver** (often provided by the ISP or local network). That resolver will check its cache; if not found, it queries step by step: root -> TLD -> next-level -> ... authoritative server[122]. Finally, it returns the answer (and caches it). The resolution uses UDP on port 53 for queries/replies in most cases. The lecture explicitly says: "DNS protocol sends messages using UDP"[104] *(Exam Q17)*. If UDP responses are too large (>512 bytes historically, or > ~1232 bytes with EDNS0 extensions), it may use TCP.
- **DNS Records**: DNS stores various record types: **A** (IPv4 address for a name), **AAAA** (IPv6 address), **CNAME** (alias to another name), **MX** (mail exchanger for domain), **TXT** (text info), **NS** (nameserver for the domain), **PTR** (reverse lookup pointer). Example: an A record: `www.sydney.edu.au -> 129.78.xxx.xxx`. MX might say `sydney.edu.au -> mail.sydney.edu.au`. The DNS query can request a specific record type. The lecture likely mentioned some, e.g., A, MX, NS.
- **Caching and TTL**: DNS replies include a Time-to-Live. Resolvers cache records to reduce load. That's why changes to DNS may take time to propagate (you wait for caches to expire).
- **Application Protocols**: Built on TCP/UDP, define how application data is structured. The lecture covers a few common ones:
- **HTTP (HyperText Transfer Protocol)**: The web's protocol, typically over TCP port 80 (or 443 for HTTPS). HTTP is a request-response protocol: client (browser) sends a request (like `GET /index.html` along with headers) and the server responds with a status code and content. It's stateless per request (each request is independent,

though cookies and sessions add state at higher level). HTTP/1.1 introduced persistent connections (to reuse TCP for multiple requests). Modern web is moving to HTTP/2 and HTTP/3 for efficiency. The exam didn't directly ask about HTTP, but knowing that it uses TCP (reliable) because we can't have parts of a webpage missing.

- **SMTP, POP, IMAP (Email protocols)**: SMTP (Simple Mail Transfer Protocol) is used to send mail between servers (and from client to server). Works over TCP port 25. POP3 (110) and IMAP (143) are protocols for clients to retrieve email from servers. POP3 is simpler (download-and-delete by default), IMAP is more complex (keeps mail on server, allows sync/folders). Likely lecture briefly described sending an email (MUA -> SMTP to send to mail server -> that server SMTP to recipient's server -> recipient uses IMAP/POP to read). Not explicitly in exam, but classic knowledge.
- **FTP (File Transfer Protocol)**: An older protocol (TCP 20/21) for file transfers. Supports interactive login, listing directories, etc. It's now largely replaced by SFTP (SSH-based) or HTTP(s) for file download.
- **SSH (Secure Shell)**: Uses TCP port 22. Provides an encrypted connection for remote login and command execution. Also supports file transfer (scp, sftp) and tunneling. Based on public-key cryptography for authentication. We'll see more of encryption in Week 11.
- **SMTP/HTTP examples**: Possibly the lecture gave examples of protocol messages. E.g., an HTTP GET request example, or an SMTP "HELO" handshake. These are textual protocols (at least HTTP/1.1 and SMTP are textual commands). MIME (Multipurpose Internet Mail Extensions) might have been mentioned – it's how emails can have attachments and be multipart, or how HTTP can carry different content types.
- **DHCP (Dynamic Host Configuration Protocol)**: Mentioned in DNS context: "DHCP for address discovery"[123]. DHCP (uses UDP, ports 67/68) is how hosts get an IP address and network config automatically. When a device joins a network, it can broadcast "DHCP Discover", a DHCP server responds with offer including IP, gateway, DNS server, etc. DHCP was likely touched on as part of "how a computer knows where to send DNS queries" – typically via DHCP the router tells the client "use me as DNS server" or gives ISP's DNS.
- **DNS Security**: Not sure if covered here or in Week 11. Possibly mentioned that DNS has vulnerabilities (like spoofing, which led to DNSSEC designs).
- **Application ports and sockets**: Understand that when an application (like a web browser) uses TCP, it needs to specify a destination IP and port (e.g., IP of server and port 80). The OS will assign it a source port (ephemeral, usually >1024). The tuple (source IP, source port, dest IP, dest port) identifies a TCP connection uniquely. Similarly for UDP "conversations". The exam didn't ask directly, but Q4 about 16-bit port count we already saw.

- **Encryption in application protocols**: Possibly a note that protocols like HTTPS = HTTP over TLS (encryption). Similarly, there's SMTPS, FTPS, etc., but more likely this is covered in Week 11.
- **Summary Example**: Visiting `http://www.sydney.edu.au/`:
- Your PC checks cache or asks DNS for `www.sydney.edu.au` → gets an IP.
- It opens a TCP connection to that IP on port 80.
- Sends an HTTP GET request for "/" page.
- Server responds with HTML content.
- Closes connection (or reuses for more requests). Under the hood, many protocols worked: ARP (to reach default gateway), IP routing, TCP reliability, DNS resolution, etc., but the user just sees the webpage load.

Key Exam Connections: DNS was specifically tested (Exam Q17, and possibly indirectly Q2 with IPv6 example). Application protocols per se weren't directly questioned except maybe the port number question (Q4). However, Q19 (RSA is asymm crypto) is security. So, for Week 8 content, understanding DNS and general port usage is important. Also, Q13 in exam about grep was an app usage question (shell). Q12 about echo we did.

Given an example final question might be: *"When you type a URL, how is the name resolved to an IP?"* – The answer would involve DNS iterative resolution. Or *"True/False: DNS uses TCP for all queries"* – False (mostly UDP). They might also ask about DHCP's purpose (to assign IP config automatically), or an email question (like which protocol is used to retrieve mail – IMAP/POP).

In sum, Week 8 cemented how high-level applications rely on the lower layers. It demonstrates the client-server model: e.g., web browser (client) and web server (server) communicating via a defined protocol (HTTP). It also highlights the importance of DNS as the "phonebook" of the Internet.

Week 9 was omitted as per instructions (likely a break or revision week).

Week 10: Cloud Computing

- **What is Cloud Computing?** The term "cloud" refers to delivering computing services (servers, storage, databases, networking, software, etc.) over the internet on a flexible, on-demand basis[124]. In simple terms, instead of running everything on your local computer or a private data center, you use resources from providers like AWS, Azure, Google Cloud, etc. Key cloud characteristics: on-demand self-service, broad network access, resource pooling (multi-tenancy), rapid elasticity (scale up/down quickly), and measured service (pay-as-you-go).

- **Cloud Service Models**:
- **IaaS (Infrastructure as a Service)**: The provider offers virtualized computing resources like virtual machines, storage, networks. Users install and manage OS and applications. Example: Amazon EC2 (Elastic Compute Cloud) provides VMs to customers[125][126]. The user chooses how many CPUs, how much RAM, etc., and launches VMs. The cloud handles provisioning them on physical servers.
- **PaaS (Platform as a Service)**: The provider offers a platform (runtime environment, databases, etc.) where developers can deploy their applications without managing underlying OS or hardware. E.g., Google App Engine, Azure App Service. The scaling and infrastructure are abstracted away.
- **SaaS (Software as a Service)**: Full applications delivered as a service. Users just use the software via web or API; they don't manage the infrastructure or platform. E.g., Gmail, Office 365, Salesforce.
- **Cloud Providers**: Major ones mentioned[127] include **Amazon AWS** (pioneer and largest), **Microsoft Azure**, **Google Cloud Platform (GCP)**, IBM Cloud, etc. They started by using cloud architecture internally and then offering it to customers[128].
- **How Cloud Works (IaaS example)**: A provider runs huge **data centers** with many physical servers ("hypervisors"). Customers request VMs with certain specs (CPU cores, memory, storage, region) via a web interface or API[126][129]. The cloud's orchestration system finds a host with enough capacity and launches your VM there. Multiple VMs can share one physical host (with isolation through a hypervisor). If you request, say, 3 small VMs, they might all run on one physical server alongside others' VMs. If you request a very large VM, it might occupy an entire physical machine. Once the VM is running, you can SSH into it, install software, etc., as if it's a normal server. The user typically pays per hour of VM usage (or per second nowadays) and for any other resources consumed (bandwidth, storage).
- **Scaling and API**: A big advantage is you can **automate scaling**. Cloud providers offer APIs to create or terminate resources programmatically[130][131]. For instance, you can have a script or service that monitors load and when CPU usage > 80%, it calls `aws.create_instance(type='t3.micro', region='ap-southeast-2')` to spin up a new VM[132][133]. This is how auto-scaling works: the cloud can dynamically add more servers to handle increased traffic and then remove them when not needed, saving cost.
- **Cloud Services Examples**: AWS has dozens of services[134]:
- **Compute**: EC2 (VMs), Lambda (serverless functions), ECS/EKS (container services).
- **Storage**: S3 (Simple Storage Service) for object storage[135], EBS (block storage for VMs), Glacier (long-term archival storage)[136]. They mentioned **glacier** being cheaper but slow retrieval[136].
- **Databases**: Relational (RDS for MySQL/Postgres/Oracle, etc.), NoSQL (DynamoDB).
- **Networking**: Virtual Private Cloud (VPC) to isolate networks, Route53 (DNS service), API Gateway, load balancers, etc.

- **Security**: IAM (Identity and Access Management), KMS (Key Management Service) etc., and **security services** were mentioned[137] possibly including monitoring, DDoS protection.
- **Serverless Computing**: e.g., AWS Lambda – you just deploy a function, and the cloud runs it on demand without you managing servers[138]. This is often event-driven and scales automatically (Functions as a Service).
- **IoT & Edge**: Cloud providers have IoT services to handle data from devices, and **edge computing** which brings computation closer to users (for lower latency)[139]. Edge locations are often CDN (Content Delivery Network) nodes to cache content globally (like Amazon CloudFront).
- **Microservices Architecture**: The cloud encourages breaking applications into microservices that run in containers or as serverless functions. These microservices communicate over the network (often using APIs or messaging). It allows each part to scale independently and be developed in isolation. The mention of **Microservices**[139] ties into using containers (e.g., Docker + Kubernetes) and cloud services to deploy them.
- **Economic and Practical Impact**: Cloud computing lets startups or projects avoid large upfront hardware costs – you rent what you need. It also enables global reach easily (deploy instances in data centers around the world to serve users with low latency). It shifts spending to an operating expense model. However, one must design for failure (since in cloud, any instance can fail or be taken down – so you typically design redundant systems).
- **Cloud vs Traditional Hosting**: Traditional data center hosting might give you fixed servers that run 24/7. Cloud adds flexibility – can spin up 100 servers for an hour and then shut them down, for example, which is not feasible if you had to own those 100 physical servers.
- **Cloud Challenges**: Possibly mentioned briefly: security (you rely on provider's security; you must configure access properly – misconfigured S3 buckets are notorious leaks), vendor lock-in (apps tailored to one cloud's services might be hard to move), and the need for reliable internet connectivity.
- **Examples**: If an exam scenario described launching a web service, they might ask how to ensure it scales – answer: use cloud auto-scaling groups behind a load balancer, and possibly use a CDN for static content, use databases managed by cloud (like AWS RDS) for scaling and reliability, etc. Or maybe a question like "What is a benefit of cloud computing?" with answers like A) on-demand resources, B) one-time purchase of hardware (wrong), C) requires manual scaling (wrong).

Key Exam Connections: The final 2024 did not have an obvious direct question on cloud computing in the parts we saw. Possibly in written section, but nothing in MCQ or T/F that we saw except maybe background for one of the coding questions scenario? Unlikely. They might have expected an understanding in concept questions if any. Nonetheless, knowledge of cloud is increasingly relevant. It's possible a question like *"List one advantage and one risk of cloud computing"* could appear. Or *"Name an AWS service for*

storing large binary objects" – answer: S3. If they referenced the course content, they might mention AWS specifically since it was used as example a lot[140].

In summary, Week 10 broadens the view from individual systems to large-scale deployments. It ties together virtualization (from Week 3) and networking (Weeks 6-8) and introduces modern computing paradigms (serverless, microservices, edge). Understanding cloud platforms is crucial for practical computing today, though the exam emphasis appears to remain on fundamentals (OS, networks, security). It's a more conceptual and less technical/configuration-heavy week.

Week 11: Security – Cryptography and Signatures

- **Security Goals**: The lecture outlines key security objectives[141]:
- **Confidentiality** – keeping information secret from unauthorized parties.
- **Integrity** – ensuring data is not altered undetectably.
- **Authenticity** – verifying identity (knowing who sent a message) and **non-repudiation** – sender cannot deny sending a message. These align with using encryption, signatures, and hashes as tools. For example, web browsing needs confidentiality (passwords protected), integrity (no tampering), and authenticity (talking to the real bank website, not an impostor).
- **Encryption Basics**: **Cryptography** is the practice of scrambling (encrypting) a message so that only intended recipients can descramble (decrypt) it[142][143].
- **Plaintext (m)** – original message.
- **Ciphertext (E(m))** – encrypted message.
- **Key (k)** – a secret parameter used in the encryption/decryption functions. Without the key, decoding is infeasible. Carol (an eavesdropper) may see ciphertext but should not get plaintext without the key[144][145].
- The encryption function E and decryption D are inverses: D(E(m)) = m (with appropriate keys).
- *Example*: A simple cipher is **Caesar cipher** (shift letters), e.g., shift by 3: "HELLO" -> "KHOOR". This is easy to break, but illustrates using a key (shift amount) to transform plaintext[146]. Modern ciphers use mathematics and large keys to be secure.
- **Symmetric vs Asymmetric Cryptography**:
- **Symmetric (Secret-Key) Encryption**: The same key is used to encrypt and decrypt (k shared by Alice and Bob)[147]. Examples: AES (Advanced Encryption Standard), DES (old). It's very fast and suitable for bulk data encryption. Challenge: how do Alice and Bob share the secret key securely? Key distribution is non-trivial if they've never met. Symmetric algorithms are sometimes called **private key cryptography**. *(Exam Q19)* tried to confuse: RSA was given as an option "private key cryptography

algorithm", which is misleading terminology – RSA actually involves a private key, but it's generally termed asymmetric. The correct answer for RSA was "asymmetric key algorithm"[148]. Symmetric algorithms would be AES, etc.

- **Asymmetric (Public-Key) Encryption**: Uses a pair of keys – a **public key** (k_pub) and **private key** (k_priv)[149][150]. The public key can be shared openly, and the private key is kept secret by the owner (e.g., Bob). If Alice wants to send Bob a confidential message, she fetches Bob's public key and encrypts the message with it: c = E(m, k_pub,Bob). Bob uses his private key to decrypt: m = D(c, k_priv,Bob)[150][151]. This solves key distribution – no pre-shared secret needed; Alice just needs Bob's public key (which can be listed in a directory). But asymmetric crypto is computationally slower and often uses larger key sizes (e.g., 2048-bit RSA keys).
    - **RSA**: The most famous public-key algorithm (named after Rivest, Shamir, Adleman, 1977)[152]. RSA's security is based on difficulty of factoring large integers. It has the property that D(E(m)) = m and also E(D(m)) = m if you flip the keys (only true for some algorithms like RSA)[153]. This means RSA can be used for encryption (as described) and for **digital signatures** (more below). RSA is indeed an **asymmetric key algorithm** *(Exam Q19)*[148].
    - **Diffie-Hellman**: Not for encryption directly, but a protocol for key exchange. It allows two parties to agree on a shared secret key over a public channel without prior secrets. They exchange some computed values and each end up with the same key, which eavesdroppers can't derive (based on discrete log problem). This was revolutionary (first published 1976)[154]. We saw an illustration with mixing colors analogy[155][156]. Modern protocols (like TLS) often use Diffie-Hellman for establishing a session key securely (especially with ephemeral keys for forward secrecy).
- **Digital Signatures**: Provide authenticity and integrity. If Alice wants to prove she sent a message (and that it wasn't modified), she can **digitally sign** it. Using public-key crypto: Alice has a key pair (k_pub,Alice, k_priv,Alice). She uses her **private key** to sign a message (or usually, a hash of the message, to make it manageable). This produces a signature (some number). Anyone can then use Alice's **public key** to verify the signature is valid for that message[157][158]. Because only Alice has the private key, only she could have produced that signature. Thus it authenticates the sender and ensures integrity (if message changed, signature wouldn't verify). Non-repudiation comes because Alice cannot claim "I didn't send that" – her unique signature is proof.
- RSA can be used for signatures (sign by decrypting with private key, verify by encrypting with public key, conceptually)[153]. Other signature algorithms: DSA, ECDSA, etc.
- *Important:* Signing is different from encryption: encryption with public key ensures confidentiality (only holder of private key can read). Signing with private key ensures authenticity (anyone with public key can check). Often, both are used: e.g., in SSL/TLS, server's certificate is signed by a CA (so you trust you have the real

server's public key), and then you exchange a key using that to encrypt communications.

- **Hash Functions**: A **hash function** maps data of arbitrary length to a fixed-length digest, ideally in a way that is one-way (can't get original from hash) and collision-resistant (hard to find two different inputs with same hash)[159]. Examples: MD5, SHA-1 (160-bit), SHA-256, SHA-384, SHA-512 etc. Hashes are used for integrity: you can hash data and later hash it again to see if it changed (if hash differs, data changed). Also used in digital signatures (usually you sign the hash of a message rather than the whole message to save time).
- *(Exam Q3)*: "How many hexadecimal digits does SHA-384 digest consist of?" SHA-384 outputs 384 bits, which is 48 bytes. Each byte is 2 hex digits, so 48 bytes = 96 hex digits[160]. Indeed the correct answer was likely **96**. This tests knowing the size of hashes (SHA-384 is in the SHA-2 family: SHA-256 = 256-bit, 64 hex; SHA-384 = 384-bit, 96 hex; SHA-512 = 512-bit, 128 hex). The lecture snippet indicated SHA-256 example[159][161]. They likely mentioned SHA-384 in passing.
- Hashes are also used for storing passwords (store the hash, not plaintext password, so if DB leaks, attackers can't directly see passwords; they have to crack hashes).
- **Encryption in Network Security (TLS, HTTPS)**: The lecture gave an overview of how HTTPS works by layering encryption in the transport layer (TLS) under HTTP[162][163]. Basically, TLS (formerly SSL) sits between Application and TCP: the data is encrypted before going into TCP. When you see `https://`, your browser does a TLS handshake with the server: verifies server identity (certificate), exchanges keys (often using Diffie-Hellman or RSA), then uses a symmetric cipher (like AES) with that key to encrypt the HTTP traffic. This provides confidentiality and integrity for web browsing. The layering diagram in slides showed TLS below application, above TCP[164][165], indicating data flow with encryption.
- **Key Establishment and Certificates**: A big challenge: how do you know a public key really belongs to who you think? This is solved by **digital certificates** issued by Certificate Authorities (CAs). A certificate is basically "CA X attests that key Y belongs to website Z", signed by CA's private key. Browsers trust certain CA public keys. So when you connect to `bank.com`, the server presents a certificate. Your browser verifies the signature using CA's public key and domain name matches. If ok, you trust the server's public key to do the encryption handshake. This prevents man-in-the-middle with fake keys. If Carol tried to pose as bank.com, she wouldn't have a cert signed by a CA for that domain.
- **SSH, SCP**: Similar concept but typically uses a known host key (you accept a server's public key on first connect or from a trusted source) and then uses it to secure the session. Also uses symmetric encryption once a secret is agreed (because symmetric is faster).
- **Common Algorithms**:
- Symmetric: AES, DES (outdated), RC4 (outdated stream cipher). AES is standard (128 or 256-bit key).

- Asymmetric: RSA, Diffie-Hellman (for key exchange), Elliptic Curve Cryptography (ECC) which yields smaller keys for similar security (e.g., ECDHE for key exchange in TLS, ECDSA for signatures).
- Hash: SHA-2 family (SHA-256, etc.), older ones like SHA-1 (not secure against collisions now), MD5 (broken).
- Message Authentication Code (MAC): A symmetric key integrity check (like HMAC which uses a hash with a secret key) – ensures message came from someone who knows the key (provides integrity & authenticity in symmetric contexts). Possibly mentioned.
- **Real-world applications**:
- PGP for email encryption uses public-key crypto where users exchange public keys or use a "web of trust".
- Code signing: OS or apps verify digital signatures on software updates to ensure they're from the legitimate developer (authentic and untampered).
- Blockchain (not sure if mentioned): uses crypto primitives (signatures, hashes) heavily but might be beyond scope.

Key Exam Connections: Several directly. Q3 (SHA-384 hex length = 96)[166]. Q19 (RSA is asymmetric)[148]. Possibly a T/F like "In symmetric encryption, the same key is used for E and D" – True[167][147]. Or "Public-key crypto is much slower than symmetric" – True[168]. They might have asked to identify what provides integrity (e.g., hashing or MAC) vs confidentiality (encryption). Q18 was IP address, not security. But Q21-25 had some T/F: specifically, none on security except maybe Q19 covers that content. But the written part might have had a question like "What is one advantage of public-key over secret-key encryption?" – likely "no need to share secret prior". Or "What problem do digital signatures solve?" – authenticity, non-repudiation. Or even given a scenario: "Alice wants to ensure Bob knows a message is truly from her – what should she use?" – Answer: digital signature with her private key. Or "What protocol enables two parties to establish a shared secret over insecure channel?" – Diffie-Hellman key exchange.

We saw Q19 covers RSA classification, Q3 covers hash length. The presence of these suggests they expected familiarity with common algorithms and their categories.

In conclusion, Week 11 provided the basic toolkit of cryptography and how it's applied for network security (HTTPS, SSH) and data protection. This knowledge not only helps exam answers but informs secure programming and system design practices.

Week 12: Performance Analysis

- **Why Performance Matters**: In distributed systems and modern applications, performance bottlenecks can degrade user experience and increase costs[169]. Small delays across thousands of microservice calls add up[170]. Also, inefficient

resource usage can mean needing more servers (higher cost). So we analyze performance to optimize reliability and scalability[170].

- **Metrics**: Key performance metrics include[171]:
- **Latency**: time to complete one request (a.k.a. response time). Often measured in milliseconds or seconds. Users notice latency – e.g., web page load time.
- **Throughput**: the rate of work done, e.g., requests per second a server or system can handle[172]. For networks, often in bits/sec. For servers, maybe transactions/sec.
- **Utilization**: the percentage of a resource's capacity in use (CPU utilization %, memory usage, network bandwidth usage)[173]. We watch this to know if a resource is saturated (100% util -> queueing delays).
- **Scalability**: how performance changes as the system grows (e.g., if we double the number of servers, do we nearly double throughput? Or do we hit some other bottleneck?)[174]. Ideally linear scalability.
- **Queueing delay**: if demand exceeds service rate, requests queue up causing wait time[175]. This is often modeled in queueing theory (Little's law etc.).
  For example, if CPU is at 100%, tasks line up (in OS run queue) and experience scheduling delay. If disk is busy, I/O requests queue.
- **Sources of Latency (Network example)**: The lecture gave the breakdown of packet delay[176][177]:
- **Processing delay (d_proc)**: router/host processing time (checking headers, doing computations). Usually very small (<1 ms).[178]
- **Queuing delay (d_queue)**: time a packet waits in router's queue before being transmitted. Depends on congestion and can vary from 0 to ms or more if congested[179][180]. If traffic intensity La/R ~ 1 (meaning link is busy), queue delay becomes large/unbounded[181].
- **Transmission delay (d_trans)**: time to push the packet's bits onto the link = packet length (bits) / link bandwidth (bps)[182]. E.g., 1500 bytes = 12,000 bits on 1 Mbps link is 0.012 s = 12 ms. If using high-speed links, it's tiny.
- **Propagation delay (d_prop)**: time for signal to travel through the medium = distance / propagation speed[183]. Roughly 2*10^8 m/s in fiber (~2/3 speed of light). So 100 km ~ 0.5 ms. Cross-country ~ 20-30 ms, transoceanic ~ 50+ ms.
  Total nodal delay = d_proc + d_queue + d_trans + d_prop[184][185]. The caravan analogy illustrated propagation vs transmission difference[186][187]. The result: first car arrives after propagation, last car arrives after propagation plus time for all cars to get through toll booths (transmission), etc., showing how the formula works out[188][189] (they got 62 minutes in example).
- **Bottlenecks and Throughput**: A system's throughput is constrained by the slowest component in the path (bottleneck). If you have a pipeline of stages (e.g., data goes through service A then B then C), the stage with lowest capacity will determine overall throughput (like smallest pipe in plumbing). Performance analysis identifies these bottlenecks. Tools: profile CPU usage, measure disk or network IO rates, find

which resource is 100% utilized. If CPU is maxed but network is free, CPU is bottleneck. If CPU is 20% but network link is saturated, network is bottleneck.

- **Parallelism and Amdahl's Law**: (Not sure if covered, but likely as a core concept.) Amdahl's Law says if you speed up a portion of a task, the overall improvement is limited by the fraction of time that portion was used originally. If 50% of time is parallelizable and you make it infinitely fast, you still have 50% time that's serial -> at best 2x speedup. Encourages focusing optimization on big contributors.
- **Performance Tools**:
- **Profilers**: in code, to see which functions consume most time (CPU profiling). e.g., gprof, perf.
- **Monitoring**: OS tools like `top` (CPU/mem usage of processes), `iostat` (disk IO stats), `sar`, etc. For network, `ping` and `traceroute` measure latency, `iperf` measures bandwidth throughput between two endpoints.
- **Load testing**: simulate many requests (with tools like JMeter, ab (ApacheBench), etc.) to see how system scales.
- **Logging and tracing**: instrument the system to measure request latencies, log events (like microservice call timings) to find slow hops.
- **Distributed Systems Issues**: In distributed systems, variance in network or node performance can cause unpredictable delays. E.g., if one microservice call is slightly slow, in a fan-out it can hold up the entire user request (tail latency problem – the slowest out of many parallel calls dictates overall latency). So one needs to design with timeouts and perhaps redundancy for critical calls. Also consider CAP theorem (consistency vs availability trade-offs) – sometimes performance means trading off strict consistency. Possibly out of scope here.
- **Reliability and Perf**: Sometimes improving performance can improve reliability (e.g., shorter queues means less chance to drop), but sometimes there's trade-off (like heavy optimization that complicates code might risk bugs).
- **Queuing Theory basics**: Maybe they gave formula Little's Law: `L = λ * W` (average number in system = throughput * response time). Or discussed how if λ (arrival rate) approaches μ (service rate) utilization→100% and wait times grow non-linearly. They did mention intensity La/R[181], basically ρ = utilization. When ρ -> 1, delays → ∞. That's why operating at high utilization is dangerous for latency. Real systems often target < 70% utilization to keep latencies low and accommodate bursts.
- **Case Study**: Possibly analyzing a simple scenario like: you have a 1 Gbps link, sending 100 MB file – how long should it take? 100 MB = 800 Mb, at 1 Gbps it's 0.8 seconds plus any latency overhead. Or if using a 100 Mbps link, would be ~8 sec. Considering maybe TCP slow start etc.
- **Performance vs Efficiency**: Performance is about speed, throughput, latency. Efficiency might be about resource usage (like operations per CPU second). They often correlate, but a high-performance solution might use more resources.
- **Measurement**: Emphasize need to measure, not guess, where time is spent. Use of logs, timestamps, system metrics. E.g., measure how much time in DB vs in app.

- **Scaling**: vertical (bigger machine) vs horizontal (more machines). Cloud elasticity (which we talked about in Week 10).

Key Exam Connections: If any, possibly theoretical. Not in the MCQ snippet. Maybe a written question: e.g., "Name four sources of packet delay" – expected: processing, queueing, transmission, propagation (from lecture)[176][190]. Or "What happens if traffic intensity La/R is > 1?" – queue will grow without bound (system unstable)[181]. Or "Define throughput vs latency." Or maybe given a scenario: "We have X requests per second, average response 200ms, what's throughput? what's capacity needed to handle 2X load?" etc.

The exam short answer Q26-37 were specific from earlier content. Perhaps performance was considered more conceptual and not directly tested, or might have been part of an optional question. Regardless, understanding performance is critical in practice, tying together knowledge from networks (for propagation vs transmission) and OS (for scheduling delays, etc.).

This concludes Week 12 and the course content.

[1] [19] [26] [28] [29] [30] [31] [38] [40] [43] [46] [53] [54] [55] [56] [59] [60] [75] [76] [77] [78] [79] [80] [81] [82] [100] [103] [105] [108] [109] [111] [148] [160] [166] INFO1112_Computing_1B_OS_and_Network_Platforms_final.pdf

file://file_00000000f4b071fa9a83de26107c6f30

[2] [3] [4] [5] [6] [7] [8] [9] [10] Lecture01B_Bits_2025.pdf

file://file_000000006ce07206b145d10076db9a62

[11] [12] [13] [14] [15] [16] [17] [18] [20] [21] [22] [23] [24] [25] [27] [47] Lecture02_Shell&Processes_2025-08-11-WEILI.pdf

file://file_00000000f68872069fbac727f2295bb0

[32] [33] [34] [35] [36] [37] [39] [41] [42] [51] [52] [57] [58] Lecture04_Processes&Filesystem_2025-08-25.pdf

file://file_0000000072b07206ad4b70e85459c309

[44] [45] c - Differences between fork and exec - Stack Overflow

https://stackoverflow.com/questions/1653340/differences-between-fork-and-exec

[48] [49] [50] Lecture03_Emulators_VirtualMachines_Containers_2025-08-18.pdf

file://file_0000000095f472068b1221fce3ba16c1

[61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] Lecture05_Memory-Scheduler-Bootstrap-Shell_2025-09-01.pdf

file://file_00000000b06c720689ceded9a264fbfd

[83] [84] [85] [86] [87] [88] [89] [90] [91] [92] [93] [94] [95] [96] [99] [106] [107] [110] Lecture06_Intro-to-Networks-Switching_2025-09-08 (1).pdf

file://file_000000009b3c720696ce3fc81000e134

[97] [98] [101] [102] Lecture07_IP-and-Routing_2025-09-15.pdf

file://file_00000000f7c47206b5d17dc9cfda2772

[104] [112] [113] [114] [115] [116] [117] [118] [119] [120] [121] [122] [123] Lecture08_DNS-App-Protocols_2025-09-22.pdf

file://file_00000000c788720685d999a38ac83d3f

[124] [125] [126] [127] [128] [129] [130] [131] [132] [133] [134] [135] [136] [137] [138] [139] [140] Lecture10_Cloud-Computing_2025-10-13.pdf

file://file_00000000a10c7206a7b7f4a6f358874a

[141] [142] [143] [144] [145] [146] [147] [149] [150] [151] [152] [153] [154] [157] [158] [159] [161] [162] [163] [164] [165] [167] [168] Lecture11_Security-Crypto-Signature_2025-10-20.pdf

file://file_00000000750c7206bd3d28eb33b46c82

[155] [156] Diffie–Hellman key exchange - Wikipedia

https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

[169] [170] [171] [172] [173] [174] [175] [176] [177] [178] [179] [180] [181] [182] [183] [184] [185] [186] [187] [188] [189] [190] Lecture12_Performance-Analysis_2025-10-27.pdf

file://file_00000000defc7206bad18d5485e77ad1