1. Compilation:

1.1 Compiled and Interpreted Languages

- "Compiled" means that programs are translated into machine language and then executed by hardware;
    - Ex: C Language
- "interpreted" means that programs are read and executed by a software interpreter.
    - Ex: Python
- Java uses an intermediate language called Java bytecode, which is similar to machine language, but it is executed by a software interpreter, the Java virtual machine (JVM).

1.2 Static Types:

- Many interpreted languages support dynamic types, but compiled languages are usually limited to static types.
- In a statically-typed language, you can tell by looking at the program what type each variable refers to. In a dynamically typed language, you don't always know the type of a variable until the program is running
    - In general, static refers to things that happen at compile time (while a program is being compiled), and dynamic refers to things that happen at run time (while a program is running).
    - Ex: in python, we can just do x + y.. But in C we need to specify data types: int x + int y;
- Due to declaring data types, when this function is called elsewhere in the program, the compiler can check whether the arguments provided have the right type, and whether the return value is used correctly.
    - these checks don't have to happen at run time, which is one of the reasons compiled languages generally run faster than interpreted languages.
- Declaring types at compile time also saves space. In dynamic languages, variable names are stored in memory while the program runs, and they are often accessible by the program
    - In compiled languages, variable names exist at compile-time but not at run time.
    - The compiler chooses a location for each variable and records these locations as part of the compiled program.1 The location of a variable is called its address.
    - At run time, the value of each variable is stored at its address, but the names of the variables are not stored at all

1.3 Compilation Process:

- Steps of Compilation:
    - Preprocessing: preprocessing directives that take effect before the program is compiled. For example, the #include directive causes the source code from another file to be inserted at the location of the directive
    - Parsing: During parsing, the compiler reads the source code and builds an internal representation of the program, called an abstract syntax tree. Errors detected during this step are generally syntax errors.

    - Static checking: The compiler checks whether variables and values have the right type, whether functions are called with the right number and type of arguments, etc. Errors detected during this step are sometimes called static semantic errors.
    - Code generation: The compiler reads the internal representation of the program and generates machine code or byte code.
    - Linking: If the program uses values and functions defined in a library, the compiler has to find the appropriate library and include the required code.
    - Optimization: At several points in the process, the compiler can transform the program to generate code that runs faster or uses less space.

2. Processes:

2.1 Abstraction and Virtualization

- An abstraction is a simplified representation of something complicated. Ex: when clicking on a link in the web, the software and network communication is abstracted away
- An important kind of abstraction is virtualization, which is the process of creating a desirable illusion.
    - a virtual machine is software that creates the illusion of a dedicated computer running a particular operating system, when in reality the virtual machine might be running, along with many other virtual machines, on a computer running a different operating system.
    - In the context of virtualization, we sometimes call what is really happening “physical”, and what is virtually happening either “logical” or “abstract.”

2.2 Isolation

- when you are designing a system with multiple components, it is usually a good idea to isolate them from each other so that a change in one component doesn’t have undesired effects on other components.
- One of the most important goals of an operating system is to isolate each running program from the others so that programmers don’t have to think about every possible interaction. The software object that provides this isolation is a process.
- A process is an object that contains the following data:
    - The text of the program, usually a sequence of machine language instructions.
    - Data associated with the program, including static data (allocated at compile time) and dynamic data (allocated at run time).
    - The state of any pending input/output operations. For example, if the process is waiting for data to be read from disk or for a packet to arrive on a network, the status of these operations is part of the process.
    - The hardware state of the program, which includes data stored in registers, status information, and the program counter, which indicates which instruction is currently executing.
- Multitasking: Most operating systems have the ability to interrupt a running process at almost any time, save its hardware state, and then resume the process later. The program behaves as if it is running continuously on a dedicated processor, except that the time between instructions is unpredictable.

- Virtual memory: Most operating systems create the illusion that each process has its own chunk of memory, isolated from all other processes. Programmers can proceed as if every program has a dedicated chunk of memory.
- Device abstraction: Processes running on the same computer share the disk drive, the network interface, the graphics card, and other hardware. If processes interacted with this hardware directly, without coordination, chaos would ensue. For example, network data intended for one process might be read by another. Or multiple processes might try to store data in the same location on a hard drive. It is up to the operating system to maintain order by providing appropriate abstractions.

2.3 UNIX Processes

- One example of a process is kthreadd:
    - kthreadd is a process the operating system uses to create new threads.
        - The k at the beginning stands for kernel, which is the part of the operating system responsible for core capabilities like creating threads
        - The extra d at the end stands for daemon, which is another name for processes that run in the background and provide operating system services.

3. Virtual Memory:

3.1 Information Theory:

- A bit is a binary digit; it is also a unit of information
- Written in 0 or 1
    - If you have two bits, there are 4 possible combinations, 00, 01, 10, and 11. In general, if you have b bits, you can indicate one of 2b values.
- A byte is 8 bits, so it can hold one of 256 values.
- In general, if you want to specify one of N values, you should choose the smallest value of b so that $2^b \geq N$. Taking the log base 2 of both sides yields $b \geq \log_2 N$.
- Equivalently, if the probability of the outcome is p, then the information content is $-\log_2 p$. This quantity is called the self-information of the outcome. It measures how surprising the outcome is, which is why it is also called surprisal.

3.2 Memory and storage

- While a process is running, most of its data is held in main memory, which is usually some kind of random access memory (RAM).
    - On most current computers, main memory is volatile, which means that when the computer shuts down, the contents of main memory are lost.
    - A typical desktop computer has 2–8 GiB of memory. GiB stands for “gibibyte,” which is 2^30 bytes.
- If the process reads and writes files, those files are usually stored on a hard disk drive (HDD) or solid state drive (SSD). These storage devices are nonvolatile, so they are used for long-term storage.
    - Currently a typical desktop computer has a HDD with a capacity of 500 GB to 2 TB. GB stands for “gigabyte,” which is 10^9 bytes. TB stands for “terabyte,” which is 10^12 bytes.

- memory is measured in binary units, and disk drives are measured in decimal units.

3.3 Address Spaces

- Each byte in main memory is specified by an integer physical address. The set of valid physical addresses is called the physical address space.
- It usually runs from 0 to N − 1, where N is the size of main memory.
- On a system with 1 GiB of physical memory, the highest valid address is 2^30 − 1, which is 1,073,741,823 in decimal, or 0x3fff ffff in hexadecimal.
- However, most operating systems provide virtual memory, which means that programs never deal with physical addresses, and don't have to know how much physical memory is available.
    - Instead, programs work with virtual addresses, which are numbered from 0 to M − 1, where M is the number of valid virtual addresses. The size of the virtual address space is determined by the operating system and the hardware it runs on.
    - 32-bit and 64-bit systems are terms that indicate the size of the registers, which is usually also the size of a virtual address. On a 32-bit system, virtual addresses are 32 bits, which means that the virtual address space runs from 0 to 0xffff ffff. The size of this address space is 232 bytes, or 4 GiB
    - On a 64-bit system, the size of the virtual address space is 2^64 bytes
    - When a program reads and writes values in memory, it generates virtual addresses. The hardware, with help from the operating system, translates to physical addresses before accessing main memory. This translation is done on a per-process basis, so even if two processes generate the same virtual address, they would map to different locations in physical memory.
    - In general, a process cannot access data belonging to another process, because there is no virtual address it can generate that maps to physical memory allocated to another process.

3.4 Memory Segments

- The data of a running process is organized into five segments:
    - The code segment contains the program text; that is, the machine language instructions that make up the program.
    - The static segment contains immutable values, like string literals. For example, if your program contains the string "Hello, World", those characters will be stored in the static segment.
    - The global segment contains global variables and local variables that are declared static.
    - The heap segment contains chunks of memory allocated at run time, most often by calling the C library function malloc.
    - The stack segment contains the call stack, which is a sequence of stack frames. Each time a function is called, a stack frame is allocated to contain the parameters and local variables of the function. When the function completes, its stack frame is removed from the stack.

- The arrangement of these segments is determined partly by the compiler and partly by the operating system.
    - The text segment is near the “bottom” of memory, that is, at addresses near 0.
    - The static segment is often just above the text segment, that is, at higher addresses.
    - The global segment is often just above the static segment.
    - The heap is often above the global segment. As it expands, it grows up toward larger addresses.
    - The stack is near the top of memory; that is, near the highest addresses in the virtual address space. As the stack expands, it grows down toward smaller addresses.
- Local variables on the stack are sometimes called automatic, because they are allocated automatically when a function is called, and freed automatically when the function returns.

3.6 Address Translation

- How does a virtual address (VA) get translated to a physical address (PA)?
    - Most processors provide a memory management unit (MMU) that sits between the CPU and main memory. The MMU performs fast translation between VAs and PAs.
        - When a program reads or writes a variable, the CPU generates a VA.
        - The MMU splits the VA into two parts, called the page number and the offset. A “page” is a chunk of memory; the size of a page depends on the operating system and the hardware, but common sizes are 1–4 KiB.
        - The MMU looks up the page number in the translation lookaside buffer (TLB) and gets the corresponding physical page number. Then it combines the physical page number with the offset to produce a PA.
        - The PA is passed to main memory, which reads or writes the given location.
    - The TLB (translation lookaside buffer) contains cached copies of data from the page table (which is stored in kernel memory). The page table contains the mapping from virtual page numbers to physical page numbers. Since each process has its own page table, the TLB has to make sure it only uses entries from the page table of the process that’s running.
        - In other words, it is a type of memory cache that stores recent translations of virtual memory to physical addresses to enable faster retrieval.
    - most processes don’t use even a small fraction of their virtual address space.
- The page table is a data structure used in virtual memory to store the mapping between virtual and physical addresses.
    - Disadvantages of page tables is the size. The page table can get very large.
    - Accessing a page table for every memory access can slow down system performance
    - The page table must be kept in memory; can consume a lot of system memory

- Multilevel page tables breaks down the page table into smaller more manageable pieces. Instead of one large table, there are several levels of smaller tables reducing the memory required to store non used entries.
    - Reduced memory usage, scalable for larger address spaces and they are flexible.
    - Linux uses multilevel page tables.
- the operating system can interrupt a running process, save its state, and then run another process. This mechanism is called a context switch. Since each process has its own page table, the operating system has to work with the MMU to make sure each process gets the right page table.
    - In newer systems, each page table entry in the MMU includes the process ID, so page tables from multiple processes can be in the MMU at the same time.

4. Files and File Systems

- When a process completes (or crashes), any data stored in main memory is lost. But data stored on a hard disk drive (HDD) or solid state drive (SSD) is “persistent;” that is, it survives after the process completes, even if the computer shuts down.
- In hard drives, data is stored in blocks, which are laid out in sectors, which make up tracks, which are arranged in concentric circles on platters.
- Solid state drives are simpler in one sense, because blocks are numbered sequentially, but they raise a different complication: each block can be written a limited number of times before it becomes unreliable.
- The abstraction of persistent storage hardware: The file system.
    - A “file system” is a mapping from each file’s name to its contents. If you think of the names as keys, and the contents as values, a file system is a kind of key-value database
    - A “file” is a sequence of bytes.
    - File names are usually strings, and they are usually “hierarchical”; that is, the string specifies a path from a top-level directory (or folder), through a series of subdirectories, to a specific file
    - The primary difference between the abstraction and the underlying mechanism is that files are byte-based and persistent storage is block-based. The operating system translates byte-based file operations in the C library into block-based operations on storage devices. Typical block sizes are 1–8 KiB.

4.1 Disk Performance

- On current HDDs, the average time to read a block from disk to memory might be 5–25 ms
- SSDs are faster, taking 25 µs (microseconds) to read a 4 KiB block and 250 µs to write one
- The gap in performance between main memory and persistent storage is one of the major challenges of computer system design. Operating systems and hardware provide several features intended to “fill in” this gap:
    - Block transfers: The time it takes to load a single byte from disk is 5– 25 ms. By comparison, the additional time to load an 8 KiB block is negligible. So systems generally try to read large blocks each time they access the disk.

    - Prefetching: Sometimes the operating system can predict that a process will read a block and start loading it before it is requested. For example, if you open a file and read the first block, there is a good chance you will go on to read the second block. The operating system might start loading additional blocks before they are requested.
    - Buffering: when you write a file, the operating system stores the data in memory and only writes it to disk later. If you modify the block several times while it is in memory, the system only has to write it to disk once.
    - Caching: If a process has used a block recently, it is likely to use it again soon. If the operating system keeps a copy of the block in memory, it can handle future requests at memory speed.
- some disk drives provide a cache that stores recently-used blocks, and many disk drives read more than one block at a time, even if only one is requested.

4.2 Disk Metadata

- The blocks that make up a file might be arranged contiguously on disk, and file system performance is generally better if they are, but most operating systems don't require contiguous allocation. They are free to place a block anywhere on disk, and they use various data structures to keep track of them.
    - In many UNIX file systems, that data structure is called an "inode," which stands for "index node".
    - information about files, including the location of their blocks, is called "metadata".
    - Since inodes reside on disk along with the rest of the data, they are designed to fit neatly into disk blocks. A UNIX inode contains information about a file, including the user ID of the file owner; permission flags indicating who is allowed to read, write, or execute it; and timestamps that indicate when it was last modified and accessed. In addition, it contains block numbers for the first 12 blocks that make up the file.
        - If the block size is 8 KiB, the first 12 blocks make up 96 KiB. On most systems, that's big enough for a large majority of files, but it's definitely not big enough for all of them. That's why the inode also contains a pointer to an "indirection block", which contains nothing but pointers to other blocks. With 1024 block numbers and 8 KiB blocks, an indirection block can address 8 MiB.
        - the inode also contains a pointer to a "double indirection block", which contains pointers to indirection blocks. With 1024 indirection blocks, we can address 8 GiB.
        - Also contains a triple indirection block, which contains pointers to double indirection blocks, yielding a maximum file size of 8 TiB.
    - As an alternative to indirection blocks, some files systems, like FAT, use a File Allocation Table that contains one entry for each block, called a "cluster" in this context.

4.3 Block Allocation

- File systems have to keep track of which blocks belong to each file; they also have to keep track of which blocks are available for use. When a new file is created, the file

system finds an available block and allocates it. When a file is deleted, the file system makes its blocks available for re-allocation

- The goals of the block allocation system are:
    - Speed: Allocating and freeing blocks should be fast.
    - Minimal space overhead: The data structures used by the allocator should be small, leaving as much space as possible for data.
    - Minimal fragmentation: If some blocks are left unused, or some are only partially used, the unused space is called “fragmentation”.
    - Maximum contiguity: Data that is likely to be used at the same time should be physically contiguous, if possible, to improve performance.
- However, a file system that is well tuned for one goal/workload might not perform as well for another.
- Due to this, Linux systems have migrated from ext2, which was a conventional UNIX file system, to ext3, a “journaling” file system intended to improve speed and contiguity, and more recently to ext4, which can handle larger files and file systems

4.4 Everything is a file

- The file abstraction is really a “stream of bytes” abstraction, which turns out to be useful for many things, not just file systems.
- One example is the UNIX pipe, which is a simple form of inter-process communication. Processes can be set up so that output from one process is taken as input into another process.
- Network communication also uses the stream of bytes abstraction. A UNIX socket is a data structure that represents a communication channel between processes on different computers (usually). Again, processes can read data from and write data to a socket using “file” handling functions.

5. Bits and Bytes

- Computers represent numbers in base 2, also known as binary.
- For negative numbers, the most obvious representation uses a sign bit to indicate whether a number is positive or negative. But there is another representation, called “two’s complement” that is much more common because it is easier to work with in hardware.
- To find the two’s complement of a negative number, −x, find the binary representation of x, flip all the bits, and add 1.
    - For example, to represent −5 ∧10, start with the representation of 5∧10, which is b00000101 if we write the 8-bit version. Flipping all the bits and adding 1 yields b11111011.
    - In two’s complement, the leftmost bit acts like a sign bit; it is 0 for positive numbers and 1 for negative numbers.
    - To convert from an 8-bit number to 16-bits, we have to add more 0’s for a positive number and add 1’s for a negative number. In effect, we have to copy the sign bit into the new bits. This process is called “sign extension”.
    - In C all integer types are signed (able to represent positive and negative numbers) unless you declare them unsigned. The difference, and the reason this

declaration is important, is that operations on unsigned integers don't use sign extension

5.2 Bitwise Operators

- The & and | operators treat integers as bit vectors and compute logical operations on corresponding bits.
    - For example, & computes the AND operation, which yields 1 if both operands are 1, and 0 otherwise. Ex: 1100 & 1010 = 1000
    - Similarly, | computes the OR operation, which yields 1 if either operand is 1, and 0 otherwise. Ex: 1100 | 1010 = 1110
    - Finally, ^ computes the XOR operation, which yields 1 if either operand is 1, but not both. Ex: 1100 ^ 1010 = 0110

5.3 Floating Point Nums

- Floating-point numbers are represented using the binary version of scientific notation. In decimal notation, large numbers are written as the product of a coefficient and 10 raised to an exponent. For example, the speed of light in m/s is approximately 2.998 · 10^8.
- Most computers use the IEEE standard for floating-point arithmetic. The C type float usually corresponds to the 32-bit IEEE standard; double usually corresponds to the 64-bit standard.
- The value of a floating-point number is (−1)^s c · 2^q
- Floating-point numbers are usually normalized so that there is one digit before the point. For example, in base 10, we prefer 2.998 · 10^8 rather than 2998 · 10^5
- The ASCII codes for the digits "0" through "9" are 48 through 57, not 0 through 9.

6.Memory Management

- Dynamic Memory Allocation: The mechanism by which storage/memory/cells can be allocated to variables during the run time. It allocates the memory during the run time which enables us to use as much storage as we want, without worrying about any wastage.
- C provides 4 functions for dynamic memory allocation:
    - malloc, which takes an integer size, in bytes, and returns a pointer to a newly-allocated chunk of memory with (at least) the given size. If it can't satisfy the request, it returns the special pointer value NULL.
    - calloc, which is the same as malloc except that it also clears the newly allocated chunk; that is, it sets all bytes in the chunk to 0.
    - Free, which takes a pointer to a previously allocated chunk and deallocates it; that is, it makes the space available for future allocation.
    - Realloc, which takes a pointer to a previously allocated chunk and a new size. It allocates a chunk of memory with the new size, copies data from the old chunk to the new, frees the old chunk, and returns a pointer to the new chunk.

6.1 Memory Errors

- Paddling - punishment
    - If you access (read or write) any chunk that has not been allocated, that's a paddling.
    - If you free an allocated chunk and then access it, that's a paddling.
    - If you try to free a chunk that has not been allocated, that's a paddling.

    - If you free the same chunk more than once, that's a paddling.
    - If you call realloc with a chunk that was not allocated, or was allocated and then freed, that's a paddling.
- in a large program a chunk of memory might be allocated in one part of the program, used in several other parts, and freed in yet another part. So changes in one part of the program can require changes in many other parts.
- Also, there might be many aliases, or references to the same allocated chunk, in different parts of the program. The chunk should not be freed until all references to the chunk are no longer in use.
- If you read a value from an unallocated chunk, the system might detect the error, trigger a runtime error called a "segmentation fault", and stop the program.
- If you write a value to an unallocated chunk, and don't get a segmentation fault, things are even worse. After you write a value to an invalid location, a long time might pass before it is read and causes problems.
- One of the most common problems with C-style memory management is that the data structures used to implement malloc and free are often stored along with the allocated chunks. So if you accidentally write past the end of a dynamically allocated chunk, you are likely to mangle these data structures. The system usually won't detect the problem until later, when you call malloc or free, and those functions fail in some inscrutable way.
- The most common source of memory errors is writing beyond the bounds of an array. Should check bounds; , every access to the array should check whether the index is out of bounds.

6.2 Memory Leaks

- If you allocate a chunk of memory and never free it, that's a "memory leak".
- if your program allocates memory, performs computations on it, and then exits, it is probably not necessary to free the allocated memory. When the program exits, all of its memory is deallocated by the operating system.
- But if a program runs for a long time and leaks memory, its total memory use will increase indefinitely. At that point, a few things might happen:
    - At some point, the system runs out of physical memory. On systems without virtual memory, the next call to malloc will fail, returning NULL.
    - On systems with virtual memory, the operating system can move another process's pages from memory to disk and then allocate more space to the leaking process.
    - There might be a limit on the amount of space a single process can allocate; beyond that, malloc returns NULL.
    - Eventually, a process might fill its virtual address space (or the usable part). After that, there are no more addresses to allocate, so malloc returns NULL.
- If malloc returns NULL, but you persist and access the chunk you think you allocated, you get a segmentation fault.

6.3 Implementation

- When a process starts, the system allocates space for the text segment and statically allocated data, space for the stack, and space for the heap, which contains dynamically allocated data.

- Not all programs allocate data dynamically, so the initial size of the heap might be small or zero. Initially the heap contains only one free chunk.
    - When malloc is called, it checks whether it can find a free chunk that's big enough. If not, it has to request more memory from the system. The function that does that is sbrk, which sets the "program break", which you can think of as a pointer to the end of the heap.
    - When sbrk is called, the OS allocates new pages of physical memory, updates the process's page table, and sets the program break.
- The run time of malloc does not usually depend on the size of the chunk, but might depend on how many free chunks there are. free is usually fast, regardless of the number of free chunks. Because calloc clears every byte in the chunk, the run time depends on chunk size (as well as the number of free chunks).
    - realloc is sometimes fast, if the new size is smaller than the current size, or if space is available to expand the existing chunk. If not, it has to copy data from the old chunk to the new; in that case, the run time depends on the size of the old chunk.
- Boundary tags: When malloc allocates a chunk, it adds space at the beginning and end to store information about the chunk, including its size and the state (allocated or free). These bits of data are called "boundary tags". Using these tags, malloc can get from any chunk to the previous chunk and the next chunk in memory. In addition, free chunks are chained into a doubly-linked list; each free chunk contains pointers to the next and previous chunks in the "free list".
- Space overhead: Boundary tags and free list pointers take up space. The minimum chunk size on most systems is 16 bytes. So for very small chunks, malloc is not space efficient. If your program requires large numbers of small structures, it might be more efficient to allocate them in arrays.
- Fragmentation: If you allocate and free chunks with varied sizes, the heap will tend to become fragmented. That is, the free space might be broken into many small pieces. Fragmentation wastes space; it also slows the program down by making memory caches less effective.
- Binning and caching: The free list is sorted by size into bins, so when malloc searches for a chunk with a particular size, it knows what bin to search in. If you free a chunk and then immediately allocate a chunk with the same size, malloc will usually be fast.

7. Caching

- When a program starts, the code (or text) is usually on a hard disk or solid state drive. The operating system creates a new process to run the program, then the "loader" copies the text from storage into main memory and starts the program by calling main.
- While the program is running, most of its data is stored in main memory, but some of the data is in registers, which are small units of memory on the CPU. These registers include:
    - The program counter, or PC, which contains the address (in memory) of the next instruction in the program.
    - The instruction register, or IR, which contains the machine code instruction currently executing.

    - The stack pointer, or SP, which contains the address of the stack frame for the current function, which contains its parameters and local variables.
    - General-purpose registers that hold the data the program is currently working with.
    - A status register, or flag register, that contains information about the current computation. For example, the flag register usually contains a bit that is set if the result of the previous operation was zero.
- When a program is running, the CPU executes the following steps, called the “instruction cycle”:
    - Fetch: The next instruction is fetched from memory and stored in the instruction register.
    - Decode: Part of the CPU, called the “control unit”, decodes the instruction and sends signals to the other parts of the CPU.
    - Execute: Signals from the control unit cause the appropriate computation to occur.
- Most computers can execute a few hundred different instructions, called the “instruction set”. But most instructions fall into a few general categories:
    - Load: Transfers a value from memory to a register.
    - Arithmetic/logic: Loads operands from registers, performs a mathematical operation, and stores the result in a register.
    - Store: Transfers a value from a register to memory.
    - Jump/branch: Changes the program counter, causing the flow of execution to jump to another location in the program. Branches are usually conditional, which means that they check a flag in the flag register and jump only if it is set
- During each instruction cycle, one instruction is read from the program text. In addition, about half of the instructions in a typical program load or store data. And therein lies one of the fundamental problems of computer architecture: the “memory bottleneck”.
- In current computers, a typical core is capable of executing an instruction in less than 1 ns. But the time it takes to transfer data to and from memory is about 100 ns. If the CPU has to wait 100 ns to fetch the next instruction, and another 100 ns to load data, it would complete instructions 200 times slower than what’s theoretically possible. For many computations, memory is the speed limiting factor, not the CPU

7.2 Cache Performance

- The solution to this problem, or at least a partial solution, is caching. A “cache” is a small, fast memory that is physically close to the CPU, usually on the same chip.
- Current computers typically have several levels of cache: the Level 1 cache, which is the smallest and fastest, might be 1–2 MiB with a access times near 1 ns; the Level 2 cache might have access times near 4 ns, and the Level 3 might take 16 ns.
    - When the CPU loads a value from memory, it stores a copy in the cache. If the same value is loaded again, the CPU gets the cached copy and doesn’t have to wait for memory.
    - Eventually the cache gets full. Then, in order to bring something new in, we have to kick something out. So if the CPU loads a value and then loads it again much later, it might not be in cache any more

- If the instructions and data needed by the CPU are usually in cache, the program can run close to the full speed of the CPU. If the CPU frequently needs data that are not in cache, the program is limited by the speed of memory
- The cache “hit rate”, h, is the fraction of memory accesses that find data in cache; the “miss rate”, m, is the fraction of memory accesses that have to go to memory.

7.3 Locality

- When a program reads a byte for the first time, the cache usually loads a “block” or “line” of data that includes the requested byte and some of its neighbors. If the program goes on to read one of the neighbors, it will already be in cache
- The tendency of a program to use the same data more than once is called “temporal locality”. The tendency to use data in nearby locations is called “spatial locality”.
  - Most programs contain blocks of code with no jumps or branches. Within these blocks, instructions run sequentially, so the access pattern has spatial locality.
  - In a loop, programs execute the same instructions many times, so the access pattern has temporal locality
  - The result of one instruction is often used immediately as an operand of the next instruction, so the data access pattern has temporal locality.
  - When a program executes a function, its parameters and local variables are stored together on the stack; accessing these values has spatial locality.
  - One of the most common processing patterns is to read or write the elements of an array sequentially; this pattern also has spatial locality.

7.5 Programming for Cache Performance

- if you are working with a large array, it might be faster to traverse the array once, performing several operations with each element, rather than traversing the array several times
- If you are working with a 2-D array, it might be stored as an array of rows. If you traverse through the elements, it would be faster to go row-wise, with stride equal to the element size, rather than column-wise, with stride equal to the row length.
- Linked data structures don’t always exhibit spatial locality, because the nodes aren’t necessarily contiguous in memory. But if you allocate many nodes at the same time, they are usually co-located in the heap. Or, even better, if you allocate an array of nodes all at once, you know they will be contiguous.
- Recursive strategies like mergesort often have good cache behavior because they break big arrays into smaller pieces and then work with the pieces.

7.6 The Memory Hierarchy

The following table shows typical access times, sizes, and costs for each of these technologies.

Device	Access time	Typical size	Cost
Register	0.5 ns	256 B	?
Cache	1 ns	2 MiB	?
DRAM	100 ns	4 GiB	$10 / GiB
SSD	10 μs	100 GiB	$1 / GiB
HDD	5 ms	500 GiB	$0.25 / GiB
Tape	minutes	1–2 TiB	$0.02 / GiB

- Caches are fast because they are small and close to the CPU, which minimizes delays due to capacitance and signal propagation. If you make a cache big, it will be slower. Hence we cannot replace memory with a bigger cache.
- Also, caches take up space on the processor chip, and bigger chips are more

expensive. Main memory is usually dynamic random-access memory (DRAM), which uses only one transistor and one capacitor per bit, so it is possible to pack more memory into the same amount of space.

- main memory is usually packaged in a dual in-line memory module (DIMM) that includes 16 or more chips. Several small chips are cheaper than one big one.
- Solid state drives (SSD) are fast, but they are more expensive than hard drives (HDD), so they tend to be smaller
- The number and size of registers depends on details of the architecture. Current computers have about 32 general-purpose registers, each storing one “word”. On a 32-bit computer, a word is 32 bits or 4 B. On a 64-bit computer, a word is 64 bits or 8 B. So the total size of the register file is 100–300 B.

7.7 Caching Policy

- At every level of the memory hierarchy, we have to address four fundamental questions of caching:
    - Who moves data up and down the hierarchy? At the top of the hierarchy, register allocation is usually done by the compiler. Hardware on the CPU handles the memory cache. Users implicitly move data from storage to memory when they execute programs and open files. But the operating system also moves data back and forth between memory and storage. At the bottom of the hierarchy, administrators move data explicitly between disk and tape
    - What gets moved? In general, block sizes are small at the top of the hierarchy and bigger at the bottom. In a memory cache, a typical block size is 128 B. Pages in memory might be 4 KiB, but when the operating system reads a file from disk, it might read 10s or 100s of blocks at a time.
    - When does data get moved? In the most basic cache, data gets moved into cache when it is used for the first time. But many caches use some kind of “prefetching”, meaning that data is loaded before it is explicitly requested. We have already seen one form of prefetching: loading an entire block when only part of it is requested
    - Where in the cache does the data go? When the cache is full, we can’t bring anything in without kicking something out. Ideally, we want to keep data that will be used again soon and replace data that won’t.
- The answers to these questions make up the “cache policy”. Near the top of the hierarchy, cache policies tend to be simple because they have to be fast and they are implemented in hardware. Near the bottom of the hierarchy, there is more time to make decisions, and well-designed policies can make a big difference.
- if a block of data has been used recently, we expect it to be used again soon. This principle suggests a replacement policy called “least recently used,” or LRU, which removes from the cache a block of data that has not been used recently

7.8 Paging

- In systems with virtual memory, the operating system can move pages back and forth between memory and storage. This mechanism is called paging or swapping.

    - Suppose Process A calls malloc to allocate a chunk. If there is no free space in the heap with the requested size, malloc calls sbrk to ask the operating system for more memory.
    - If there is a free page in physical memory, the operating system adds it to the page table for Process A, creating a new range of valid virtual addresses.
    - If there are no free pages, the paging system chooses a “victim page” belonging to Process B. It copies the contents of the victim page from memory to disk, then it modifies the page table for Process B to indicate that this page is “swapped out”.
    - Once the data from Process B is written, the page can be reallocated to Process A. To prevent Process A from reading Process B’s data, the page should be cleared.
    - At this point the call to sbrk can return, giving malloc additional space in the heap. Then malloc allocates the requested chunk and returns. Process A can resume.
    - When Process A completes, or is interrupted, the scheduler might allow Process B to resume. When Process B accesses a page that has been swapped out, the memory management unit notices that the page is “invalid” and causes an interrupt.
    - When the operating system handles the interrupt, it sees that the page is swapped out, so it transfers the page back from disk to memory.
    - Once the page is swapped in, Process B can resume.
- When paging works well, it can greatly improve the utilization of physical memory, allowing more processes to run in less space. Here’s why:
    - Most processes don’t use all of their allocated memory. Many parts of the text segment are never executed, or execute once and never again. Those pages can be swapped out without causing any problems.
    - If a program leaks memory, it might leave allocated space behind and never access it again. By swapping those pages out, the operating system can effectively plug the leak.
    - On most systems, there are processes like daemons that sit idle most of the time and only occasionally “wake up” to respond to events. While they are idle, these processes can be swapped out.
    - A user might have many windows open, but only a few are active at a time. The inactive processes can be swapped out.
    - Also, there might be many processes running the same program. These processes can share the same text and static segments, avoiding the need to keep multiple copies in physical memory.
- If you add up the total memory allocated to all processes, it can greatly exceed the size of physical memory, and yet the system can still behave well.
    - However, When a process accesses a page that’s swapped out, it has to get the data back from disk, which can take several milliseconds. The delay is often noticeable. If you leave a window idle for a long time and then switch back to it, it

might start slowly, and you might hear the disk drive working while pages are swapped in.
    - Thrashing is when the page fault and swapping happens very frequently at a higher rate, and then the operating system has to spend more time swapping these pages.

8. Multitasking

- In many current systems, the CPU contains multiple cores, which means it can run several processes at the same time. In addition, each core is capable of “multitasking”, which means it can switch from one process to another quickly, creating the illusion that many processes are running at the same time. (virtualizing the CPU)
- The part of the operating system that implements multitasking is the “kernel”. In a nut or seed, the kernel is the innermost part, surrounded by a shell. In an operating system, the kernel is the lowest level of software, surrounded by several other layers, including an interface called a “shell.” Computer scientists love extended metaphors.
- At its most basic, the kernel’s job is to handle interrupts. An “interrupt” is an event that stops the normal instruction cycle and causes the flow of execution to jump to a special section of code called an “interrupt handler”.
- A hardware interrupt is caused when a device sends a signal to the CPU. For example, a network interface might cause an interrupt when a packet of data arrives, or a disk drive might cause an interrupt when a data transfer is complete.
- A software interrupt is caused by a running program. For example, if an instruction cannot complete for some reason, it might trigger an interrupt so the condition can be handled by the operating system. Some floating-point errors, like division by zero, are handled using interrupts.
- When a program needs to access a hardware device, it makes a system call, which is similar to a function call, except that instead of jumping to the beginning of the function, it executes a special instruction that triggers an interrupt, causing the flow of execution to jump to the kernel. The kernel reads the parameters of the system call, performs the requested operation, and then resumes the interrupted process

8.1 Hardware State

- Handling interrupts requires cooperation between hardware and software. When an interrupt occurs, there might be several instructions running on the CPU, data stored in registers, and other hardware state.
- Usually the hardware is responsible for bringing the CPU to a consistent state; for example, every instruction should either complete or behave as if it never started. No instruction should be left half complete. Also, the hardware is responsible for saving the program counter (PC), so the kernel knows where to resume
- Then, usually, it is the responsibility of the interrupt handler to save the rest of the hardware state before it does anything that might modify it, and then restore the saved state before the interrupted process resumes.
    - When the interrupt occurs, the hardware saves the program counter in a special register and jumps to the appropriate interrupt handler.

    - The interrupt handler stores the program counter and the status register in memory, along with the contents of any data registers it plans to use.
    - The interrupt handler runs whatever code is needed to handle the interrupt.
    - Then it restores the contents of the saved registers. Finally, it restores the program counter of the interrupted process, which has the effect of jumping back to the interrupted instruction.

8.2 Context Switching

- Interrupt handlers can be fast because they don't have to save the entire hardware state; they only have to save registers they are planning to use.
- when an interrupt occurs, the kernel does not always resume the interrupted process. It has the option of switching to another process. This mechanism is called a "context switch".
- the kernel doesn't know which registers a process will use, so it has to save all of them. Also, when it switches to a new process, it might have to clear data stored in the memory management unit. And after the context switch, it might take some time for the new process to load data into the cache.
    - For these reasons, context switches are relatively slow, on the order of thousands of cycles, or a few microseconds.
- In a multitasking system, each process is allowed to run for a short period of time called a "time slice" or "quantum".
- During a context switch, the kernel sets a hardware timer that causes an interrupt at the end of the time slice.
- When the interrupt occurs, the kernel can switch to another process or allow the interrupted process to resume. The part of the operating system that makes this decision is the "scheduler".

8.3 Process Life Cycle

- When a process is created, the operating system allocates a data structure that contains information about the process, called a "process control block" or PCB. Among other things, the PCB keeps track of the process state, which is one of:
    - Running, if the process is currently running on a core.
    - Ready, if the process could be running, but isn't, usually because there are more runnable processes than cores.
    - Blocked, if the process cannot run because it is waiting for a future event like network communication or a disk read.
    - Done, if the process has completed, but has exit status information that has not been read yet.
- Here are the events that cause a process to transition from one state to another:
    - A process is created when the running program executes a system call like fork. At the end of the system call, the new process is usually ready. Then the scheduler might resume the original process (the "parent") or start the new process (the "child").
    - When a process is started or resumed by the scheduler, its state changes from ready to running.

    - When a process is interrupted and the scheduler chooses not to let it resume, its state changes from running to ready.
    - If a process executes a system call that cannot complete immediately, like a disk request, it becomes blocked and the scheduler usually chooses another process.
    - When an operation like a disk request completes, it causes an interrupt. The interrupt handler figures out which process was waiting for the request and switches its state from blocked to ready. Then the scheduler may or may not choose to resume the unblocked process.
    - When a process calls exit, the interrupt handler stores the exit code in the PCB and changes the process's state to done.

8.4 Scheduling

- the primary goal of the scheduler is to minimize response time; that is, the computer should respond quickly to user actions.
- Response time is also important on a server, but in addition the scheduler might try to maximize **throughput**, which is the number of requests that complete per unit of time.
- Usually the scheduler doesn't have much information about what processes are doing, so its decisions are based on a few heuristics:
    - A process that does a lot of computation is probably CPU-bound, which means that its run time depends on how much CPU time it gets
    - A process that reads data from a network or disk might be I/O-bound, which means that it would run faster if data input and output went faster, but would not run faster with more CPU time.
    - Finally, a process that interacts with the user is probably blocked, most of the time, waiting for user actions
- The operating system can sometimes classify processes based on their past behavior, and schedule them accordingly.
    - For example, when an interactive process is unblocked, it should probably run immediately, because a user is probably waiting for a reply. On the other hand, a CPU-bound process that has been running for a long time might be less time-sensitive.
- Most schedulers use some form of priority-based scheduling, where each process has a priority that can be adjusted up or down over time.
    - Here are some of the factors that determine a process's priority:
        - A process usually starts with a relatively high priority so it starts running quickly.
        - If a process makes a request and blocks before its time slice is complete, it is more likely to be interactive or I/O-bound, so its priority should go up.
        - If a process runs for an entire time slice, it is more likely to be long running and CPU-bound, so its priority should go down.
        - If a task blocks for a long time and then becomes ready, it should get a priority boost so it can respond to whatever it was waiting for.
        - If process A is blocked waiting for process B, for example if they are connected by a pipe, the priority of process B should go up.

        - The system call nice allows a process to decrease (but not increase) its own priority, allowing programmers to pass explicit information to the scheduler

8.5 Real Time Scheduling

- a program that reads data from sensors and controls motors might have to complete recurring tasks at some minimum frequency and react to external events with some maximum response time. These requirements are often expressed in terms of “tasks” that must be completed before “deadlines”
- Scheduling tasks to meet deadlines is called “real-time scheduling”. For some applications, a general-purpose operating system like Linux can be modified to handle real-time scheduling. Modifications may include:
    - Providing richer APIs for controlling task priorities.
    - Modifying the scheduler to guarantee that the process with highest priority runs within a fixed amount of time.
    - Reorganizing interrupt handlers to guarantee a maximum completion time.
    - Modifying locks and other synchronization mechanisms to allow a high-priority task to preempt a lower-priority task.
    - Choosing an implementation of dynamic memory allocation that guarantees a maximum completion time

9. Threads

- A thread is a kind of process.
- When you create a process, the operating system creates a new address space, which includes the text segment, static segment, and heap; it also creates a new “thread of execution”, which includes the program counter and other hardware state, and the call stack.
- “single-threaded”, means that only one thread of execution runs in each address space.
- **Multithreading** is a feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU. Each part of such a program is called a thread. So, threads are lightweight processes within a process.
- Within a single process, all threads share the same text segment, so they run the same code. But different threads often run different parts of the code.
    - they share the same static segment, so if one thread changes a global variable, other threads see the change. They also share the heap, so threads can share dynamically-allocated chunks.
    - But each thread has its own stack, so threads can call functions without interfering with each other. Usually threads don’t access each other’s local variables (and sometimes they can’t)
- The idea behind a thread pool is to set up a number of threads that sit idle, waiting for work that they can perform. As your program has tasks to execute, it encapsulates those tasks into some object (typically a Runnable object) and informs the thread pool that there is a new task.

9.1 Creating Threads

- The most popular threading standard used with C is POSIX Threads, or Pthreads for short.

9.3: Joining Threads
- Overview
- Describes how to wait for threads to complete using pthread_join.
- Wrapper Function for pthread_join
- join_thread: A wrapper for pthread_join to simplify its usage and handle errors.

Functionality
- A thread can wait for another thread to finish.
- The parent thread typically creates and joins all child threads.
- Example Usage
- Looping over child threads to join them in the order they were created.
- Handling the completion of child threads even if they don't finish in creation order.

9.4: Synchronization Errors
- Addresses the problem of synchronization errors when multiple threads access a shared variable.

Key Issues
- Multiple threads can read and modify a shared variable, leading to inconsistent results.

9.5: Mutex
- Introduces mutex (mutual exclusion) as a solution to synchronization errors.

Key Concepts
- Mutex: A synchronization tool that ensures only one thread accesses a code block at a time.

Mutex Implementation
- Mutex is a wrapper around pthread_mutex_t from the POSIX threads API.
- Includes functions like make_mutex, mutex_lock, and mutex_unlock for easier use of mutexes.
- Mutexes are used to lock critical sections of code to prevent concurrent access by multiple threads.

10. Condition Variables

10.1 The Work Queue
- In some multi-threaded programs, threads are organized to perform different tasks. Often they communicate with each other using a queue, where some threads, called “producers”, put data into the queue and other threads, called “consumers”, take data out.
    - For example, in applications with a graphical user interface, there might be one thread that runs the GUI, responding to user events, and another thread that processes user requests. In that case, the GUI thread might put requests into a queue and the “back end” thread might take requests out and process them.

    - To support this organization, we need a queue implementation that is “thread safe”, which means that both threads (or more than two) can access the queue at the same time. And we need to handle the special cases when the queue is empty and, if the size of the queue is bounded, when the queue is full.
- A condition variable is a data structure associated with a condition; it allows threads to block until the condition becomes true.
- The malloc() function is used to allocate a block of memory in the heap.

10.2 Producers and Consumers

- The producer-consumer problem is a classic example of a multi-process synchronization problem, where the producer and consumer share a common, fixed-size buffer used as a queue.
- Producers: They are responsible for generating data, pushing it into the queue, or producing work for consumers. In a threading context, a producer thread inserts items into a shared data structure.
- Consumers: They take data out of the queue or consume the work produced by the producers. A consumer thread removes items from the shared data structure.
- The interaction between producers and consumers must be carefully managed to avoid two primary issues:
    - Race Conditions: When multiple threads access and modify shared data concurrently without synchronization, it can lead to inconsistent or unexpected behavior.
    - Deadlock: This occurs when producer and consumer threads wait indefinitely for each other, often due to improperly managed access to the shared queue.

10.2: Producers and Consumers

- In this chapter, the concept of producers and consumers in the context of multithreading is explored. The producer-consumer problem is a classic example of a multi-process synchronization problem, where the producer and consumer share a common, fixed-size buffer used as a queue.
- Producers: They are responsible for generating data, pushing it into the queue, or producing work for consumers. In a threading context, a producer thread inserts items into a shared data structure.
- Consumers: They take data out of the queue or consume the work produced by the producers. A consumer thread removes items from the shared data structure.
- The interaction between producers and consumers must be carefully managed to avoid two primary issues:
- Race Conditions: When multiple threads access and modify shared data concurrently without synchronization, it can lead to inconsistent or unexpected behavior.
- Deadlock: This occurs when producer and consumer threads wait indefinitely for each other, often due to improperly managed access to the shared queue.

Chapter 10.3: Mutual Exclusion

- The chapter introduces the concept of mutual exclusion (mutex) as a solution to the problem of race conditions in the producer-consumer scenario. Mutexes are synchronization primitives that protect access to shared resources (like the queue).

- Mutex Usage: Before a thread accesses a shared resource, it locks the mutex. After accessing the resource, it releases (unlocks) the mutex. This ensures that only one thread at a time can access the shared resource, thereby preventing race conditions.
- Queue Safety: By integrating a mutex into the queue data structure, access to the queue becomes thread-safe. It means that concurrent attempts by producer and consumer threads to access the queue will be properly synchronized, ensuring that each thread's view of the queue is consistent and accurate.

10.4 Condition Variables

- Condition variables are introduced as a way to solve another significant aspect of the producer-consumer problem: blocking when certain conditions are not met.
- Purpose of Condition Variables: They are used to block a thread until a particular condition is true. For example, a consumer thread might need to wait (block) if the queue is empty, and a producer thread might need to wait if the queue is full.
- Working Mechanism: When a condition variable is used, a thread that cannot proceed (e.g., a consumer with an empty queue) will block itself on the condition variable. It will remain blocked until another thread changes the state of the shared resource (e.g., a producer adding an item to the queue) and signals the condition variable to wake up the blocked thread.
- Avoiding Spurious Wakes: Threads typically re-check the condition after waking up from a condition variable. This re-checking is crucial because POSIX condition variables do not guarantee that the condition the thread is waiting for is indeed true when the thread wakes up.