Lecture 1 Operating Systems – Main Points – Ahmed Nouralla

- **Operating System:** A layer of software that:
  - Provides the user with a simpler model.
  - Manages computer resources.
- **Why using an OS:**
  - Hardware is very complicated; OS provides a layer of abstraction.
  - Resources are limited and can be used arbitrarily.
    - OS controls the allocation of memory, processors, and I/O devices.
    - OS manages sharing (multiplexing) resources using two ways:
      - Time multiplexing: different programs or users take turns using it (example: CPU, printers).
      - Space multiplexing: each program takes a part of the resource (example: memory, disks).
- **Kernel-mode & User mode:**

Kernel (supervisor) mode	**User mode**
○ Has complete access to the hardware. ○ Can execute any instruction the machine is capable of.	○ Has restricted access to the hardware. ○ Can execute only a subset of the machine instructions ○ I/O are forbidden, to use them, a user makes a “TRAP” system call to the kernel.

- **Computer History:**

<table>
<tr><th><b>1945-1955 - The 1st Generation:</b> Vacuum tubes</th><th><b>1955-1965 - The 2nd Generation:</b> Transistors</th></tr>
<tr><td>▪ No OS.
▪ Computers were slow & huge machines.
▪ Computers produced high heat and consumed high electricity.
▪ Limited storage capacity
▪ Magnetic tapes & Magnetic drums were used for secondary memory
▪ <b>Programming was mechanical (rewiring the machine for each program).</b>
▪ Input through Punch cards & paper tapes.
▪ Output data were printed.
▪ “Colossus” and “ENIAC”.</td><td>▪ Mainframes <b>(room-sized computers in data centers)</b> used batch operating systems, in which there is an operator which takes similar jobs having the same requirement and group them into batches before execution.
▪ Computers were faster and smaller than the 1st generation.
▪ Computers produced less heat and consumed less electricity.
▪ Magnetic tapes & discs were used as a core memory.
▪ <b>Programming was in Machine Language & Assembly Language.</b></td></tr>
<tr><th colspan="2"><b>1965-1980 - The 3rd Generation:</b> Transistors and Integrated Circuits</th></tr>
<tr><td colspan="2">▪ Power consumption was low.
▪ <b>Multiprogramming:</b> computers were capable of doing multiple tasks at the same time
• <b>Spooling: (simultaneous peripheral operations on-line)</b> is a specialized form of multi-programming for copying data between different devices.
▪ <b>Commonly used with printers</b>
• <b>Timesharing:</b> the sharing of a computing resource among many users at the same time through multiprogramming.</td></tr>
</table>

Part 1	Part 2
▪ Small Scale Integration (SSI) and Medium Scale Integration (MSI) technologies were used. • **SSI** is the first generation of integrated circuits, contained only a few transistors. • **MSI** is a generation of integrated circuit chips which contain hundreds of transistors ▪ High-level languages were used. ▪ **Example:** IBM 3/60	▪ Large Scale Integration (LSI) and Very Large-Scale Integration (VLSI) technologies were used. • Portable computers were developed. • **Computers were used in many fields, including** virtual reality, multimedia, simulation, data communication, parallel processing, superconductors, speech recognition, intelligent robots, and artificial intelligence. • **Computer memory has developed** ○ Different types of memories with very high accessing speed & storage capacity aroused. ○ **Redundant Array of Inexpensive/Independent Disks (RAID)** technology for data storage was used. • **Examples:** ○ Honeywell-6000 series ○ PDP (Personal Data Processor) ○ IBM-360 and 370/168 series ○ TDC-316
1980-1990 - The 4th Generation	**1990-Present - The 5th Generation**
▪ Personal computers were developed ▪ The first microcomputer was invented, with the first disk-based OS: • The OS was called CP/M (Control Program for Microcomputers) • Intel 8080 CPU + attached 8-inch floppy disk. ▪ In the 1980s IBM designed the IBM PC and contacted Bill Gates for an operating system.	▪ Mobile Computers were invented.

- **Computer Hardware review (More details in CA notes)**
  - **Processors:** the electronic circuit within a computer that executes instructions
    - **CPU** follows a basic cycle of instruction execution:
      - Fetch Instruction ⇒ Decode instruction ⇒ Calculate Operands ⇒ Fetch Operands ⇒ Execute Instruction ⇒ Write Operand.
    - **CPU contains some registers to hold key variables and temporary results:**
      - **General registers:** hold variables and temporary results
      - **Program counter:** contains the address of the next instruction to be fetched
      - **Stack pointer:** points to the current stack in memory
      - **Program Status Word (PSW):** contains bits of results of comparison instructions, the CPU priority, the mode (kernel or user), and various other control bits.
    - **Two different models of CPU:**
      - **The three-stage model:** Fetch ⇒ Decode ⇒ Execute.
        - It can be parallelized (i.e. different units can be used at the same time).
      - **Superscalar model:** where fetch and decode are pipelined together then held using a holding buffer, then executed.

Multithreading (Hyper-threading)	Multiprocessing
▪ allows the CPU to hold the state of two different threads (lines of execution) and then switch between them in nanoseconds • If one of the processes needs to read a word from memory (long operation), a multithreaded CPU can just switch to another thread thus saving time • Multithreading does not offer true parallelism (as there is no multiple processing power, it's just an optimization to a single processor).	▪ Many CPU chips now contain multiple processors (cores), thus, the OS needs to be able to allocate tasks between them ▪ This provides true parallelism (concurrent execution of process) ▪ An example is the GPU, a processor with thousands of tiny cores which are very good for many small computations done in parallel, like rendering polygons in graphics applications. ▪ Caches can be shared between cores, or separated for each core.

- **Memory:**
    - Memory is organized in a hierarchy, with the top layer having the fastest, smallest, most expensive pieces of memory, and the bottom layer having the slowest, biggest, and the cheapest ones.
    - A typical example: **Registers** > **Caches** > **Main Memory** > **Disk**
    - *Check cache section in CA notes*

Random Access Memory (RAM)	Read-Only Memory
• Serves all CPU requests that cannot be satisfied out of the cache. • Is divided up into cache lines, typically 64 bytes • The most heavily used cache lines are kept in a high-speed cache located inside the CPU	• Does not lose its contents when the power is switched off • Is programmed at the factory and cannot be changed afterward. • On some computers, the bootstrap loader used to start the computer is contained in ROM • EEPROM (Electrically Erasable Programmable ROM) can be erased and rewritten.

-
    - **Flash memory** is non-volatile, rewritable, portable, intermediate in size and speed between RAM and disk.
    - **Complementary Metal-Oxide-Semiconductor (CMOS) memory** is a small amount of memory on a computer motherboard that stores the BIOS settings, such as date and time, boot options, and hardware settings.

- **Disks:** may refer to the multi-platter HDD or the Compact Disc (CD)
    - HDD is slow, sequential in access, large in capacity, and consists of **rotating platters**, coated with a **magnetic material** and paired with **magnetic heads** usually arranged on an **actuator arm** that can read/write data on the **track.**
    - Disks are slow because they have:
        - **Seek time**: the physical positioning time of the read/write head.
        - **Rotation delay (latency):** the amount of time it takes for the disc to rotate to the required position for the read/write head
        - **Transfer time (data rate):** is the amount of time it takes for data to be read or written.

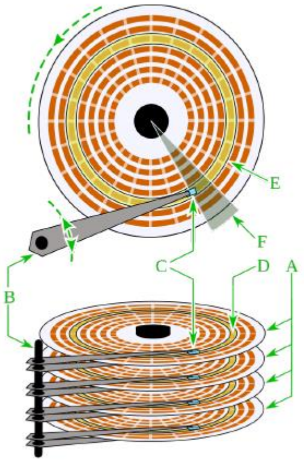
// **Background knowledge** (**check this**)

- Hard drive is made of several identical discs or platters **(A)**
- Each platter has 2 magnetic read/write heads (for each side) **(C)**

  #Platters per disk = 2 * #Heads per disk
- Each platter has several tracks **(E)**
- Each track has several sectors **(F)**
- Each sector has a capacity (in bytes)
- Cylinder: a stack of tracks

  #Tracks per platter = #Cylinders per disk

  #Tracks per cylinder = #Platters per disk



- **I/O Devices**
  - **Generally, consist of two parts:**
    - **The controller** accepts commands from the operating system and carries them out. Every controller has a small number of registers that are used to communicate with it.
      - **I/O port space**: a set of addresses (independent from memory addresses) used by the OS to communicate with the controller registers (each register is assigned an address).
        - **This approach is called "Port-mapped I/O"**
      - Kernel-mode allows the device drivers to send special IN/OUT instructions to read/write these registers.
      - On some computers, these registers are mapped into the operating system's **address space** (the addresses it can use), so they can be read and written like ordinary memory words, without special instructions.
        - **This approach is called "memory-mapped I/O"**
    - **The device**: has a simple standard interface (for data transfer to/from the memory)
      - **Device driver**
        - Software that communicates with the controller, giving it commands, and accepting responses. It has to run in kernel mode.
        - It's usually written for a specific device by its manufacturer and shipped with it.

- **I/O implementation:** the driver starts the device and data transfer begins.

Busy waiting (Programmed I/O)	**Interrupt:**	**Direct Memory Access (DMA)**
The CPU repeatedly checks (polls) the device to see if the transfer is done.	The device sends an interrupt signal to the CPU when it's done. ○ The **interrupt vector** in memory holds the device number which generated the interrupt and uses it to find the corresponding interrupt handler, run it, and returns to the user program.	I/O is done using a special DMA chip that can control the flow of bits between memory and a controller without constant CPU intervention. ○ The CPU programs the DMA chip, telling it what and where to transfer, and lets it go, doing other work. ○ When the DMA chip is done, it causes an interrupt, which is handled in the same way described.

- **Buses:**
    - The system has many buses, each with a different transfer rate and function. The OS must be aware of all of them for configuration and management.
    -

Shared Bus Architecture	Parallel Bus Architecture	Serial Bus Architecture
Multiple devices use the same wires to transfer data which needs an arbiter (the OS) to determine who can use the bus.	Sending each word of data over multiple wires. • For instance, in regular PCI buses, a single 32-bit number is sent over 32 parallel wires.	Sending data sequentially through a single lane. • Can be parallelized, (i.e. having multiple lanes in parallel).

-
    - **Examples:**
        - **Double Data Rate (DDR3, DDR4)** connects CPU and RAM.
        - **Peripheral Component Interconnect (PCIe)** is a bus to an external graphical device.
        - **Direct Media Interface (DMI)** is a link between the northbridge and the southbridge – a hub for all the other devices
        - **SCSI, SATA:** connects hard disks.
        - **Universal Serial Bus (USB):** is a centralized bus in which a “root device” polls all the I/O devices to see if they have any traffic.
            - In the past, I/O devices had fixed registers addresses and might generate the same interrupt, which caused conflicts. **Plug & Play** devices are the devices that don’t need user intervention for configuration and solving these resource conflicts.

- **Booting the computer**
    - **Basic Input Output System (BIOS)** is a program on the motherboard that contains low-level I/O software.
    - **Power-On Self-Test (POST)**: a test performed after the BIOS starts.
        - It tests integrity and sees how much RAM is installed and other basic devices are installed and responding correctly.
    - The computer starts by scanning the buses to detect all the devices attached to them.
        - Then it determines the boot device by trying a list of devices stored in the CMOS memory.
        - Then the boot sector is load into memory and executed.
            - **The boot sector** is the first sector from a boot device
                - It contains a program that examines the **partition tables** to locate the active partition, which may contain a code to load the OS or other applications.
    - The OS then queries the BIOS to get the configuration information. For each device, it checks to see if it has the device driver.
    - Once it has all the device drivers, the OS loads them into the kernel. Then it initializes its tables, creates whatever background processes are needed, and starts up a login program or GUI

Avoid confusion

- **Multiprogramming:** a computer running more than one program at a time (like running Excel and Firefox simultaneously).
- **Multiprocessing:** a computer using more than one CPU at a time.
- **Multitasking:** tasks sharing a common resource (like 1 CPU).
- **Multithreading:** an extension of multitasking.

Common System Calls: link

OS types:

Mainframe OS (2nd generation)	**Server OS**	**Multiprocessor OS**
▪ Is designed to process many jobs (that need a huge amount of I/O) at once. ▪ Offers services for: • **A batch system** that processes routine jobs without interactive user intervention. • **Transaction-processing systems** that handle large numbers of small requests • **Timesharing systems** that allow multiple remote users to run jobs on the computer at once, such as querying a big database. ▪ **Example:** z/OS.	▪ Runs on servers (large PCs, workstations, or mainframes) ▪ Serves multiple users at once over a network. ▪ **Example:** Linux, Windows Server, FreeBSD, and Solaris.	▪ Connects multiple CPUs in a single system. ▪ **Example:** Linux and Windows
Personal Computer OS	**Handheld Computer OS**	**Embedded OS**
▪ For general personal computer usage. ▪ Nowadays contains small-scale multiprocessor OS. ▪ **Example:** Linux, Windows, Mac OS X, and FreeBSD.	▪ For Personal Digital Assistants (PDA) and mobiles. ▪ **Example:** Google Android, Apple iOS	▪ Runs on devices that don’t accept software like microwave ovens, MP3 players, TV sets, cars, etc. ▪ **Example:** Embedded Linux, QNX, and VxWorks.
Sensor Node OS	**Real-Time OS**	**Smart Card OS**
▪ **A sensor node** is a part of a Wireless Sensor Network (WSN). • **WSN** is a group of sensors that communicate over a wireless network. • They are used for monitoring physical conditions of the environment like the temperature, the sound, the wind, etc. ▪ A sensor node is a computer because it has a CPU, RAM, ROM, and sensor(s). ▪ **Example:** TinyOS.	▪ **Hard Real-Time system:** guarantees that some action will happen by a certain time, as in a pacemaker or autopilot system. ▪ **Soft Real-Time system:** missing deadline is not desirable, but acceptable as in a CD reader or a robot controller. ▪ **Example:** QNX.	▪ A tiny operating system that runs on cards that contain a CPU chip like this ⇒ ▪ **Java oriented smart cards:** • Customizable cards that have an interpreter for the Java Virtual Machine (JVM) on its ROM. • It can run small programs downloaded onto the card, called Java Applets.

Lecture 2-3: Refer to “MIT - Practical Programming in C”

Lecture 4

- **Process: an abstraction/instance of a running program, including the current values of the program counter, registers, and variables.**
  - A **program** is a set of instructions, while a **process** is an activity, it has a program, I/O, and a state.
  - Processes allow pseudo (concurrent execution/parallelism) of programs, in which the CPU – running the multiprogramming system – switches between tasks quickly, having only one process running in a single instant of time.
  - A process can also be considered a way to group related resources, that are being used to accomplish some task.
  - Conceptually, each process runs in its own virtual CPU.
  - There is only one physical Program Counter available, when a process is about to run, it’s logical program counter is loaded into memory, when it stops (terminate/get interrupted), the program counter is saved again into the memory.
    - The program counter contains the address of the next instruction to be executed.
  - **Causes of process creation:**
    - System initialization.
      - The (init process/process 0/swapper) is the first process executed by the kernel at boot time, it starts all other processes.
    - Initiation of a batch job
      - Batch jobs run without user interactive user intervention.
    - A “user request” to run a new process.
    - A running process requests a “process creation = fork = clone” system call.
      - **fork() system call** creates an exact clone of the caller (called the parent process) except that the clone (called the child process) has a different Process ID (PID), the parent and the child then go independently from each other.

      - The child process can manipulate its **file descriptor** (FD) to redirect I/O streams, it can then execute a **execve()** or a similar system call which runs a new program with different arguments and environment variables, leading to a different memory image for the child process.
        - The child process can create more child processes, forming a **process hierarchy**, in which a parent and all of its children and descendants form a **process group**.

// Background info

A **file descriptor** (FD) is an abstract indicator used to access a file or an I/O resource, is represented in C using an integer, with 0, 1, 2 reserved for STDIN, STDOUT, and STDERR streams respectively, negative FDs indicates no value or error.

- **Line-buffered FD:** data written to the file described by the FD will be buffered, until a ‘\n’ is seen, or the buffer is full, in both cases the buffer is flushed.
  - **Flushing the buffer** means transferring the data from it to the file.
- **Unbuffered FD**: no buffering occurs whatsoever, and function calls that read or write data occur immediately (and will block).
  - **A blocking function** blocks its thread of execution until the action is completed, like read() system call (by default).
- **Fully-buffered FD:** a fixed-size buffer is used, and read or write calls simply read or write from the buffer. The buffer isn't flushed until it fills up.
- A pipe can be implemented by redirecting STDOUT of some process to STDIN of another process, this can be achieved in C using fork(). The parent process writes data to the pipe, and the data is read by the child process0.

- **Causes of process termination**
  - **Normal exit (voluntary)**
    - A process terminates when it finishes executing its final statement and asks the OS to delete it by using the **exit()** system call.
    - The process may then return a status value (typically an integer, zero in C) to its parent process.
    - All the resources of the process are deallocated by the OS.
  - **Error exit (voluntary)**
    - An error caused by the process, often due to a program bug, the program can choose to handle the error and exit normally.
  - **Fatal error (involuntary)**
    - When the process tries something that is not acceptable, like trying to access a portion of memory that is not allocated, or a user typing a certain command to compile the program and no such file exists.
  - **Killed by another process (involuntary)**
    - A parent process can cause the termination of another process via an appropriate system call.
- **Process states (diagram 1): A process can be**
  - **Running** (in kernel mode or user mode): using the CPU at that instant.
  - **Ready:** runnable, but temporarily stopped to let another process run (ready to run as soon as the scheduler algorithm chooses it to run next).
    - **The scheduler** is the part of the OS responsible for changing processes states and managing processor time.
  - **Blocked:** in a sleeping state, it cannot execute for some reason.
    - Unable to run, waiting for some external event (usually an input) to happen**.**
    - The OS decides to allocate the CPU for another process to run for a while
- **A more detailed list (diagram 2): The process can be**
  - Executing in user mode/kernel mode.
  - Not executing but is ready to run as soon as the kernel schedules it
  - Sleeping and resides in main memory.
  - Waiting for an event and has been swapped to secondary storage (a blocked state)
    - The processes usually reside in the main memory, but they can be moved temporarily to a second memory for memory-management purposes, this action is called **"swapping".**
  - Ready to run, but the swapper (process 0) must swap the process into main memory before the kernel can schedule it to execute.
  - Returning from kernel to user mode, but the kernel preempts it and does a context switch to schedule another process.
    - **Preempting** a process means pausing it temporarily, while storing its state, to be resumed later, such change is called **"context switch"**.
      - In UNIX, kernel processes cannot be preempted.
  - Newly created and is in a transition state; it is not yet ready to run, nor it is sleeping (the initial state for all the processes except process 0)
  - Finished after exit system call and no longer exists, but it leaves a record for its parent process to collect (the final state of a process)
    - In this case, the process is called a "zombie process".
- **Process state transitions:**
  - **Running → Blocked**:
    - OS discovers that this process shouldn't continue and should wait, it can then be blocked automatically or asked to issue a pause or similar system call**.**
  - **Blocked → Ready:**

        - The external event for which the process was waiting (such as the arrival of some input) happens.
        - It can directly switch from Ready → Running if no other process is running at the moment.
      - **Running → Ready:**
        - The scheduler decides that it is time to let another process have some CPU time.
      - **Ready → Running:**
        - The scheduler decides that it is time for the process to get the CPU to run again since all the other processes have run long enough.
  - **Process implementation:**
    - The OS maintains an array of structures called the **process table**, each structure resembles a **process control block**, which contains information about one process such as its PID, program counter, stack pointer, memory allocation, the status of its open files, and other accounting and scheduling information.
    - When some process is interrupted, the interrupt hardware pushes the process data into the stack, then the CPU jumps to the address of the interrupt handler specified in the **interrupt vector**.
- **A probabilistic model of CPU usage**
  - Suppose that a process spends a fraction p of its time waiting for I/O to complete
  - With n processes in memory at once, the probability that all n processes are waiting for I/O (means CPU is idle) is p^n
  - The (CPU utilization/CPU usage/degree of multiprogramming): $1 - p^n$
- **Thread:** the smallest independent line of execution in a process (mini/lightweight process)
  - In traditional OSs, each process has an address space and a single thread of control.
  - **Multithreading:** decomposing an application into multiple threads that run in quasi-parallel.
    - * Check multithreading vs multiprocessing comparison table above *

Threads in the same process share	**Each thread has its own**
• Address space • Global variables • Open files • Child processes • Pending alarms • Signals and signal handlers • Accounting information	• **State:** A thread can be Running, Blocked, Ready, or Terminated. • **Program counter** that keeps track of which instruction to execute next. • **Registers**, holding the current working variables. • **Stack**, containing the execution history with one frame for each procedure called, but not returned from yet, the frame holds the procedure's local variables and the return address to use when the procedure call has finished.

  - **Example**: A word processing application can have 3 threads, one handling the I/O from the user, another one responsible for processing the text visualization and reformatting, and a third one for performing disk backups for the document file.
- **Types of threads:**
  - **User-level threads**
    - Faster to create, manipulate and synchronize
    - User managed
    - Not integrated with OS (uninformed scheduling)
  - **Kernel-level threads**
    - Slow to create, manipulate and synchronize

        - Managed by the OS and acting on kernel
        - Integrated with OS (informed)
- **Why multithreading:**
    - Makes the programming model simpler by avoiding “interrupts”.
    - Threads are easier (faster) to create and destroy than processes.
    - Having threads allows I/O and computing to overlap.
    - Threads with multiple CPUs provide real parallelism.
- **Complications caused by multithreading:**
    - If the parent process has multiple threads, should the child also have them?
    - What happens if a thread in the parent was blocked on a read call from the keyboard?
    - When a line is typed, do both threads get a copy of it?
    - What happens if one thread closes a file while another one is still reading from it?
    - Programs that were written to work in a single-threaded process are hard to convert to work with multithreading for several reasons:
        - Having to deal with the variables that a global to the process.
            - Suggested solutions to this problem:
                - Prohibit global variables)
                - Introduce private global variables specific for each thread.
                - Create special library procedures for creating, setting, and reading thread-wide global variables.
        - Many library procedures are not **reentrant** (multiple invocations of them can’t run concurrently).
            - A solution is to rewrite the whole library to use a “bit” to indicate if the library is being used and prohibit using it while the bit is true, but this will eliminate the potential parallelism.
        - Dealing with non-thread-specific signals (or even with thread-specific signals in user space)
        - Stack management

Lecture 5

- **Inter-Process Communication (IPC):**
    - The mechanism used by the OS to manage the shared data between different processes**.**
    - **IPC** should be carried in a well-structured way not using interrupts.
- **How do processes share data?**
    - Some of the shared data structures, such as the semaphores, can be **stored in the kernel** and accessed only through system calls.
    - Most modern OSs offer a way for processes to **share some portion of their address space** with other processes. In this way, buffers and other data structures can be shared.
- **Race condition**
    - Occurs when the behavior of the system depends on the order of execution of uncontrollable events.
    - **A typical case**: consider a multithreaded system in which two or more threads try to access the same resource or change the shared data at the same time, the result is then dependent on the scheduling algorithm.
- **Ways to avoid race conditions**
    - **Mutual Exclusion with Busy Waiting:**
        - If process B is trying to access a shared resource, while process A is already accessing it, B should stay in a "busy waiting" state until the resource becomes free again.
        - **Busy waiting:** repeatedly checking a condition until it becomes true.
        - **Critical region (section):** a part in the program that accesses a shared resource.
        - **Requirements:**
            1. No two processes may be simultaneously inside their critical regions
            2. No assumptions may be made about speeds or the number of CPUs
            3. No process running outside its critical region may block other processes
            4. No process should have to wait forever to enter its critical region
        - **Implementation methods:**
            - **Disabling interrupts**
                - Each process disables all interrupts after entering its critical region and re-enables them before leaving it.
                - **Issues:**
                    - If the process didn't re-enable interrupts, it will stop the system.
                    - Won't work for multiprocessor systems (only 1 CPU will stop).
            - **Lock variables**
                - Use a "lock" flag to indicate if the resource is being used.
                - Same issues as the "disabling interrupts" method.
            - **Strict Alternation**
                - Use a "turn" variable that indicates which process is allowed to access the resource at the moment.
                - Other processes keep "spinning" in a "busy waiting" state **(spinlock)**.
                - This violates the third requirement, a slow process can be in its noncritical region, its turn came, but it can't take it while blocking a faster process.
            - **Peterson's Solution (a correct way)**
                - A process can enter a critical region if its turn came, or no other processes are interested to enter at the moment.
                - The algorithm uses a interested[] and a turn variable. interested[n] = true indicates that the process n wants to enter the critical section.
                - A process entering the critical region should enable the "interested" flag and take the turn, it waits until its turn comes or other processes are not "interested".
                - A process leaving the critical region should disable its "interested" flag.
            - **Test and Set Lock (TSL) instruction (a correct way)**
                - Hardware-assisted approach; works with multiple CPUs

    - TSL instruction sets the “lock” flag and stores the old value in a CPU register, these two atomic operations cannot be interrupted.
    - A process accessing a shared resource does a “TSL” instruction
      - If the lock was already set, another process is using the resource and will free it soon, try again. Otherwise, the lock is already set, go ahead.
    - Another similar approach uses the exchange “XCHG” instruction.
  - **Busy waiting defect: Priority inversion problem**
    - A high priority task is indirectly preempted by a lower priority task effectively inverting the relative priorities of the two tasks.
    - **Example:** A high priority process H (according to the scheduler) is “busy waiting” while another low priority process L is in the critical region, L is never scheduled when H is running, resulting in both processes waiting forever.
- **Sleep and Wakeup**
  - Instead of wasting the CPU time on busy waiting, use sleep and wakeup system calls.
    - Sleep causes the caller to be suspended (blocked) until another process wakes it up
    - Wakeup accepts the process to be awakened as a parameter and wakes this process up (probably, changing the state from Blocked to Ready)
  - **Sleep and Wakeup defect: Producer-Consumer (bounded-buffer) problem**
    - Two processes, the producer and the consumer share a common, fixed-size buffer.
    - The producer is generating data, adding it to the buffer.
    - The consumer is removing data from the buffer.
    - The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
    - An inadequate solution using only “sleep and wakeup” is to
      - Make the producer sleep when the buffer is full. When the consumer removes an item from the buffer, it wakes up the producer.
      - Make the consumer sleep when the buffer is empty. When the producer puts data into the buffer, it wakes up the consumer.
      - This solution will lead to a “deadlock” (i.e. the consumer found the buffer empty and is about to sleep, the producer already started and sent the wake-up signal to the about-to-sleep consumer, the signal is lost and both will sleep forever).
    - A hotfix is to use a wakeup-waiting bit, but the essence of the problem is still there.
    - A general fix is to use 3 **semaphores** (full, empty, and mutex) or put the producer and consumer procedures into a **monitor**, to ensure that only one of them is active at a time.
- **Semaphores**
  - A semaphore is a variable (non-negative integer) that is used to control access to a shared resource, to avoid synchronization problems (e.g. critical section, race condition).
  - **Instead of the “sleep and wakeup” approach, we use a counting semaphore** to count the number of wakeups saved for future use.
    - Wakeup is increment (up()) and sleep is decrement (down(), if not 0).
    - For the method to work, checking the semaphore, changing it, and possibly going to sleep should be an atomic operation that cannot be interrupted.
      - This can be done by implementing up() and down() as system calls.
    - If multiple CPUs are used, each semaphore should be protected using a “lock” variable with TSL, or XCHG instruction.
  - Semaphores can be used to implement mutex locks (through **binary semaphores** that can take values 0 and 1 only) or to ensure synchronization (messages are sent in a proper way to guarantee that certain event sequences do or do not occur).
- **Mutexes**
  - A shared variable (usually an integer) that can be either locked (non-zero) or unlocked (0).

        - Mutexes are good only for managing mutual exclusion to a shared resource or piece of code
        - They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in the user space
        - When a thread (or process) needs access to a critical region, it calls mutex_lock. If the mutex is currently unlocked, the call succeeds and the calling thread is free to enter the critical region. Otherwise, the calling thread is blocked (yield_thread, no busy waiting) until the thread in the critical region finishes and calls mutex_unlock. If multiple threads are blocked on the mutex, one of them is chosen at random
        - mutex_trylock obtains a lock or returns a failure code but doesn't block the caller.
        - Mutexes can be implemented in user space and don't require kernel calls.
    - **Monitors**
        - A higher-level synchronization primitive/a programming-language construct/a collection of procedures, variables, and data structures that are all grouped in a special kind of module or package.
            - Processes can call these procedures anytime, but internal data structures are hidden.
            - It is up to the compiler to implement mutual exclusion on monitor entries, but a common way is to use a mutex or a binary semaphore.
            - Only one process can be active in a monitor at any moment
                - Each procedure asserts the caller is the only active process before executing.
                - If it's not, it blocks the caller process by doing a "wait" operation on some condition variable, which can be released again by a "signal" operation.
                - After a signal, we can either
                    - Let the signalee run, suspending the signaler.
                    - Require that the signaler exit immediately after sending the signal.
                    - Wait until the signaler finishes, then allow the "signaled" to enter.
            - **Issues:**
                - Monitors make parallel programming less error-prone than using semaphores, however, they are language concepts, C, Pascal, and other languages don't have them.
                - They don't work for a distributed system, where there are multiple CPUs, each having a private memory.
    - **Message Passing**
        - Uses two primitives, send and receive, which (like semaphores and unlike monitors) are system calls rather than language constructs.
        - If no message is available, the receiver can block until one arrives. Or it can return immediately with an error code.
        - **The main issue:** messages can be lost by the network. The receiver might not be able to distinguish a new message from the retransmission of an old one.
    - **Barriers**
        - Intended for groups of processes; Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase.
            - This behavior may be achieved by placing a barrier at the end of each phase.
            - When a process reaches the barrier, it is blocked until all processes have reached the barrier. This allows groups of processes to synchronize
    - **Avoiding Locks: Read-Copy-Update**
        - Having no locks at all, two processes can access the data at the same time, while making sure that no bad effect will happen by freeing the variables that are safe to be accessed.

Lecture 6

- **The scheduler** is the part of OS responsible for changing processes states and managing processor time.
- **Scheduler algorithm:** decides which process to run (from the set of **ready** processes) and which one to suspend when multiple processes are competing for the CPU.
    - Especially important and complex for **multiprogramming systems, networked servers** and not for **batch systems and PCs**, scheduling for mobile devices is sometimes required for battery optimization.
- **CPU-bound process:** spends most of its time computing
- **I/O-bound process:** spends most of its time waiting for I/O
- **Scheduling algorithms categories:**
    - **Preemptive**
        - Picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available).
    - **Non-preemptive**
        - Picks a process, lets it run until it blocks (for I/O interrupt, waiting for another process, or voluntarily releases the CPU); no context switch, thus, improving performance.
        - No scheduling decisions are made during clock interrupts.
- **Scheduling algorithms goals (in general):**
    - **Fairness:** all processes should have a fair share of the CPU.
    - **Policy enforcement:** the algorithm should be carried out.
    - **Balance:** all parts of the system should be kept busy while they have work to do.

	Batch systems	**Interactive Systems**	**Realtime systems***
Proper scheduling algorithm type	a non-preemptive or a preemptive one with a long period is ok since there is one task executing at a time and no interactive user waiting.	preemptive scheduling is necessary, a slow/buggy process shouldn't be able to stop the CPU from its other jobs.	preemption is sometimes not needed since the system is designed for a specific purpose and runs processes that usually execute one quick action then block.
Specific scheduling goals	• Maximizing throughput (jobs per time). • Maximize CPU utilization. • Minimize Turnaround Time** (TAT)	• Minimize response time: user requests over background work. • Proportionality: time to do a task should be proportional to its size, as expected by the user.	• Meeting deadlines, avoid losing data. • Predictability: order of process execution must be predictable, especially for systems involving sound and multimedia, as the human would notice jitter.
Common algorithms	• First-Come First-Served • Shortest Job First • Shortest Remaining Time Next	• Round-Robin Scheduling • Priority Scheduling • Multiple Queues • Shortest Process Next • Guaranteed Scheduling • Lottery Scheduling • Fair-Share Scheduling	• Cooperative scheduling (non-preemptive) • Round-robin scheduling

* For a real-time system: the load can be handled and the system is said to be schedulable if for m periodic events with event i occurring with period P_i and requiring C_i seconds to complete we have $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$

** Turnaround time: the time between submitting a request/task and getting a return value.

- Maximizing throughput doesn’t necessarily minimize TAT; If there is a mix of short and long jobs and the scheduler always picks the short jobs first then both throughput and TAT for long jobs will be maximized.

- **First-Come First-Served (FCFS) – Can be Preemptive or Non-Preemptive**
    - Processes are arranged in a queue, executed from the front to the back, a newly added process (or one that was blocked and now is ready) is added to the end of the queue.
- **Shortest Job First/Next (SJF/SJN) – Non-preemptive**
    - The process with the shortest run time is executed first; minimizing the average TAT.
- **Shortest Remaining Time First (SRTF) – Preemptive (a variant of SJF)**
    - Choosing the process whose remaining run time is the shortest, if a waiting job has a smaller running time, the current one is suspended and the waiting one is executed.
- **Round-Robin Scheduling - Preemptive**
    - Each process is assigned a time interval, called a **quantum**
    - If the process is still running at the end of the quantum, the CPU is preempted and given to another process.
    - If the process has blocked or finished before its quantum, the CPU switching is done when the process blocks.
- **Priority Scheduling – Non-preemptive**
    - Each process is assigned a priority; the runnable process with the highest priority is allowed to run
        - Background processes take lower priority than real-time ones.
    - To prevent high-priority processes from running indefinitely, the scheduler may:
        - Decrease the priority of the currently running process at each clock tick.
        - Assign a quantum to each process (algorithm becomes preemptive in this case).
- **Multiple-Level Queues – Can be Preemptive or Non-Preemptive**
    - It is more efficient to give CPU-bound processes a large quantum once in a while, rather than giving them small quanta frequently (to reduce swapping), but, giving all processes a large quantum would mean poor response time.
    - The solution is to set up priority classes, in which processes in the highest class run for 1 quantum, next one: 2 quanta, next one: 4 quanta (number doubles).
    - If a process in some class uses up all the quanta allocated to it, maybe it’s CPU-bound, thus, moved down one class, and will be given more quantum per run next time, improving efficiency.
- **Shortest Process Next (SPN) – Non-Preemptive**
    - Same as SJF, but for interactive systems.
    - Each user-command execution is considered a separate “job”, the shortest job is run first.
    - We determine the shortest one by making estimates based on its past behavior.
- **Guaranteed Scheduling - Preemptive**
    - Having n processes, optimally, each one should take 1/n of the CPU time.
    - For each process, the system calculates the ratio: $\frac{\mathit{CPU\ time\ taken\ since\ creation}}{n}$
    - The lowest ratio process is run until it becomes bigger than its closest competitor, that one is chosen to run next
- **Lottery Scheduling – Can be Preemptive or Non-Preemptive**
    - Giving processes lottery tickets for various system resources, such as CPU time
    - Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource.
    - Important processes can be given extra tickets, to increase their odds of winning
- **Fair-Share Scheduling – Preemptive**
    - Each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it.
    - Can be considered a round-robin, but at the level of users instead of processes.

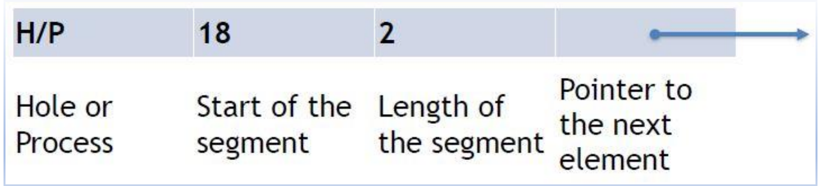
- **Thread scheduling**
    - **Thread Quantum:** the amount of time that the scheduler allows a thread to run before scheduling a different thread to run, if the thread exceeds its quantum, it's suspended.
    - For user-level threads: the kernel is not aware of their existence, it picks a process and gives it control for its quantum, the thread scheduler inside the process decides which thread to run, so the kernel doesn't control the time each thread can take.
        - A thread switch is easier at user-level (some machine instructions) than in the kernel (a full context switch)
        - Using an application-specific thread scheduler can improve performance.
    - For kernel-level threads: the kernel picks a particular thread to run, and gives it control for its quantum, it does not have to take into account which process the thread belongs to.
- **Useful terms:**
    - Burst time (BT): the total time, taken by the process for its execution on the CPU
    - Arrival time (AT): the time when a process enters into the ready state and is ready for its execution.
    - Exit time (ET): the time when a process completes its execution and exit from the system.
    - Response time (RT): the time spent when the process is in the ready state and gets the CPU for the first time.
        - RT = Time at which the process gets the CPU for the first time - AT
    - Waiting time (WT): the total time spent by the process in the ready state waiting for CPU
        - WT = TAT - BT
    - Turnaround time (TAT): the total amount of time spent by the process from coming in the ready state for the first time to its completion
        - TAT = BT + WT
        - TAT = ET - AT

Lecture 7

- **Memory manager:** the part of OS that keeps track of used sections in memory, allocates, and deallocates memory for processes when needed.
- **Memory abstraction:** providing a layer of abstraction between the program and the memory it uses, so the program can send a command like “allocate memory for a variable x” and doesn’t need to bother how and where it’s allocated.
  - In the early stages of computing, there was **no memory abstraction (diagram 3)**.
    - All the physical memory was available to all programs, a program could easily erase/modify data belonging to another one, or even the OS (protection problem)
    - It was not possible to have two running programs in memory at the same time
      - A program needed to be swapped and saved to disk for another one to run.
    - Early computers provided non-optimal solutions to these problems, usually with the help of additional hardware as follows:
        - Memory was divided into blocks, each with a protection key.
        - The PSW register contains the protection key of the currently running program.
        - The hardware traps any attempt by a running process to access memory with a protection code different from the PSW key
      - This idea was used in the early models of IBM 360, but it has a major **drawback**: all programs can reference the whole physical memory; thus, a jump instruction might not work as expected, leading to a crash (relocation problem)
      - A solution was to use **static relocation**: when a program is loaded at address 0020, the constant 0020 is added to every program address during the load process. For example, JMP 8 becomes JMP 28
        - But this is not a general solution, it slows-down loading, and requires extra work to determine the relocatable addresses in programs.
  - **Memory abstraction** solved the protection and relocation problems efficiently by introducing the concept of address space.
    - **Address space:** the set of addresses that a process lives in (can access and use addresses only from this set) independently from other processes.
    - **Dynamic relocation:** mapping each process’ address space onto a different part of physical memory, it is usually needed after **swapping,** and can be achieved by setting values for the base and limit registers.
    - **Base and limit registers** are two special hardware registers. **The base register** contains the physical address of the beginning of the program. **The limit register** contains its length.
      - Using them, programs can be loaded into consecutive memory locations; thus, static relocation is not needed during loading.
      - When the program tries to access memory at some address x, the CPU:
        - Adds the base register contents to x.
        - Compare x with the limit register contents.
      - A disadvantage is the need to do addition and comparison for each memory access, which can be slow.
- **Memory compaction**: combining the multiple **holes** (discrete free chunks in memory, usually caused by swapping) into one consecutive part, by relocating all processes to take addresses downward as far as possible
  - This technique requires a lot of time and is not done usually.
- **Memory allocation for newly created/swapped in processes**
  - If processes are created/swapped in with a fixed size that never changes, the OS allocates exactly what is needed in any free space.
  - If processes’ data segments can grow, for example, by dynamically allocating memory from the heap, a problem occurs whenever this process tries to grow.

        - If a hole (with enough space) is adjacent to the growing process, it can be allocated and the process is allowed to grow into the hole.
        - If the growing process is adjacent to another process, the growing process will either have to be moved to a hole in memory large enough for it, or one or more processes will have to be swapped out to create a large enough hole.
        - If a process cannot grow in memory and the swap area on the disk is full, the process will have to be suspended until some space is freed up (or it can be killed).
        - If it is expected that a process will grow as it runs, the OS allocates a little extra memory whenever it is swapped in or moved.
        - If processes can have two growing segments (the data segment being used as a heap and a stack segment), the memory between them can be used for either segment.

- **Memory management with bitmaps**
    - A bitmap is an array of bits that tracks blocks of memory that are free/in-use.
    - With a bitmap, memory is divided into blocks (called **allocation units)** ranging from few words up to several KBs.
    - Each allocation unit corresponds to a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa).
        - The smaller the allocation unit, the larger memory the bitmap will take.
        - The larger the allocation unit, the smaller the bitmap, but more memory might be wasted in the last unit of the process if the process size is not an exact multiple of the allocation unit.
    - When a k-allocation-unit process is loaded into memory, the memory manager searches the bitmap to find a run of k consecutive 0’s (or ones) to find free space for the process, this can be slow.
- **Memory management with linked-list**
    - To keep track of memory, we maintain a linked list of allocated and free memory segments.
    - Each element in the list has the following structure:

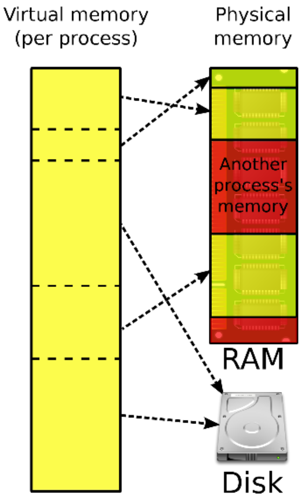


H/P	18	2	→
Hole or Process	Start of the segment	Length of the segment	Pointer to the next element

    - The list can be sorted by address, which makes updating the list when a process terminates or gets swapped-out straightforward (we replace P by H) and merge the consecutive holes (might need a doubly-linked list to make it more convenient).
- **Memory allocation algorithms**
    - **First fit:** the memory manager scans the list of segments until it finds the first suitable hole.
    - **Next fit:** same as “first fit”, except that it starts scanning from the place where it left off last time.
    - **Best fit:** searches the entire list, from beginning to end, and takes the smallest hole that is adequate
    - **Worst fit:** takes the largest available hole, so that the new hole will be big enough to be useful.
    - **Quick fit**: maintains separate lists for some of the most common sizes requested.
- **Memory management techniques:** for dealing with the problem of programs that cannot run due to insufficient space in memory.
    - **Swapping:** already discussed, too slow due to transfer rate from/to the hard disk
    - **Overlay:** a programming method, in which the programmer manually splits the program into self-contained object codes, called overlays, that are kept on disk and can be swapped in/out when needed.
        - When the program starts, only the overlay manager is loaded into the memory, which loads overlay 0.
        - When it’s done, overlay 1 can be loaded either above layer 0, or on top of it (if there was no space).
    - **Virtual memory:**
        - **Informal description:** virtual memory is when the program/process is given a virtual address space to operate in, with virtual addresses, which are not always located in memory

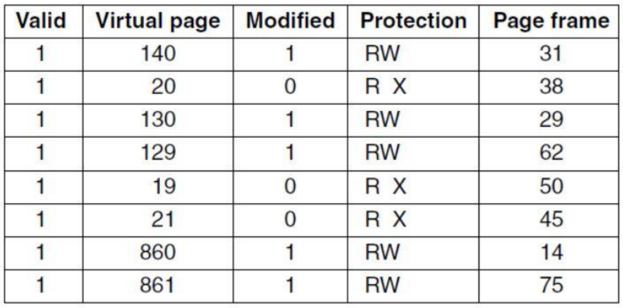
    (they can be located on disk to save space), creating the illusion that there is plenty of continuous memory available for the program, but in fact, it’s the **Memory Management Unit (MMU)** which maps these virtual addresses (covering a bigger range) to the physical ones.

      - **MMU** is integrated into the CPU chip nowadays.



    - **Basic idea:**
      - Each program has its own (virtual) address space, broken up into chunks called **pages** (a page is the smallest unit of data for transfers between RAM and disk).
        - Each (virtual) **page** can be mapped (temporarily) to a **page frame** in the physical memory **(diagram 4)**.
          - Such mapping is stored in a **page table.**
            - A page table typically contains information like:
              - Page frame number
              - Present/absent bit
              - Protection bits (r/w: one bit or r/w/x: three bits)
              - Modified (dirty) bit **M**
              - Referenced bit **R**
              - Caching disabled bit
          - **Paging** refers to the mechanism that is used by the OS to store/retrieve data (in the form of pages) between main memory and secondary storage (typically, RAM and disk, but the concept applies to any system).
      - Not all pages have to be in physical memory at the same time to run the program.
      - When the program references a part of its address space that is not in physical memory, the OS gets the missing piece (from disk) and re-executes the instruction that failed.
        - **A more detailed explanation:**
          - A **present/absent bit** in the page table keeps track of which pages are physically present in memory.
          - If the program references an unmapped address, the MMU causes the CPU to trap the OS (such a trap is called a **page fault**)
          - The OS swaps the page that is needed with a less used **page frame**, changes the map, and restarts the instruction.
- **Problems with paging**
  - If the virtual address space is large, the page table will be large.
    - A solution is to use a multilevel/inverted page table (video)
      - **Multilevel page tables**
        - A paging scheme that is used to avoid keeping all the page tables in memory all the time.
        - **Example (2-level paging):** the top-level page contains PT1, PT2 fields, and an offset, PT1 contains the number of the required lower page table, PT2 field contains the frame number, the offset is used to access the required memory address in the frame.
      - **Inverted page tables**
        - Using one page-table for every memory frame instead of using them for every process, one entry in such a table contains a pair (PID, virtual memory address).
        - This approach has a serious pitfall, we cannot find the physical page using the virtual page quickly, a solution is to use the TLB with a HashMap for each entry (pair) in the table to make the “lookup” faster.
  - The mapping from the virtual address to the physical one must be fast, since it’s done very often, and memory referencing is slow.

- The solution is to use a **Translation Lookaside Buffer (TLB)**: a special hardware memory cache (integrated into the MMU) that stores the recent translations of virtual memory to physical memory for faster retrieval.
- **How TLB works:**
    - When a virtual address is given to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries in parallel.
    - If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB (a protection fault is generated otherwise)



Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

A TLB Structure

    - If there is no match, the MMU does an ordinary page table lookup, then replaces one of the entries from the TLB and with the page table entry just looked up.
        - The page to remove is chosen using a **“page replacement algorithm”**
    - When an entry is purged from the TLB, only the modified bit is copied back into the page table entry in memory since the other values are already there except the reference bit.
    - When the TLB is loaded from the page table, all the fields are taken from memory
- **Software TLB management:**
    - A software can be used to manage a moderately large TLB, leading to better performance (reduce miss rate) and simpler MMU.
        - A **soft miss** occurs when the page is not in the TLB but is in memory.
        - A **hard miss** occurs when the page is neither in TLB nor in memory (a page fault). A page table walk (looking up the map) is performed.
            - A **minor page fault** occurs when the page is in memory, but not in this process’ page table (it may have been brought in from disk by another process).
            - A **major page fault** occurs if the page needs to be brought in from disk.
            - A **segmentation fault** occurs when the program simply accesses an invalid address and no mapping needs to be added in the TLB.

Lecture 8

- **Page replacement algorithms:** decide which memory pages to page out (swap out/write to disk) when a page fault occurs.
    - **Optimal algorithm:**
        - Remove the page that won't be referenced for the longest amount of time.
        - **The algorithm is impossible to implement** because, at the time of the page fault, the OS does not know when each of the pages will be referenced next.
            - However, it can be used to measure the performance of other algorithms by running the program on a simulator and track all page references beforehand.
    - **Not recently used algorithm**
        - Categorize pages based on R&M bits values:
            - Class 0: not referenced, not modified.
            - Class 1: not referenced, modified.
            - Class 2: referenced, not modified.
            - Class 3: referenced, modified.
        - Remove a random page from the **lowest-numbered non-empty class**.
        - **The algorithm is easy to understand, moderately efficient.**
    - **First-in, first-out (FIFO) algorithm**
        - The oldest page to arrive is removed and the new one is added to the tail of the queue.
        - **The algorithm has low overhead, but the oldest page might still be useful.**
    - **Second-chance algorithm**
        - A modification of "FIFO algorithm".
        - Before removing the oldest page, check the R bit
            - If it's 0, the page is both old and unused, remove it.
            - Otherwise, give it a second chance, add it to the tail of a queue.
        - **The algorithm moves pages around constantly, which is ineffective.**
    - **Clock algorithm**
        - A modification of the second-chance algorithm.
        - Keep pages on a circular list and have a "clock hand" that points to the oldest page.
        - On a page fault, the page being pointed to by the hand is inspected
            - If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position.
            - If R is 1, it is cleared and the hand is advanced to the next page.
        - **This process is repeated until a page with R = 0 is found.**
    - **Least Recently Used (LRU) algorithm**
        - On a page fault, Remove the least recently used page.
        - It is necessary to maintain a linked list of all pages in memory, with the MRU page at the front and the LRU page at the rear, the list must be updated on every memory reference. **This is a very time-consuming operation.**
            - **Hardware-assisted implementation of LRU:** use a hardware 64-bit counter stored in each page table entry and incremented every instruction.
            - **Software simulation of LRU:**
                - **Not Frequently Used (NFU) algorithm**
                    - A software counter is associated with each page, initially 0.
                    - At each clock interrupt, the OS scans all the pages in memory and updates the counter by adding the R bit to it
                - **Aging algorithm**
                    - Modification of NFU, the counters have a smaller number of bits (typically 8) and the increment is done differently:

            - The counter is shifted right by 1 bit, then the R bit is added to the leftmost bit.

- **Working set algorithm**
    - **Demand paging:** a process is started with no pages in memory, they're loaded upon request, after several page faults, the number of them becomes relatively small.
        - Slow and wastes CPU time; since it causes thrashing.
        - **Thrashing:** A program's behavior when it is causing page faults every few instructions (due to overuse of the virtual memory resources).
            - With no demand paging, programs don't normally thrash, due to the principle of locality (of reference).
        - **The locality of reference:** the process only references a relatively small fraction of its pages, during any phase of execution.
    - **Working set of a process:**
        - **Definition:** The set of pages that a process is currently using.
            - $W(k,t)$ is the set of the k most recent pages referenced at time t.
        - **Alternative definition:** the set of pages used in the past τ seconds of the CPU execution (virtual) time.
        - **Working set model (Pre-paging):** to keep track of each process's working set and make sure that it is in memory before letting the process run.
    - **The algorithm (diagram 7):**
        - Remove a page that is not in the working set.
        - To determine the working set: several methods are used, one common way:
            - A periodic clock interrupt resets all the R bits in the page table.
            - On a page fault:
                - **The entire page table is scanned**, for each entry
                - If $R = 1$ then update "time of last use" = "current virtual time", **the page is in the working set**.
                - If $R = 0$ and age $> \tau$ then remove the page, **the page is not in the working set.**
                    - $\text{age} = \text{current virtual time} - \text{time of last use}$
                - If $R = 0$ and age $\leq \tau$ then record its age, **the page is in the working set but is a candidate for removal.**
                - If all the pages were in the working set, the one with $R = 0$ and with the greatest age is evicted.
                - Worst case: if all pages have $R = 1$, one is chosen at random, preferably a non-modified one (if exists).
- **WSClock algorithm**
    - Based on the clock, but uses the working set information.
    - Widely used in practice: simple and provides good performance.
    - Uses a circular data structure, storing the list of page frames, each entry contains the "time of last use", the R and M bits.
    - On a page fault: the page at the clock hand is examined.
        - If R = 1, the page has been used during the current tick so it is not an ideal candidate to remove, set $R = 0$, and advance.
        - If $R = 0$ and age $> \tau$ and $M = 0$ then remove the page
            - If $M = 1$, the page should be written to disk before removal. To avoid process switch, the writeback is scheduled and the clock hand advances.
        - Otherwise, the page is in the working set and is not a good candidate for removal.
    - If the hand reaches the start point
        - If at least one write was scheduled, the hand keeps moving until it reaches an entry with $M = 0$

            - If no write was scheduled, all the pages are in working set, find any page with $M = 0$ and remove it, if no such page exists, remove the current one.

- **Design Issues for Paging Systems**
    - **Local versus Global Allocation Policies**
        - **Local Page Replacement**: the algorithm operates on pages used by the process which created the page fault (WSClock/FIFO).
        - **Global Page Replacement:** the page with the lowest age value is removed, regardless of whose page it is (FIFO).
        - If a local algorithm is used and the working set grows, a thrashing will result.
            - If the working set shrinks, local algorithms waste memory
            - To monitor the working set size: use aging bits or an algorithm for allocating page frames.
        - If a global algorithm is used, the system must dynamically allocate page frames to each process as they run
            - One way to manage it is to use a **Page Fault Frequency (PFF) algorithm** that tells when to increase or decrease a process' page allocation (more pages assigned = fewer page faults".
    - **Load Control**
        - A lot of processes thrashing → high load on CPU
        - The number of processes is low → low load on CPU
        - Consider not only process size and paging rate when deciding which process to swap out, but also its characteristics, such as whether it is CPU bound or I/O bound, and what characteristics the remaining processes have.
    - **Page Size**
        - **Small page size:**
            - Reduces internal fragmentation (fewer parts of pages are wasted).
            - Will help to allocate small programs consisting of several phases.
            - A larger page table, larger space is used from TLB.
            - Transferring a small page takes almost as much time as transferring a large page.
        - **Optimal page size**
            - s: average process size, e: page entry size, p: page size (all in bytes)
            - Overhead = $se/p + p/2 \Longrightarrow$ optimal page size = $\sqrt{2se}$
    - **Separate Instruction and Data Spaces**
        - A single address space usually holds both programs and data, it shouldn't be small.
        - Having separate address spaces for instructions (I-space) and data (D-space), each with its own page table and mapping is a good idea that doubles the available address space and is used nowadays to divide the L1 cache.
    - **Shared Pages**
        - Read-only duplicated pages (such as program text) for the same programs can be shared.
        - **Sharing I-space** is easy when Separate Instruction and Data Spaces are used, but it can be problematic when one of the sharing processes is evicted.
        - **Sharing D-space** is more complicated
            - In UNIX, after a fork, the parent and child, each has its own page table, but they point to the same pages.
            - **Copy on write:** When a process updates one of its pages for the first time, OS is trapped and a copy of the modified page is made, both copies are set to R/W so the upcoming writes won't require a trap.
    - **Shared Libraries**
        - **How traditional linker works**
            - All object files are linked and libraries that are named in the command to the linker are scanned.

- Any functions called in the object files but not present there (e.g., printf) are called undefined externals and are sought in the libraries.
- Any functions called in the object files are included in the executable binary if they are found. Other library functions are not included.
- When the linker is done, an executable binary file is written to the disk containing all the functions needed.

- Statically linking hundreds of programs with all these libraries wastes RAM, that's where **shared libraries** come in.
  - When a program is linked with shared libraries, the linker includes a small stub routine that binds to the called function at run time
  - Shared libraries may be loaded either when the program is loaded or when functions in them are called for the first time
  - The entire library is paged in into memory, page by page, as needed, so functions that are not called will not be brought into RAM
    - A DLL can be updated without recompilation of programs using it.
  - Shared libraries won't work with position-dependent code
    - A solution is to compile shared libraries with a special flag telling the compiler not to produce any instruction that uses absolute addresses.

- **Mapped Files**
  - The idea of shared libraries can be generalized to shared files between processes
    - A memory-mapped file is loaded to pages, modified, and then written back to memory after the process exits or un-maps it.
  - If two or more processes map onto the same file at the same time, they can communicate over shared memory, by live-writing and reading the same file.
- **Cleaning Policy**
  - **Paging daemon:** a background process that sleeps most of the time, but is awakened periodically to inspect the state of memory.
    - If too few page frames are free, it begins selecting pages to evict using some page replacement algorithm, to optimize paging.
- **Virtual Memory Interface**
  - In some advanced systems, programmers have some control over the memory map and can use it in nontraditional ways to enhance program behavior.

Lecture 9

- **OS page-related work**
    - **On process creation**
        - Allocates and initialize space in RAM for the page table.
        - Allocates and initialize space on disk for the swap area.
        - Record this information in the process table
    - **On process execution**
        - Reset the MMU and flush the TLB
        - Make the process page table current.
        - Optionally, bring some pages into memory to reduce page faults.
    - **On page fault**
        - Save program counter, general registers, and other volatile info.
        - Check registers to determine the virtual address caused the fault.
        - Get the required page from disk and assign it a free clean page frame.
            - If no frame was available, a page replacement algorithm is used.
            - If the frame was modified, the page overwrites the one in the disk, suspending the process temporarily (context switch).
        - Update the page table to reflect the changes made.
        - Reload saved state and re-execute the instruction that caused the fault.
    - **On process termination**
        - Release the resources, namely: page table, pages, and swap area on the disk
        - If a page was shared, release only when the last page using it terminates.
- **Implementation issues related to paging**
    - **Instruction backup:**
        - **Problem:**
            - How to determine the starting address of the instruction that caused a page fault, to restart it after the page fault has been resolved.
            - The program counter could or couldn't have been already **autoincremented,** it depends on the microarchitecture, there is no way for the OS to guess.
        - **A solution**:
            - Copy the program counter before each instruction is executed and store it into a **hidden internal register**, store information about the registers that have already been autoincremented or autodecremented, and by how much into **another register**.
    - **Locking pages in memory**
        - **Problem:**
            - Consider the situation where a process waiting for I/O is suspended for another process to run, the second process gets a page fault.
            - One of the pages that belong to the first one is chosen by the algorithm to be evicted.
            - After the I/O completes, it will write on the old place that currently belongs to the second process!
        - **Possible solutions**
            - Lock (pin) pages engaged with I/O, so they can't be evicted.
            - Do all I/O in kernel buffers and copy to user pages later.
    - **Separation of policy and mechanism (diagram 8)**
        - **Problem**: the paging system is complex and runs entirely in kernel space.
        - **Possible solution:** run most of the memory management system at the user-level, by dividing it into:
            - A low-level MMU handler (machine-dependent code describing in detail how the MMU works).

        - A page fault handler that is part of the kernel (machine-independent code describing most of the mechanism for paging).
        - An external pager running in user space (describing the paging policy)
            - Creates the page table and allocates space when necessary.
            - Can contain the page replacement algorithm, but the access R&M bits should be given.
        - This implementation is more flexible and provides more modular code, However, it has the overhead of crossing the user-kernel boundary several times.
- **Segmentation: dividing memory into segments**
    - **Segments:** logically independent (usually very large) address spaces that have a dynamic length.
        - Addresses in the segmented memory are pairs: (segment number, address in the segment).
        - A segment might contain a procedure, an array, a stack, or a collection of scalar variables, all of the same type (usually).
        - Segmentation facilitates linking, protection, and sharing procedures/data between processes.
    - **Checkerboarding/External fragmentation**
        - Dividing memory into several chunks, some contain data, others are holes.
        - This can be solved using memory compaction.
        - If the segments are large, we can page them instead.
- **Segmentation with paging:**
    - **MULTICS**
        - Programs are assigned a segmented virtual memory.
        - Each segment is treated as a virtual memory that can be paged and has a page table.
        - Each segment has a descriptor, indicating whether (a part of) the segment is in memory.
            - If it's in memory, the descriptor will contain a pointer to its page table.
            - Descriptor also contains the segment size, the protection bits, and other items, it doesn't contain the addresses of segments that are in the disk.
        - Segment descriptors are stored in the segment table.
            - The segment table is itself a segment and was paged.
        - Addresses and memory referencing in MULTICS **(diagram 9)**
    - **Intel x86**
        - x86-64 CPUs: segmentation is obsolete
        - x86-32 CPUs: segmentation was used.
            - The virtual memory model contained two tables:
                - Local Descriptor Table (LDT): for each program's local segments.
                - Global Descriptor Table (GDT): shared by all programs, for system segments
            - Addresses and memory referencing
                - Segments were accessed using a selector that's loaded into segment registers.
                - Addresses have the form (selector, offset) and are converted into linear (virtual if paging is enabled, physical otherwise) addresses.
                    - Dir field is used to get the page table in case of paging.
                - Segment descriptor contains base address and limit, selector uses them (sometimes adding an offset), to get the **linear address**.
                - During execution, segments are stored into (Code Segment) and (Data Segment) registers.

Lecture 10

- **Files:**
    - Logical units of information created by a process, are independent of each other, have a name and an extension.
    - **File contents:** can be
        - An unstructured sequence of bytes, to be translated by the user-level program
            - Used in Windows and UNIX.
        - A sequence of fixed-length records, each with some internal structure.
        - A tree of (non-fixed) records, each record has a key used to order the tree for quick access.
    - **File types:**
        - **Regular files (ASCII/Binary):** contain user information.
            - **ASCII files** contain lines of text, lines end with Carriage Return (CR), or Line Feed (LF) characters.
                - LF '\n' adds a new line.
                - CR '\r' returns to the beginning of the line (like typewriters).
                - UNIX use LF, Windows uses CRLF.
            - **Binary files** have a structure known to programs using it.
                - **Example:**
                    - Executable files (header, text, data relocation bits, symbol table).
                    - Archive (header, object module).
        - **Directories:** system files for maintaining the structure of the file system
        - **Character special files:** used to model serial I/O devices (terminals, printers, networks)
        - **Block special files:** used to model disks.
    - **File access:** can be sequential or random.
    - **File attributes:** fields and flags for customization (protection, password, creator, owner, …)
    - **File operations:**
        - **Create, Delete, Open, Close, Read, Write, Append, Seek, Get/Set Attributes, Rename.**
            - **Seek** repositions a file pointer to a specific position inside the file.
- **Directories:**
    - System files for structuring files, commonly known as folders.
    - Typically organized in a tree hierarchy, with the root directory as the parent that contains all of the child subdirectories
        - Paths in the tree can be absolute (from root) or relative (from the current working directory).
    - **Directory operations: Create, Delete, Opendir, Closedir, Readdir, Rename, Link, Unlink.**
        - A directory needs to be opened and closed like a file to be used.
        - Only empty directories can be deleted.
        - Readdir returns the next entry in the open directory.
    - **UNIX implementation of directories:** a directory is a set of pairs (i-number, ASCII name)
        - Each pair is called a **directory entry**, a file is accessed using it.
        - I-number is an index in the table of **inodes**, one node per file containing all info about it.
        - Creating a **hard link** means adding a new entry with the same i-number and a different ASCII name, effectively creating a different name (alias) for the same file
            - A file with no hard links is removed by the system.
        - A **soft link** (shortcut) is a file containing a text of a filename or its location.

- **File storage implementation:**

Method	Contagious allocation	Linked-list allocation	File allocation table	I-nodes
Description	Store each file as a contagious run of disk blocks.	A disk block contains data and a pointer to the next block.	A modification of linked list allocation, store pointers in a **file allocation table** in memory instead of storing them in disk blocks.	An Index-node is associated with each file, containing its attributes and disk addresses of its blocks.
Advantages	- Simplicity - Read performance.	- No disk fragmentation.	- Random access is faster - Blocks contain data only.	- Only inodes for open files are in RAM. - Memory required: an array of inodes with size = max possible number of open files at once.
Disadvantages	- Disk fragmentation (after the file is removed, the disk contains discrete holes).	- Slow random access - Wasted space for pointers.	- Big table in RAM.	

- **File system:** controls how data on storage devices (disks) are stored and retrieved
  - Disks can be divided into **partitions with independent file systems.**
    - Sector 0 of the disk is called the **Master Boot Record (MBR)** used to boot the computer.
      - MBR also contains the partition table containing the starting and ending addresses of each partition.
      - When booting, MBR locates the active partition and loads its boot block, which loads the OS stored in the partition.
  - **In UNIX,** a partition contains a superblock describing the file system, it contains the magic number for the file system, the number of blocks it has, and other info.
  - **Log structured file system**
    - The entire disk is a big circular buffer called log, inodes are used, but has no fixed position.
  - **Journaling file system**
    - Writes a log of steps needed to accomplish a task before doing it.
      - If the task is done successfully, the log is erased.
      - If the system crashed, the system checks the log upon reboot for pending operations.
  - **Virtual file system**
    - An abstraction of a file system, it has an interface for the common file system operations, which calls the underlying file systems that actually manipulates the data.
- **Disk space management**
  - When storing files in a disk divided into blocks of fixed size, the fixed size is an important factor.
  - **A large block size** means that all files, even 1-byte will take a whole block, **wasting space.**
  - **A small block size** means that most of the files won't fit in a block and will be divided, thus multiple seeks and rotational delays are needed, **reducing performance.**
  - These days, 64KB block size is ok for performance, since space is no longer a big problem.
- **Keeping track of free blocks**
  - Using a linked list of disk blocks.
  - Using a bitmap (a bit for each block).
- **Disk quotas:** Multi-user OSs often provide a mechanism for restricting disk usage for each user.

- **File system backup**
    - Backup is needed to recover from disaster/stupidity
    - Not all files have to be backed up, either because they are shipped with the system, or because it can be dangerous (like I/O special devices in UNIX /dev)
    - **Incremental backup (dump): the** backup is done only for blocks that were changed since the last backup, they are faster and more efficient than
        - **A physical dump** starts at position 0 and writes all blocks up to the last one
        - **A logical dump** starts at one specified directory recursively dump all changed files there
- **File system consistency**
    - If blocks are modified, and the system crashes before they are written out, the FS is said to be in an **inconsistent** state.
    - Utilities to check consistency: sfc in Windows, fsck in UNIX.
    - Consistency can be checked by creating two lists, one for free blocks, the other one for blocks in use, if a block is present/absent/present multiple times in both lists, the file system is inconsistent.
- **File system performance**
    - Strategies to improve disk read/write performance
        - **Block (buffer) cache.**
        - **Block Read Ahead**: try to get blocks into cache before they are needed, to increase the hit rate.
        - **Reduce the disk-arm movement**, by putting blocks that are likely to be accessed in sequence close to each other and in the same cylinder.
- **Disk defragment**
    - Reorganize disk blocks to have the free space as a single contagious block following the used space.
        - Windows has a tool “defrag” that does this.

Lecture 11

- **I/O devices categories**

Block devices	Character devices	
Stores/transfers information in fixed-size blocks, each one has an address	Deliver/accept a stream of characters, not addressable.	Some devices don't fit in any category.
Examples: Hard disk, Blu-ray disk, USB sticks	**Examples:** Printer, mouse, …	**Examples:** Clocks, memory-mapped screens, and touch screens.

- **I/O Hardware**
  - **I/O unit consists of (refer to 1st lecture)**
    - **The controller (adapter): "the electronic part"**
      - On PCs, it can be
        - A chip on the parent board
        - A printed circuit card that can be inserted into a PCIe expansion slot
    - **The device "the mechanical part"**
      - The mechanical device itself; can be connected to the computer to transfer data.
  - **I/O controller**
    - **Has several (control) registers**, used for communication with the CPU
      - The CPU can **write** to these registers, to ask the device to deliver/receive data, switch itself on/off, or perform some action.
      - The CPU can **read** these registers to get the state of the device, whether it's idle or not.
    - May have a **data buffer,** used for communication with the OS
      - **The data buffer** is a region of memory used to temporarily store data while it is being moved from one place to another.
      - The OS can write to the **video ram** (a data buffer) of a **screen** (the device) the data of pixels to be displayed by the screen.
  - **Methods for performing I/O communication between CPU and device:**
    - **Port-mapped I/O**
      - Each register is assigned an address (**I/O port number**)
      - **I/O port space**: a set of protected addresses used **only** by the OS to communicate with the controller registers
        - I/O port space is independent of memory addresses.
      - The kernel provides special IN/OUT commands to read/write registers. Examples:
        - **IN** REG, PORT // stores contents of PORT register in REG
        - **OUT** PORT, REG // write the contents of REG into PORT register
    - **Memory-mapped I/O**
      - Control registers are mapped into ordinary memory addresses
      - **Advantages:**
        - **No assembly code is needed (will make I/O faster)**
          - IN/OUT assembly instructions add overhead to controlling I/O.
          - Control registers can be directly accessed as C variables.
          - An I/O device driver can be completely written in C
        - **No special protection is needed**
          - The OS can simply reserve register addresses, not assigning them to user address space.
          - OS can grant restricted access to a specific device if each register address is stored on a separate page.
      - **Disadvantages**

- A busy waiting assembly code may loop forever if the state of the device is polled from a register as the register contents may get cached in memory without polling the device each time.
  - A solution is to disable caching, but this will add more complexity to the system doing the “selective caching”.
- The process of all devices checking the bus signals can get complicated if there are multiple buses, which is the case for modern computers. **Check (*)**

- **A Hybrid scheme:**
  - Uses memory-mapped I/O for data buffers and I/O ports for control registers

- **How these methods work**
  - When reading from control registers/data buffers, the CPU sends on the bus: a READ signal and a signal to tell which space (port/memory) is needed.
  - The I/O device (in case of port space) or the memory (in case of memory space) responds to the signal and sends back the required data.
  - If only memory-mapped I/O is used, all the I/O devices and memory modules have to examine the bus signals to check if the signal is sent to them and respond **(*)**

- **Direct memory access (DMA)**
  - DMA is a memory-to-device communication method that bypasses the CPU.
  - It allows doing I/O or memory-to-memory transfers (like disk transfers) without keeping the CPU busy while the transfer is in progress.
  - **“Disk-to-memory” transfer without DMA**
    - When the OS is starting, it needs to get data from the disk.
    - The disk controller reads a block of data bit by bit and stores it in its internal buffer
    - It computes the checksum to make sure no reading errors happened
    - When the block is fully in the buffer, the disk controller interrupts the CPU which starts reading from the controller buffer (using byte or word as a unit) and transferring to the memory.
  - **Disk-to-memory transfer using DMA (diagram 10)**
    - The CPU program the DMA by setting its controller **registers** to know what to transfer and where and move on doing other work.
      - **Memory address register**: of the disk (device) to read from/write to
      - **Byte count register:** contains the size of data to be transferred.
      - **Control registers:** specifying
        - The I/O port to use
        - The direction of transfer (read/write)
        - The transfer unit (byte/word) at a time
    - The CPU also commands the disk controller to read data and store it in its internal buffer
    - The DMA has direct access to the bus, it requests a transfer from the disk controller to the memory
      - **Bus access modes:** word-at-a-time/cycle-stealing, block/burst, fly-by
    - When the disk-to-memory transfer is done, the disk controller sends an acknowledgment signal to the DMA, which increments the **Memory address** and decrement the **byte count** registers
      - If there are still more bytes to transfer (byte count > 0), the process is repeated.
      - Otherwise, the DMA sends an interrupt to the CPU when done.
    - When the OS is starting, it doesn’t need to get data from disk since it’s already there.
  - DMA controllers can be **improved** (and get more complex) to handle multiple transfers at the same time, a different acknowledgment line on the bus is often used to remove signal ambiguity.
  - DMA controllers usually operate on **physical addresses,** to make it work
    - The OS has to convert addresses before writing them in the DMA register (most used)
    - Write the virtual addresses in the DMA and let it communicate with the MMU for translation.

Lecture 12

- **Goals of I/O software**
  - **Device independence:** code doing I/O should be generic for all different I/O devices.
  - **Uniform naming:** names used to communicate with the device shouldn't depend on the device.
  - **Low-level Error handling:** error handling should be done at the lowest level possible before it affects the user.
  - **Synchronous (blocking) transfers:** I/O device transfers are mostly **asynchronous (interrupt-driven)**, but user programs are much easier to write if the I/O operations are blocking, so it's up to the OS to make operations that are actually interrupt-driven look blocking to the user programs.
  - **Optimize buffering:**
    - For some I/O devices (e.g. digital audio/video), output data should be stored in a buffer in advance to avoid **buffer underrun** (e.g. lagging video on YouTube)
    - On the other hand, buffering data too many times affects performance.
- **I/O implementation** **(refer to the 1st lecture)**

Programmed I/O with Busy waiting	**Interrupt-driven I/O:**	**Programmed I/O with DMA (refer to 1st and 11th lecture)**
The CPU repeatedly checks (polls) the device to see if the transfer is done.	When I/O transfer is done, the device causes **an interrupt** by asserting a specific signal on a bus line. The signal is detected by the **interrupt controller** chip, which sends the number of the device that caused the interrupt to the CPU, to check the **interrupt vector** for the corresponding **interrupt handler** and run it (change the program counter to point to the beginning of the handler procedure, then return to the user). **Interrupt types** **1. Precise interrupt**: leaves the machine in a well-defined state - Program Counter is saved. - Instructions **before** PC are fully executed. - Instructions **after** PC are never executed - Current instruction execution state is known. **2. Imprecise interrupt:** not precise	Same with busy waiting, but the DMA controller is doing the work instead of the CPU
This keeps the CPU busy waiting for I/O to finish, thus wasting time	This allows the CPU to do other work while waiting for the transfer to finish	This totally frees the CPU while I/O is being done.

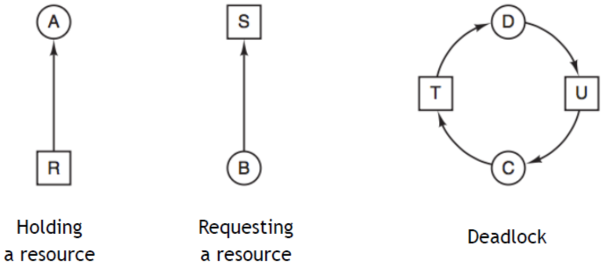
- **I/O software layers** **(diagram 11)**: hierarchy of the software types used for I/O, each layer has a function and an interface to other layers
  - **User-level I/O software**
    - A small portion of I/O software consists of libraries linked together with user programs
  - **Device-independent OS software**
    - Typical functions include:

- **Uniform interacting with device drivers:** to have the same interface with all I/O devices, an OS shouldn’t be modified for each device
- **Buffering:** loading data into memory (storing it in a buffer) before actually using it. Data transfer for I/O devices can be
  - **Unbuffered** (character by character)**: inefficient**
  - **Buffered in user space: risky** (buffer may get paged out during transfer)
  - **Buffered in the kernel** (then copied to user space once full)**: efficient**
    - To handle the characters that arrive while the kernel buffer is full and being copied, a **double buffer** or **a circular buffer** can be used.
- **Error reporting:** there can be two types of errors
  - **Programming errors**
    - Asking for an impossible operation (e.g. writing to an input device)
    - **Action:** Just report the error to the caller
  - **Other I/O errors**
    - e.g. writing to a damaged block, accessing a disabled device
    - **Action:** up to the device driver.
- **Allocating and releasing dedicated devices**
  - OS blocks callers that try to access an unavailable device
  - After the device becomes available, the queue of blocked callers is served
- **Providing device-independent block sizes**
  - OS provides uniform data transfer and storage units for all devices, regardless of their actual specifications.

- **Device driver**
  - Low-level software that communicates with the device controller.
  - Usually written for a specific device by its manufacturer and shipped with it.
- **Interrupt handler**
  - Does all the required actions for handling I/O interrupt: saving registers, reserving memory for handler procedure, running the procedure, and returning to the user.

Lecture 13

- **Resource:** a hardware device, or a piece of information, utilized by a process, resource allocation can be managed by the system or the user process.
  - **Preemptable resource:** can be taken from the process utilizing it with no harmful effects
  - **Non-preemptable resource:** cannot be taken without causing failure
- **Resource allocation**
  - A **counting semaphore** can be used to allow a user process to manage resource allocation as follows
  - **To use a resource**
    - **Request the resource** (decrement the semaphore)
    - **Use the resource** (decrement the semaphore)
    - **Release the resource** (increment the semaphore)
- **(Resource) Deadlock:**
  - A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
  - **Four conditions that should hold to have a resource deadlock**
    - **Mutual exclusion**: a resource cannot be assigned to multiple processes at once.
    - **Hold-and-wait**: a process can hold a resource and ask for another.
    - **No preemption**: resources are not preemptable.
    - **Circular wait**: each process is waiting for a resource held by the next member of the chain.
  - **Deadlock modeling as a directed graph** (square: resource, circle: process, directed edge)
  - **Dealing with deadlocks:**
    - **Ostrich Algorithm:** ignore the problem, since it happens rarely
    - **Detection and recovery:**
      - Let deadlocks occur.
      - Detect them (by modeling the resource/processes graph and checking for cycles).

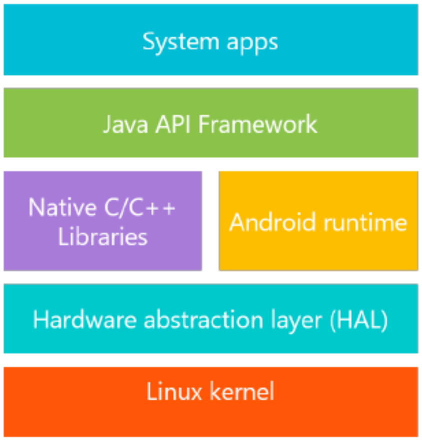


      - Take action if a cycle is detected
        - **Preemption:** take the resource temporarily from the process, give it to another process (depends on the resource)
        - **Rollback:** return to an earlier checkpoint of the process state, and avoid deadlock by waiting for the resource to be available.
        - **Killing process:** kill one or more processes involved, choose a process that can be rerun without damage.
    - **Dynamic avoidance:** by careful resource allocation
      - The system must be able to decide whether granting a resource is safe or not, and grant it only if it's safe.
      - A state is said to be safe if there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
      - An example is the **banker algorithm**, in which the banker (system) grants the customers (processes) the loans (resources) only if it leads to a safe state (with enough free units for a process reaching max), otherwise, the request is postponed
    - **Prevention:** by structurally negating (attacking) one of the four required conditions
      - **Mutual exclusion:** by avoiding resource allocation unless absolutely necessary
      - **Hold-and-wait**:
        - By forcing the process to request all resources before execution
        - Or forcing it to temporarily release all its resources before trying again to get them all at once (**Livelock**)

        - may not work if all processes are doing the same)
      - **No preemption**: some resources can be virtualized (i.e. printer)
      - **Circular wait**:
        - Forcing the process to use only one resource at a time
        - All the resources are numbered, a process is granted access only to a resource with a number greater than the one it had before.
- **Communication deadlock**
  - When there is a circular wait loop of processes trying to send messages to each other
  - A technique to solve it is timeouts (if the time ran out, the message is assumed to be lost and sent again)
- **Starvation:**
  - A process is never getting served, even though there is no deadlock
  - Can be avoided by using FIFO resource allocation algorithm

Lecture 14

- **Android:** a software company, was focusing on cameras, then switched to smartphones
  - **First android device:** HTC Dream – 2008
- **Android design goals:**
  - Provide an open-source platform for mobile devices
  - Support 3rd party apps with a robust API
    - Allowing these apps to compete fairly.
    - Maintaining security, so users don't have to trust these apps
  - Support quick and frequent process switching.
  - Manage processes, simplifying the user experience



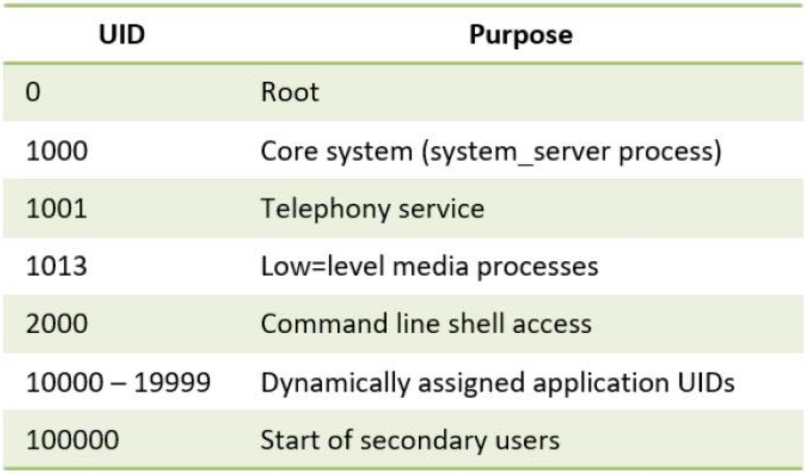
- **Android architecture**
  - **Linux kernel:**
    - **Used for basic functionalities such as**
      - Threading
      - Low-level memory management.
      - Power Management.
      - Inter-Process Communication
      - Direct access to hardware via drivers.
    - **Android uses the standard Linux kernel, with few significant extensions for**
      - **Power Management.**
        - **Wake locks**: Android mechanism for managing how the device sleeps.
          - When the screen is on, the system is holding a wake lock.
          - When a wake lock is held (by system/app), the device cannot go to sleep.
      - **Binder Inter-Process Communication (Binder IPC).**
        - **Binder architecture**
          - **Binder kernel module**: implements the cross-process communication
          - **Binder user Space API:** an object-oriented interface that communicates with the kernel module
          - **High-level interface-based programming model:** used by the applications to declare their IPC interfaces.
        - **How it works**
          - Applications communicate with the **interface-based programming model** through method calls.
            - **Android Interface Definition Language (AIDL):** an interface compiler, generates a source code from an abstract description of an interface.
          - The interface sends and receives **transactions (messages/function calls)** in queues to the **user space API**
            - Binder IPC is thus message-based, not stream-based.
          - The user space API sends command codes to the **kernel module**, which does the operation and returns the result code.
      - **Low Memory Killer (LMK)**
        - Called when the device is low on memory
        - Categorizes process into priority groups, then kills one process from the lowest group.
      - **Anonymous Shared Memory (Ashmem)**
        - Allows non-related processes to share memory maps.
      - **RAM Console and Log Device**
        - Used to read/write logs.

                - **RAM console** stores kernel messages.
                - **Log device** adds a special logging module for user processes.
            - **Android Debug Bridge (ADB)**
                - A command-line tool that is used by the desktop developer to communicate with the android device through WiFi/USB.
                - It’s used to install, uninstall, and debug apps.
    - **Hardware Abstraction Layer (HAL)**
        - Provides standard interfaces for device hardware (camera, Bluetooth, etc.) to the developers for creating user/system using Java API framework.
    - **Android Runtime (ART)**

Dalvik	**Android Runtime (ART)**
Register-based VM using by early versions of android as the runtime environment.	A runtime environment used currently by android (replacing old **Dalvik**)
Converts java bytecode (. class files) into Dalvik executables (. dex files)	Translates Java bytecode, supplied through . dex files to native machine instructions
Uses Just-In-Time compilation	Uses Ahead-Of-Time (AOT) compilation (apps are compiled upon installation)
Lower memory requirements and storage space	Higher memory requirements and storage space
Slower to run applications, but installation will take less time	Faster to run applications, but installation will take more time
Allows multiple instances to run at once.	Allows multiple instances to run at once.
	- Improved garbage collection. - Improved development and debugging tools (profiler, more debugging features, more diagnostic details)

    - **Native C/C++ libraries**
        - HAL and ART require C/C++ libraries to work, these libraries can also be accessed by app developers through the Java API framework
            - OpenGL ES (native C/C++ library) can be accessed through Java OpenGL API to support 2D/3D graphics.
    - **Java API Framework**
        - Set of APIs written in Java to simplify the use of system components and features, allowing developers to create user/system apps.
    - **System apps**
        - Default apps that can be directly used by users or by developers embedding the functionalities of system apps into their apps.
- **Booting an Android** **(Diagram 12)**
    - When the power button is held, the “Boot ROM” code is executed, the code is stored in a predefined, hardcoded location, it loads the bootloader into the RAM and executes it.
    - The bootloader detects the RAM and sets the network, low-level memory management, and security options.
    - As the kernel executes, it sets the cache, protected memory, and schedules different load drivers, then looks for init system files.
    - **“/system/etc/init”** is an important directory in an android system
        - **Init contains 2 programs**
            - **init. rc**: runs generic initialization instructions.
            - **init.** [**machine_name**]. **rc**: runs machine-specific initialization instructions.
        - **Directories such as “/dev”, “/sys”, “/proc” are mounted in init.**
        - **Init runs daemons such as:**

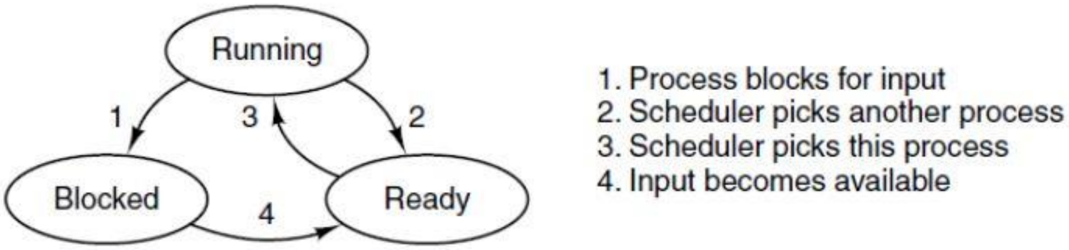
            - **Installd** receives requests from Package Manager Service via Linux Domain Socket and can be used to install APKs with root permission.
            - **servicemanager**: acts as an index for all Binder services running in the system.
            - **adbd**: manages the connection between the target and the host's ADB command.
        - **Init runs the "zygote" process; the parent of all other processes.**
            - "Zygote" process starts the system server which contains all OS services
            - With each request to run a new application, the zygote process is forked, a child is created with a VM and the application is bound to the thread of this process.
- **Android Package (APK)**
    - A container for everything that makes an android app, it contains
        - A manifest file: contains data about the application (name, functionality, and how to run)
        - Resources needed by the application
        - The application code (ART bytecode, or native library code)
        - Author signing information
    - **Package manager:** an API that allows querying data about the installed packages from their manifests.
    - **Package installer:** the default application for installing APKs
- **Android application components**
    - **Activities:** the part of the application that interacts with the user through an interface.
    - **Services:** a component that performs operations in the foreground, background or it can be bound to an interface.
    - **Broadcast Receivers:** receives and registers events that happen outside user interaction.
    - **Content providers:** supplies data from an application to another upon request, the requests are handled by the other application's **ContentResolver**.
- **Intent:** the mechanism android system uses to identify application components to do actions such as running an activity, starting a service, or delivering a broadcast
    - **Explicit intents:** directly identifies an application component to be matched
    - **Implicit intents:** describes characteristics of the desired component, the matching is done by the "intent resolution", which can match none, single, or multiple components
- **Security in android apps**
    - Each application is assigned a User ID (UID), upon boot or installation, or it can be predefined.
    - UIDs can be shared by several apps
    - Each used UID has a corresponding directory, containing files used by applications holding that UID
        - These applications can only access their own files in their "App sandbox"
    - **Application permissions:** specific well-defined abilities that can be granted to an application at install time.



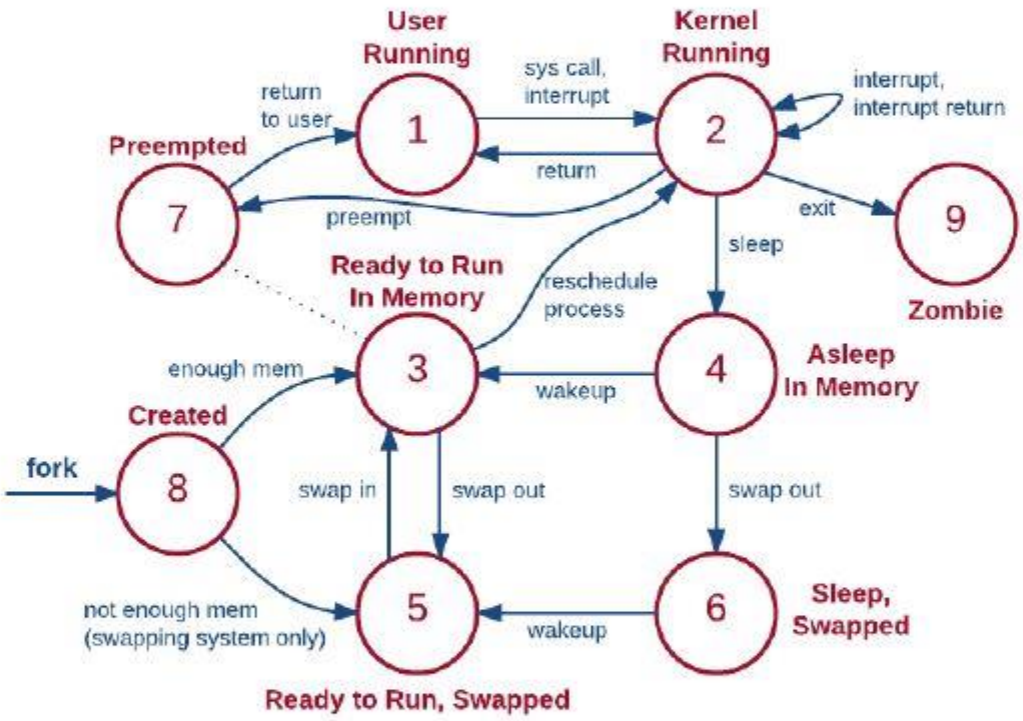
UID	Purpose
0	Root
1000	Core system (system_server process)
1001	Telephony service
1013	Low=level media processes
2000	Command line shell access
10000 – 19999	Dynamically assigned application UIDs
100000	Start of secondary users

Useful diagrams:

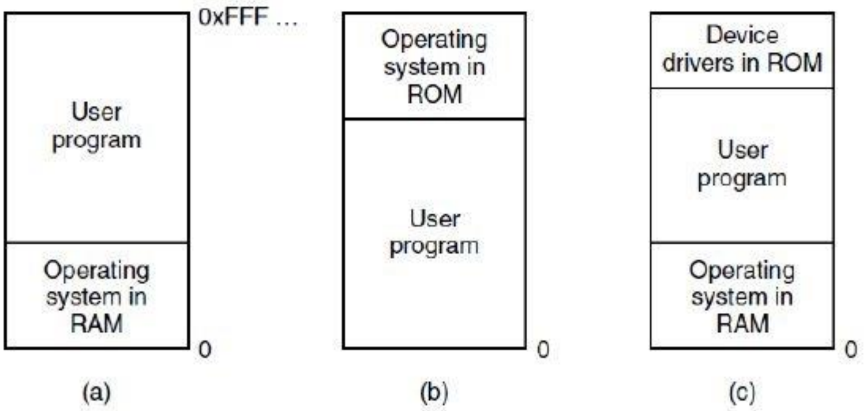
1. **Process state transition diagram (simplified)**


Running
Blocked
Ready
1
2
3
4
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

2. **Process state transition diagram (complete)**



3. **Three possible ways of organizing RAM between OS and a user program.**

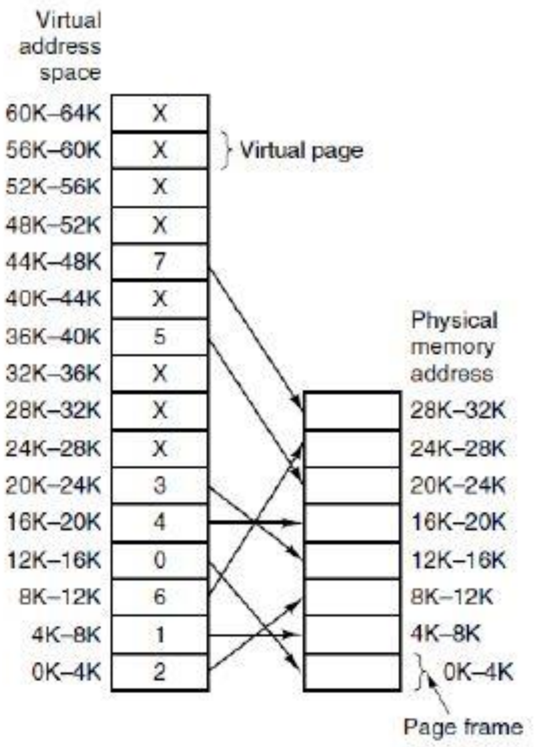


(a) Mainframes and minicomputers

(b) Handheld and embedded systems.

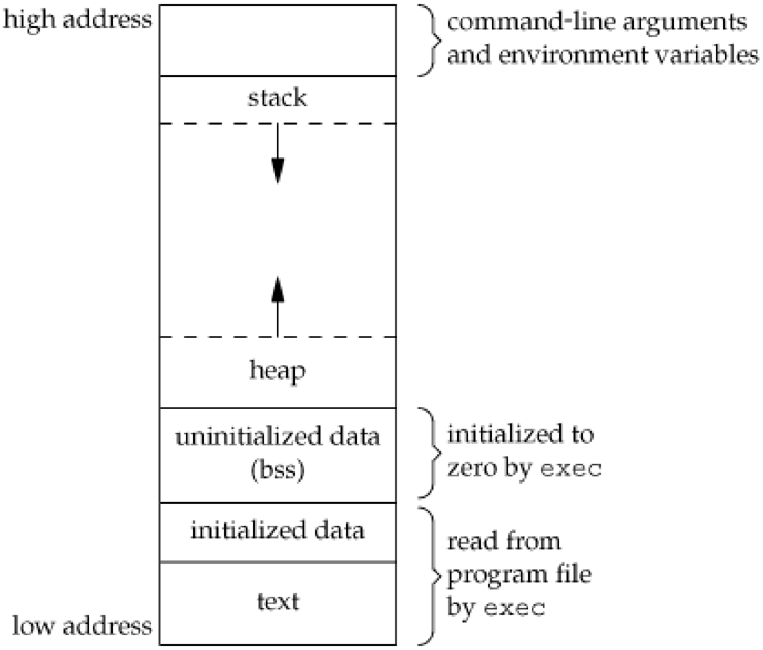
(c) Early PCs with BIOS in ROM and MS-DOS in RAM.

4. The relation between virtual addresses and physical memory addresses

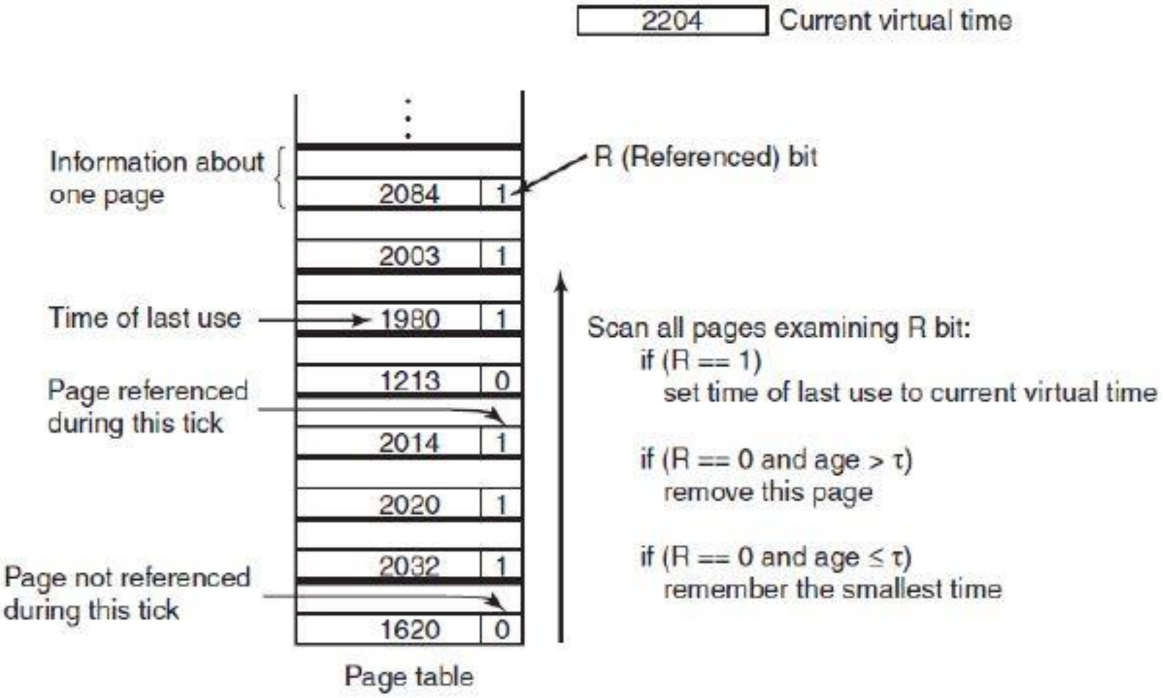


5. Memory layout of a C program

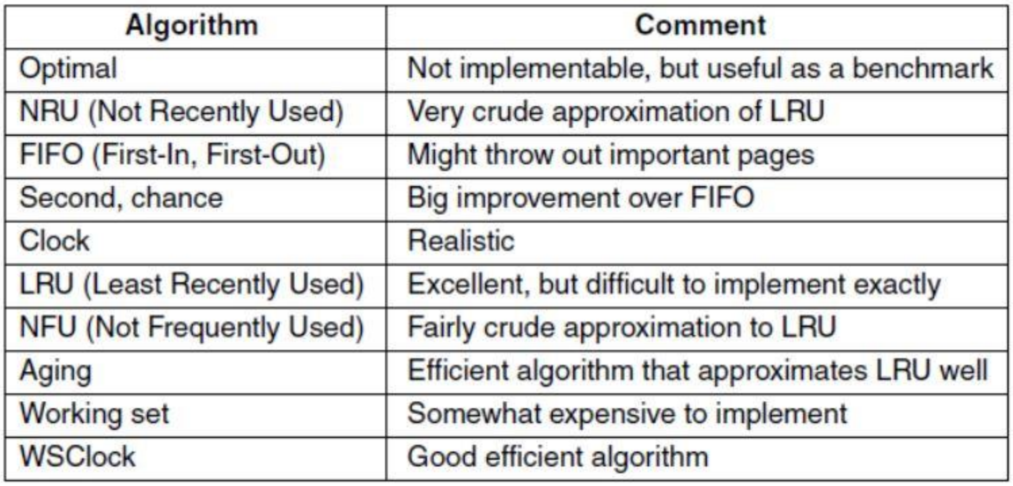
- **Text or Code Segment:** Contains machine code of the compiled program. The “text segment” of an executable object file is often a read-only segment that prevents a program from being accidentally modified.
- **Initialized Data or Data Segment:** Stores all global, static, constant, and external variables (declared with extern keyword) that are initialized beforehand.
- **Uninitialized Data or $.bss$ Segment:** Stores all uninitialized global, static, and external variables (declared with extern keyword)
- **Stack Segment**: Stores all local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over.
- **Heap Segment:** Part of RAM where dynamically allocated variables are stored. In C language dynamic memory allocation is done by using $malloc$ and $calloc$ functions


high address
low address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized data
text
initialized to zero by exec
read from program file by exec

6. Working set page replacement algorithm

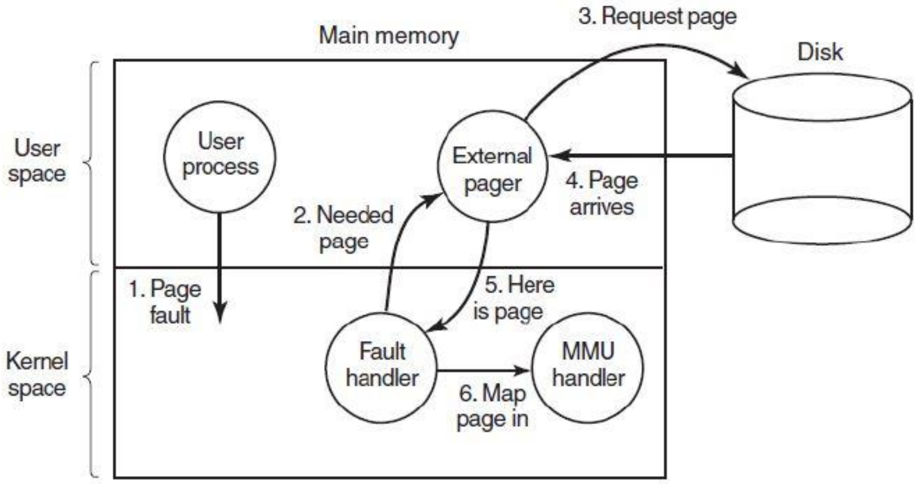

2204
Current virtual time
Information about one page
R (Referenced) bit
2084
1
2003
1
Time of last use
1980
1
1213
0
Page referenced during this tick
2014
1
2020
1
2032
1
Page not referenced during this tick
1620
0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

7. Page replacement algorithms

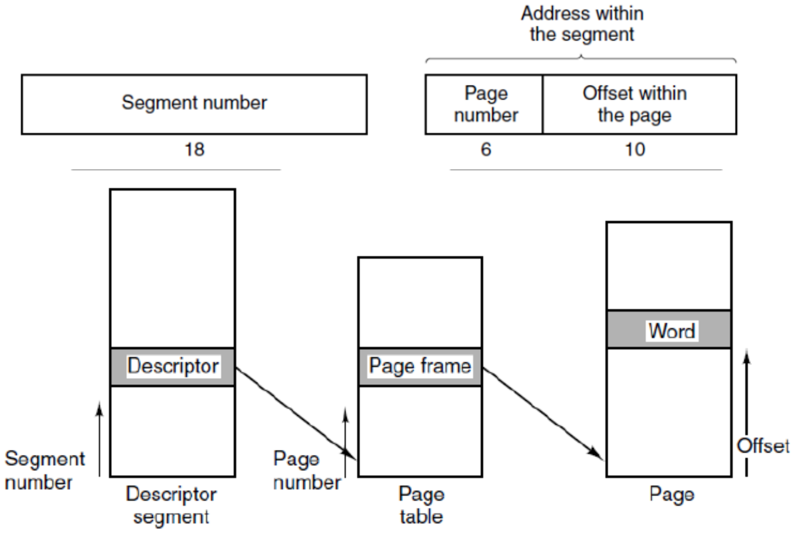


Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second, chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

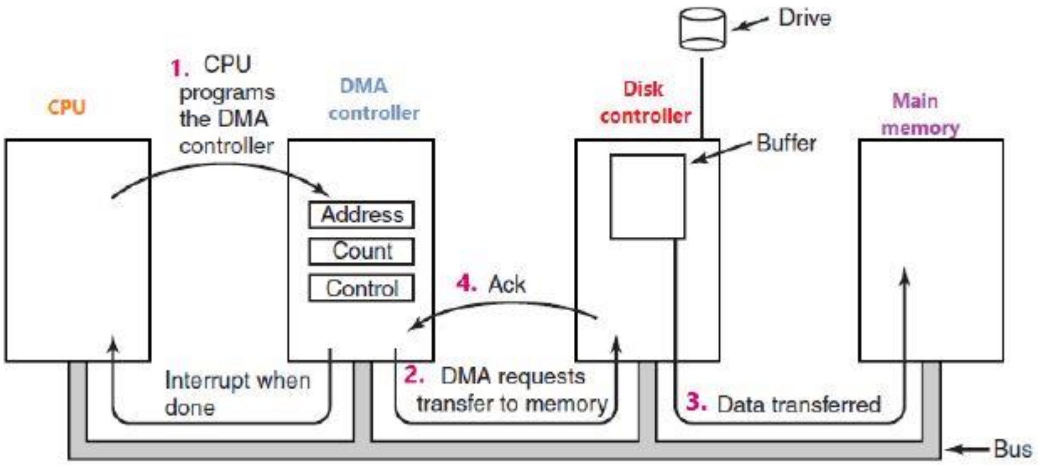
8. **Page fault handling with an external pager.**



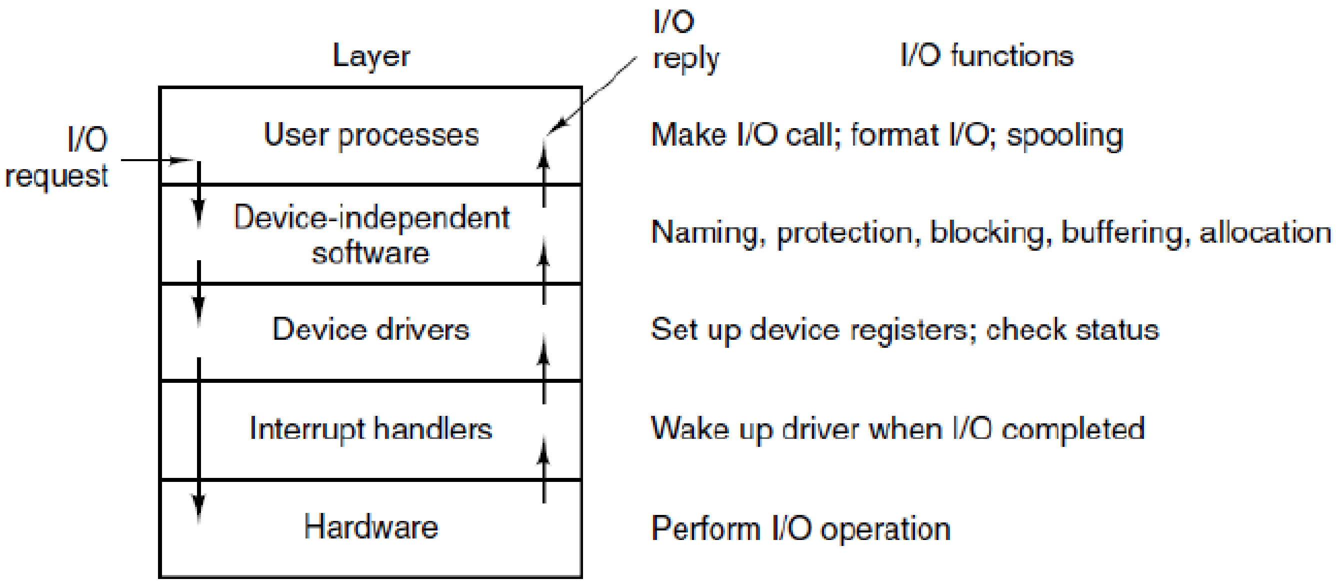
9. **Addresses and memory referencing in MULTICS**



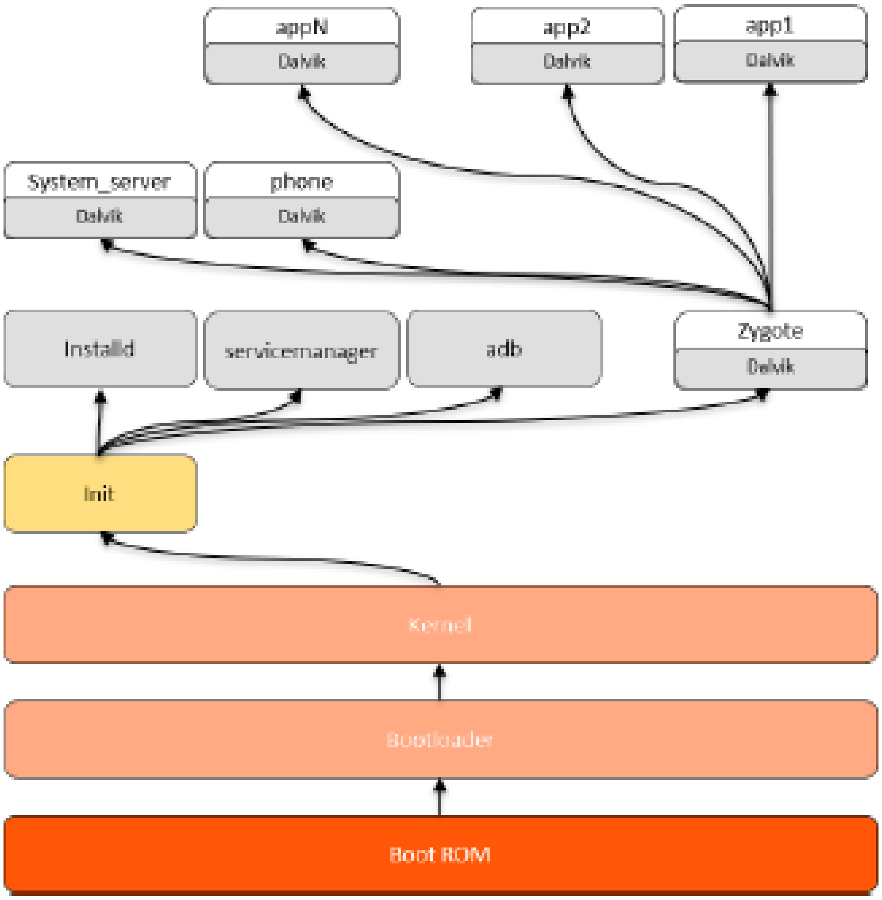
10. **Disk to memory transfer using DMA**



11. I/O software layers


Layer
I/O reply
I/O functions
I/O request
User processes
Make I/O call; format I/O; spooling
Device-independent software
Naming, protection, blocking, buffering, allocation
Device drivers
Set up device registers; check status
Interrupt handlers
Wake up driver when I/O completed
Hardware
Perform I/O operation

12. Booting an Android


appN
Dalvik
app2
Dalvik
app1
Dalvik
System_server
Dalvik
phone
Dalvik
Installd
servicemanager
adb
Zygote
Dalvik
Init
Kernel
Bootloader
Boot ROM