ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ОПЕРАЦІЙНИХ СИСТЕМ

Питання 1.5

Механізми та політики ОС

- Один з важливих принципів – відокремлення політик (policy) від механізмів.
  - Механізми визначають, ***як*** виконувати дії; політики визначають, ***що*** потрібно зробити.
  - Наприклад, таймер – механізм забезпечення захисту ЦП, проте тривалість роботи таймеру для конкретного користувача – вибір політики.
- Відокремлення важливе для гнучкості.
  - Політики можуть змінюватись з часом та залежно від місця.
  - У найгіршому випадку кожна зміна політики вимагає зміни механізму в її основі.
  - Краще мати загальний механізм, достатньо гнучкий для роботи з набором політик. Тоді зміна політики вимагатиме лише переозначення деяких параметрів системи.
- Мікроядерні ОС використовують цей принцип на повну, реалізуючи базовий набір примітивних будівельних блоків.
  - Ці блоки майже незалежні від політик, дозволяють додавати складніші механізми та політики до user-created kernel modules або user programs themselves.
  - З іншого боку, Windows, де механізми та політики closely encoded в систему, щоб робота з усіма пристроями була в єдиному стилі, додатки мали схожий інтерфейс (описаний в ядрі та системних бібліотеках).
  - Apple використала аналогічну стратегію в macOS та iOS.

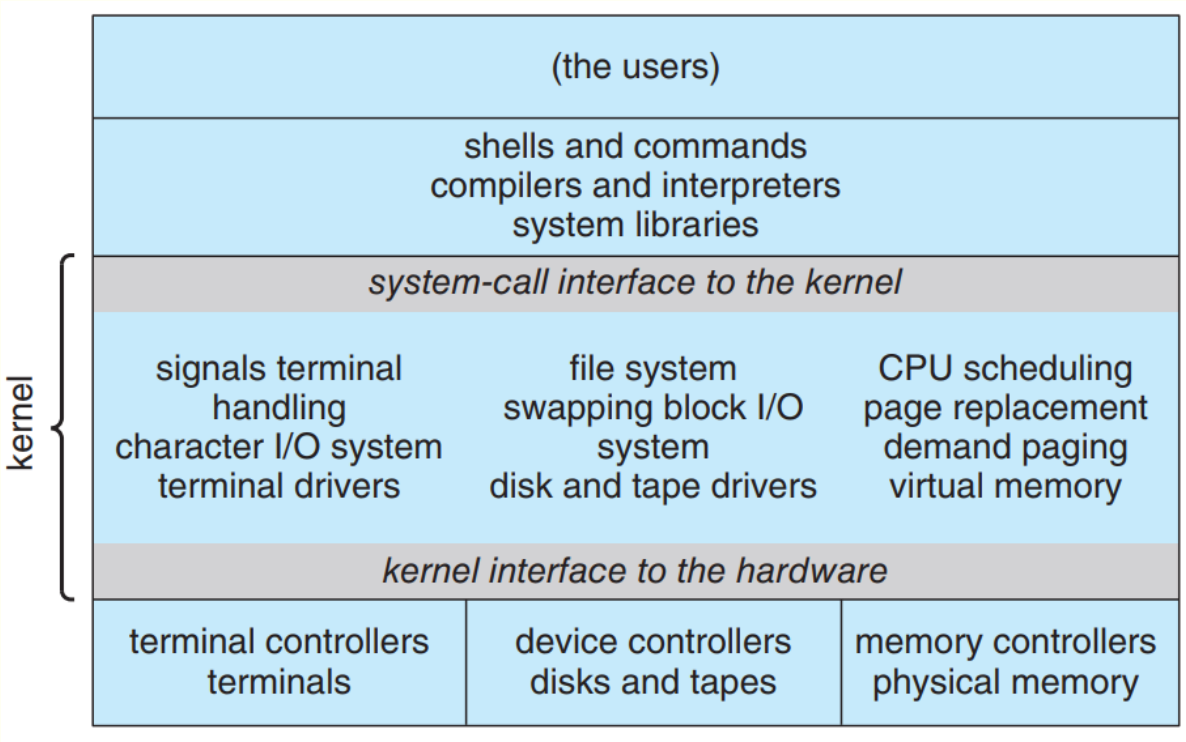
Механізми та політики ОС

- Аналогічно порівняємо комерційні ОС та ОС з відкритим кодом.
  - На відміну від Windows, ОС Linux має відкритий код. «Стандартне» ядро Linux має специфічні алгоритми планування роботи ЦП, які є механізмом, що підтримує певну політику.
  - Проте будь-хто може змінити чи замінити планувальник, щоб підтримувати іншу політику.
- Передбачені в політиці рішення важливі для виділення ресурсів.
  - Політика повинна вирішувати, чи необхідно виділяти ресурси.
  - Whenever the question is ***how*** rather than ***what***, it is a mechanism that must be determined.

Реалізація ОС

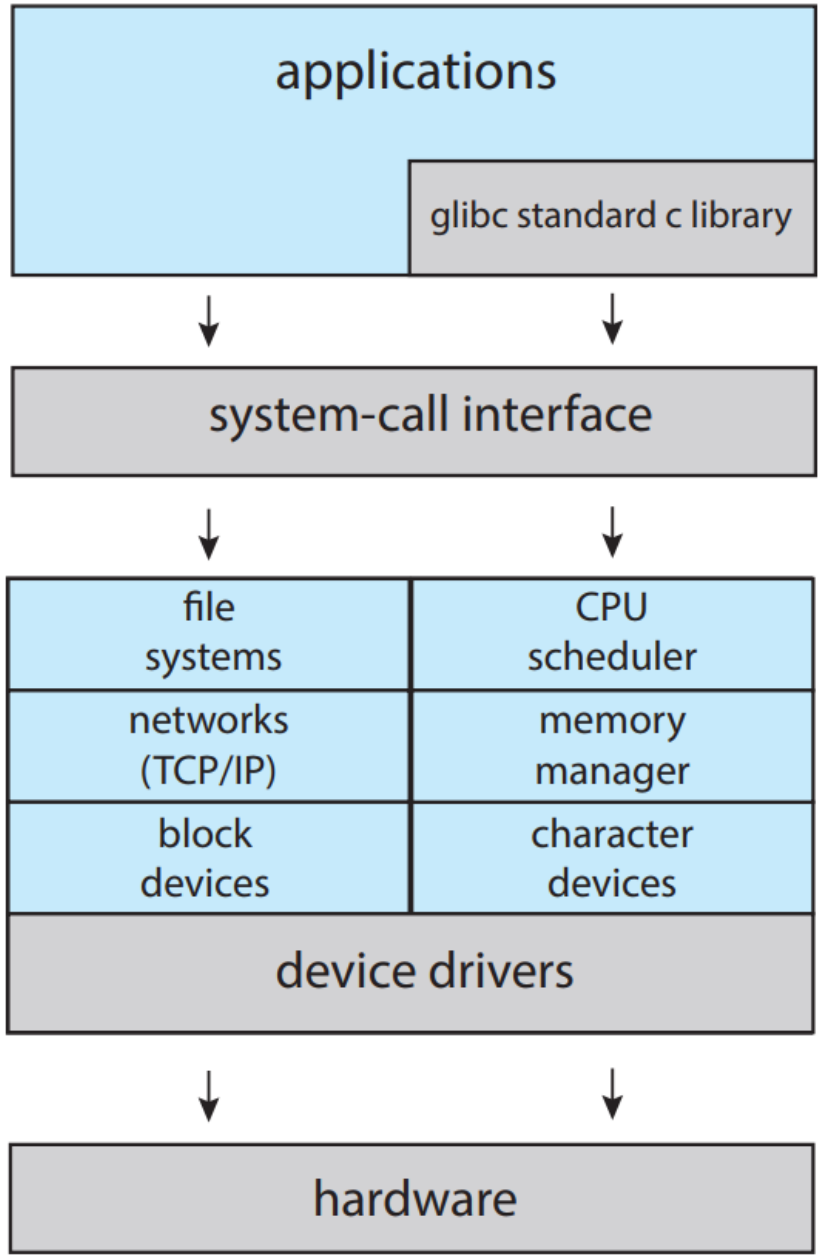
- Ранні ОС писались мовою ассемблера.
  - Нині більшість з них пишуться високорівневими мовами на зразок C або C++ з невеликими вкрапленнями ассемблера. In fact, more than one higherlevel language is often used.
  - Найнижчі рівні ядра можуть писатись мовами C та ассемблера.
  - Вискорівневі підпрограми можуть писатись мовами C / C++, а системні бібліотеки – на C++ або ще більш високорівневих мовах.
  - Ядро ОС Android написано переважно мовою C + трохи ассемблера. Більшість системних бібліотек пишуться мовами C / C++, а фреймворк для додатків – переважно мовами Java / Kotlin.
- Код високорівневих мов пишеться швидше, компактніше, простіше для розуміння та налагодження.
  - Розвиток компіляторів покращує згенерований код усієї ОС простою перекомпіляцією.
  - ОС набагато простіше портувати на інше «залізо», якщо вона написана високорівневою мовою.
  - Особливо важливе для ОС, які працюють на пристроях з різними архітектурами (вбудовані пристрої, x86-системи, ARM-чіпи на смартфонах та планшетах).
- Можливий недолік – зменшена швидкодія та збільшення займаного простору.
  - Хоч ОС великі, тільки невелика частина коду критична для швидкодії: interrupt handlers, I/O manager, memory manager, CPU scheduler та ін.
  - При коректній роботі ОС вузькі місця можна визначити та виконати їх рефакторинг.

Структура ОС



Традиційна структура UNIX

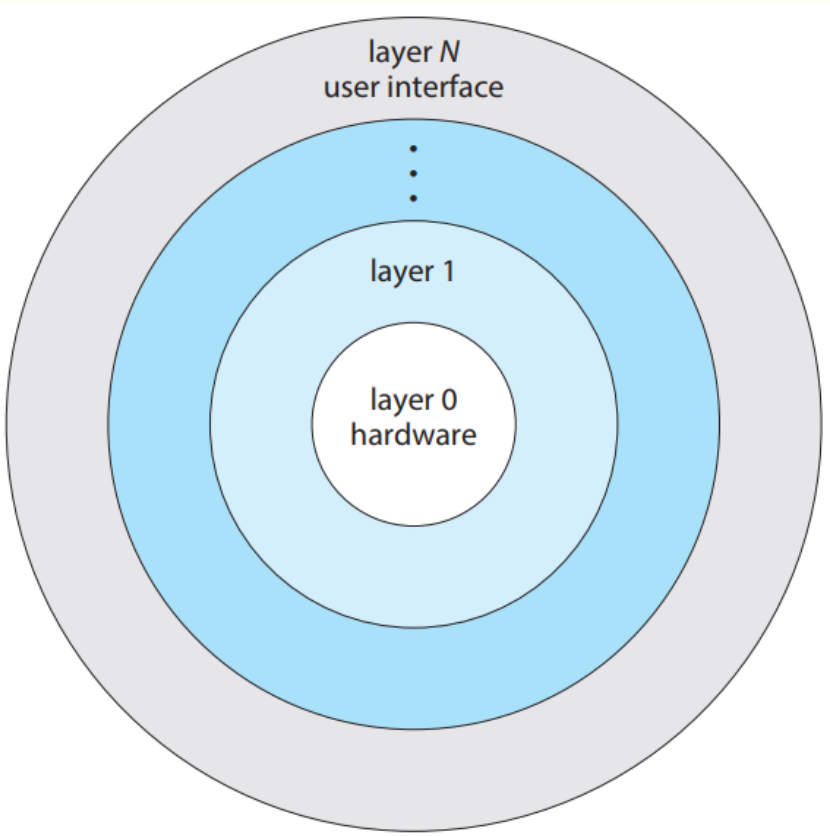
- Поширений підхід – розбивати задачу на невеликі компоненти (модулі), а не мати єдину систему.
  - Кожен з цих модулів should be a well-defined portion of the system, with carefully defined interfaces and functions.
  - You may use a similar approach when you structure your programs: rather than placing all of your code in the main() function, you instead separate logic into a number of functions, clearly articulate parameters and return values, and then call those functions from main()
- Найпростіша структура організації ОС – відсутність структури – ***монолітна структура***.
  - Розміщує всю функціональність ядра в єдиний, статичний бінарний файл, який запускається в єдиному адресному просторі.
  - Оригінальна ОС UNIX складається з 2 відокремлюваних частин: ядра та системних програм.
  - Далі ядро розбивається на набір інтерфейсів та драйверів пристроїв, які додаються та розширюються роками.



Структура ОС Linux

- Додатки зазвичай використовують стандартну бібліотеку glibc мови С для взаємодії з інтерфейсом системних викликів до ядра.
  - Ядро Linux монолітне в тому, що воно повністю працює в режимі ядра в єдиному адресному просторі, проте воно має модульний дизайн, який дозволяє вносити в ядро зміни during run time.
  - Незважаючи на простоту ідеї монолітних ядер, їх важко реалізовувати та розширяти.
- Монолітні ядра мають переваги в швидкодії, проте there is very little overhead in the system-call interface, and communication within the kernel is fast.
  - Therefore, despite the drawbacks of monolithic kernels, their speed and efficiency explains why we still see evidence of this structure in the UNIX, Linux, and Windows operating systems.

Шаруватий (layered) підхід

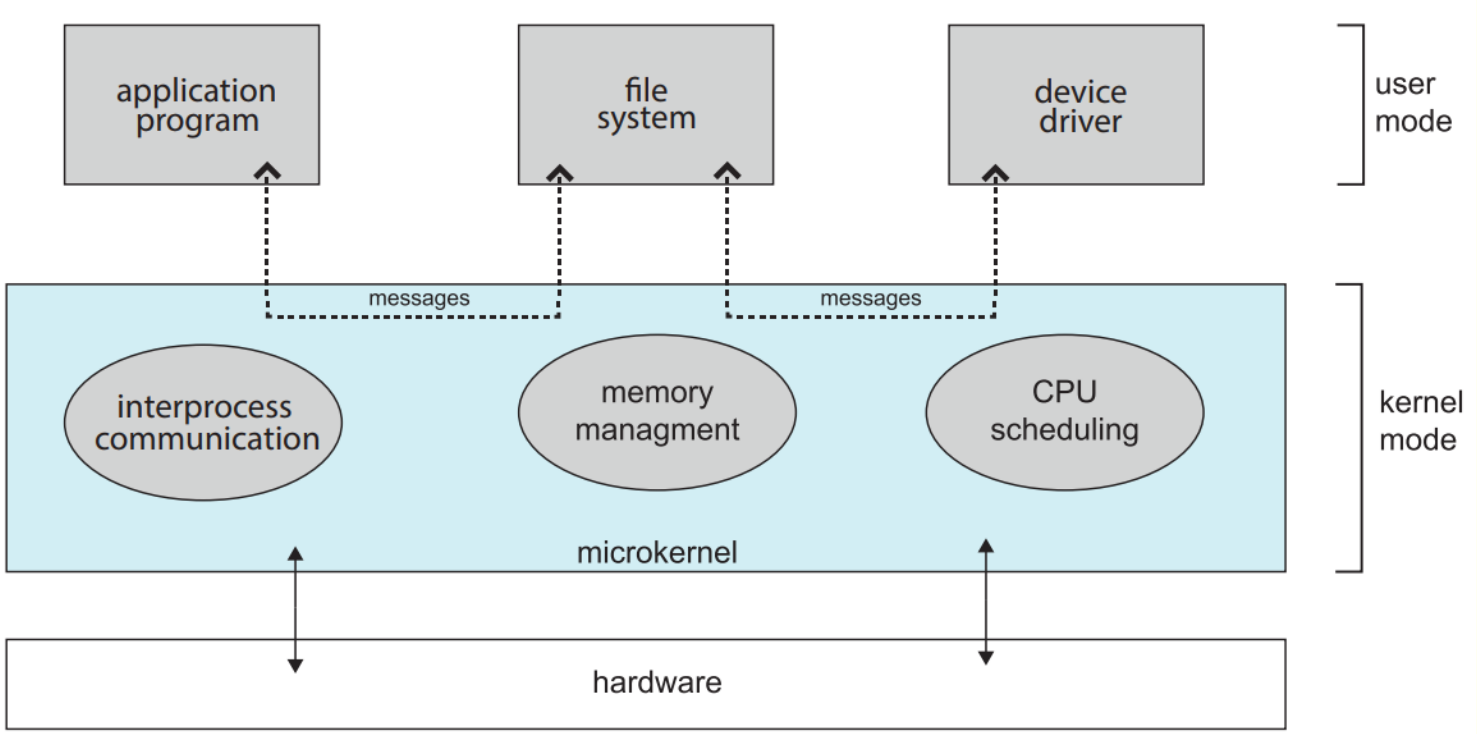


- Монолітний підхід часто відомий як **tightly coupled** system, оскільки зміни в одній частині системи різноманітні ефекти на решту ядра.
- Альтернативно проектують **loosely coupled** систему.
  - Вона ділиться на окремі, менші компоненти з конкретною та обмеженою функціональністю.
  - Перевага: зміни одного компоненту впливають лише на нього самого, allowing system implementers more freedom in creating and changing the inner workings of the system.
- Система може бути модульною різними способами.
  - ***Шаруватий (layered) підхід*** – ОС розбивається на кілька рівнів (прошарків).
  - Нижній прошарок (layer 0) – «залізо»; найвищий (layer *N*) – інтерфейс користувача.
  - Прошарок ОС – реалізація of an abstract object made up of data and the operations that can manipulate those data.
  - Типовий прошарок ОС складається зі структур даних та набору функцій, які можна викликати на прошарках вищого рівня. А цей прошарок може звертатись до операцій з нижчих прошарків.

Шаруватий (layered) підхід

- Основна перевага – простота конструювання та налагодження.
  - The layers are selected so that each uses functions (operations) and services of only lower-level layers.
  - Підхід спрощує налагодження та верифікацію системи.
  - The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions.
  - Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on.
  - If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Так проектування та реалізація системи спрощуються.
- Кожен прошарок реалізується лише з операціями, які забезпечуються прошарками нижчого рівня.
  - Прошарку не потрібно знати, як ці операції реалізовані, а лише те, що ці операції роблять.
  - Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.
- Шаруваті системи успішно використовуються в комп'ютерних мережах (таких як TCP/IP) та веб-додатках.
  - Проте відносно мало ОС використовують чисто шаруватий підхід.
  - One reason involves the challenges of appropriately defining the functionality of each layer.
  - In addition, the overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service.
  - ***Some*** layering is common in contemporary operating systems, however.
  - Generally, these systems have fewer layers with more functionality, providing most of the advantages of modularized code while avoiding the problems of layer definition and interaction.

Мікроядерна структура (Microkernels)



- Розширяючись, UNIX-ядро стало великим та складним для керування.
  - В середині 1980-х рр. дослідники з Carnegie Mellon University розробили ОС Mach, яка модуляризувала ядро за допомогою мікроядерного підходу.
  - Цей метод структурує ОС, видаляючи всі неважливі компоненти з ядра та реалізуючи їх як користувацькі програми, які розміщуються в окремих адресних просторах.
- Результат – менше ядро.
  - Консенсусу щодо того, які служби залишити в ядрі, а які винести – майже немає.
  - Зазвичай мікроядра забезпечують мінімальне керування процесами та пам'яттю, in addition to a communication facility.

Мікроядерна структура (Microkernels)

- Основна функція мікроядра – забезпечити взаємодію між клієнтською програмою and the various services that are also running in user space.
  - Взаємодія забезпечується шляхом передачі повідомлень.
  - Наприклад, якщо клієнтська програма потребує доступу до файлу, їй необхідно взаємодіяти з файловим сервером (опосередковано через обмін повідомленнями з мікроядром).
- One benefit of the microkernel approach is that it makes extending the operating system easier.
  - All new services are added to user space and consequently do not require modification of the kernel.
  - When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.
  - The resulting operating system is easier to port from one hardware design to another.
- The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes.
  - If a service fails, the rest of the operating system remains untouched.
  - Perhaps the best-known illustration of a microkernel operating system is ***Darwin***, the kernel component of the macOS and iOS operating systems.
  - Darwin, in fact, consists of two kernels, one of which is the Mach microkernel.

Another example is QNX

- ОС реального часу для вбудованих систем.
  - The QNX Neutrino microkernel provides services for message passing and process scheduling.
  - It also handles low-level network communication and hardware interrupts.
  - All other services in QNX are provided by standard processes that run outside the kernel in user mode.
  - Unfortunately, the performance of microkernels can suffer due to increased system-function overhead.
  - When two user-level services must communicate, messages must be copied between the services, which reside in separate address spaces.
- In addition, the operating system may have to switch from one process to the next to exchange the messages.
  - The overhead involved in copying messages and switching between processes has been the largest impediment to the growth of microkernel-based operating systems.
  - Consider the history of Windows NT: The first release had a layered microkernel organization.
  - This version’s performance was low compared with that of Windows 95. Windows NT 4.0 partially corrected the performance problem by moving layers from user space to kernel space and integrating them more closely.
  - By the time Windows XP was designed, Windows architecture had become more monolithic than microkernel.

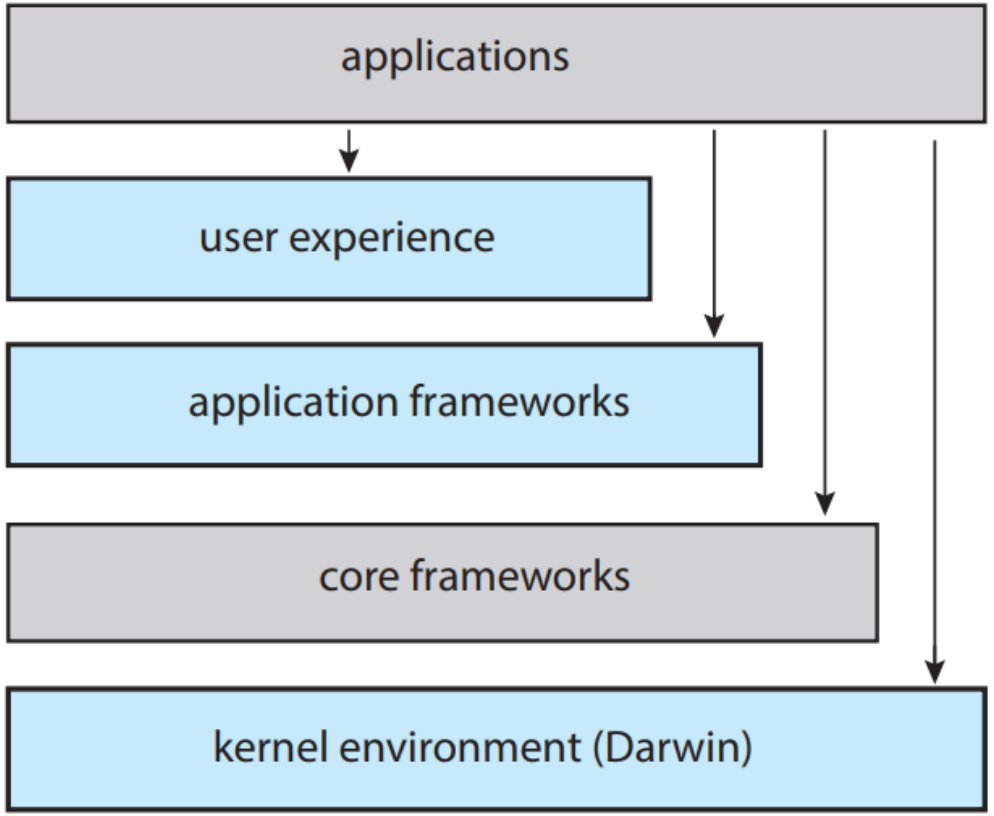
Модулі

- Одна з найкращих сучасних методологій проектування ОС включає ***завантажувані модулі ядра (loadable kernel modules, LKM)***.
  - Ядро – набір базових компонентів and can link in additional services via modules, either at boot time or during run time.
  - This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows.
  - Ідея проектування: для ядра постачати базові служби, а решту – динамічно імплементувати під час роботи ядра.
  - Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.
  - Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.
  - The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module.
  - The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.
- Linux використовує LKM-и, в основному для підтримки драйверів пристроїв та файлових систем.
  - Вони можуть «вставлятись» в ядро при запуску чи роботі системи (наприклад, USB-пристрій підключається до працюючої машини). Якщо Linux-ядро не матиме потрібного драйверу, його можна динамічно підвантажити.
  - LKMs can be removed from the kernel during run time as well. For Linux, LKMs allow a dynamic and modular kernel, while maintaining the performance benefits of a monolithic system.

Гібридні системи

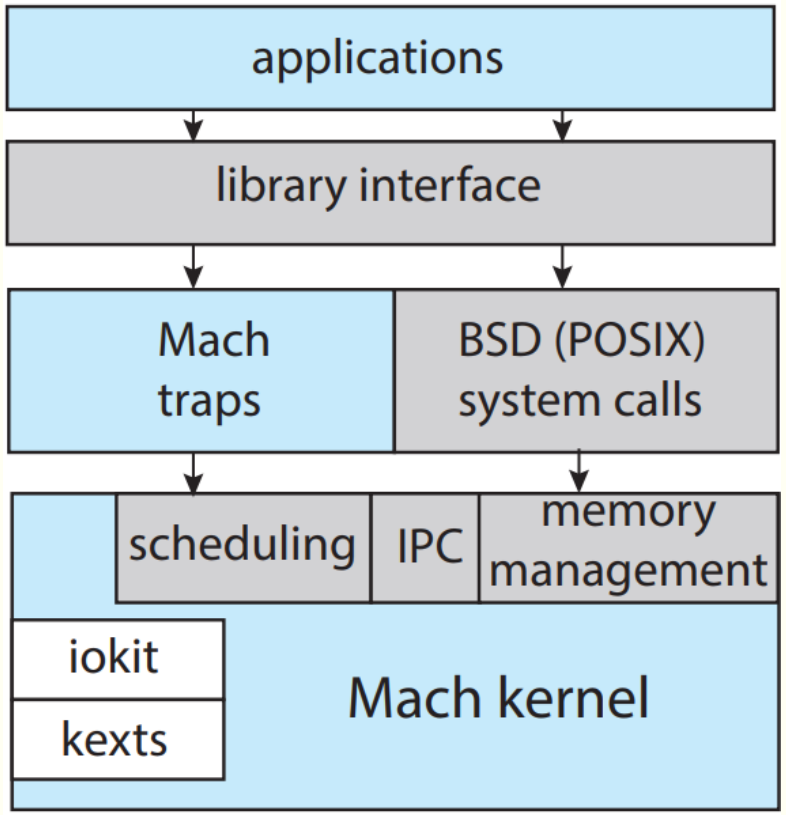
- In practice, very few operating systems adopt a single, strictly defined structure.
  - Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues.
  - For example, Linux is monolithic, because having the operating system in a single address space provides very efficient performance.
  - However, it also modular, so that new functionality can be dynamically added to the kernel.
  - Windows is largely monolithic as well (again primarily for performance reasons), but it retains some behavior typical of microkernel systems, including providing support for separate subsystems (known as operating-system ***personalities***) that run as user-mode processes.
  - Windows systems also provide support for dynamically loadable kernel modules.

macOS та iOS



- Архітектурно macOS та iOS мають багато спільного.
  - **User experience layer**. Визначає програмний інтерфейс, який дозволяє взаємодіяти з обчислювальними пристроями. macOS використовує інтерфейс ***Aqua***, спроектований для миші чи трекпаду, iOS використовує інтерфейс ***Springboard*** для сенсорних пристроїв.
  - **Application frameworks layer**. Включає фреймворки ***Cocoa*** та ***Cocoa Touch***, які постачають API для мов програмування Objective-C та Swift. Основна відмінність між Cocoa та Cocoa Touch – перший для розробки macOS-додатків, а останній – для iOS.
  - **Core frameworks**. Прошарок визначає фреймворки з підтримкою графіки та мультимедіа, включаючи Quicktime та OpenGL.
  - **Kernel environment**. Також відоме, як **Darwin**, включає мікроядро Mach та ядро BSD UNIX.

macOS та iOS



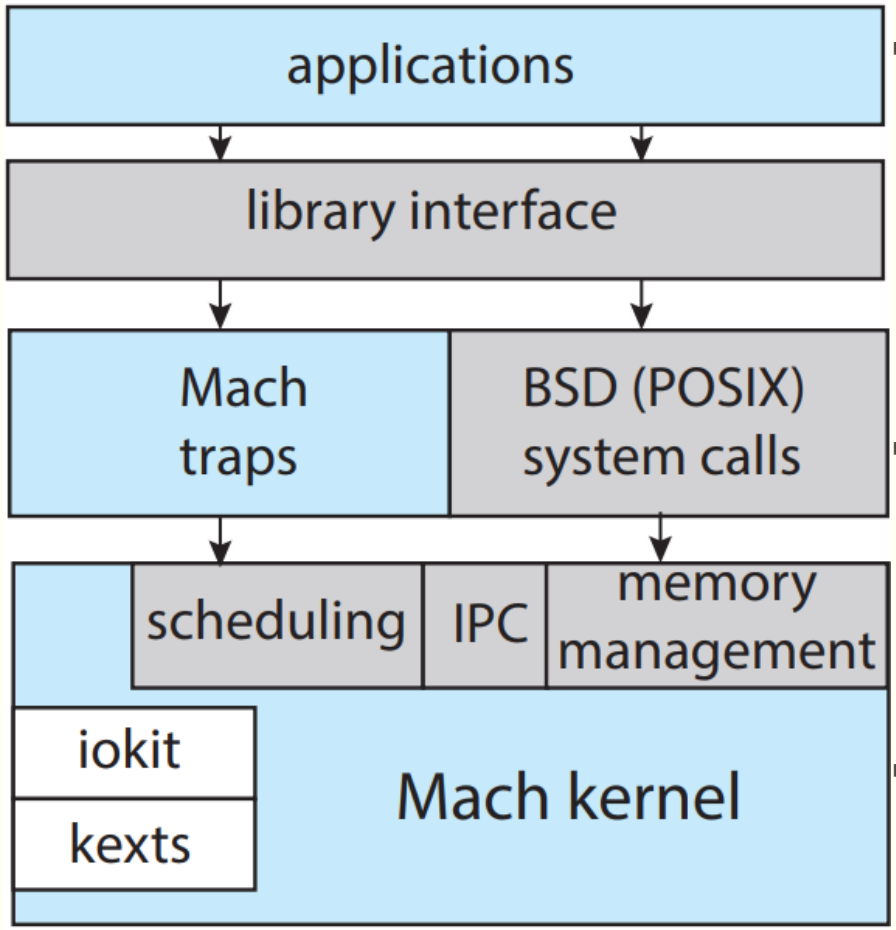
Деякі значні відмінності між macOS та iOS:

- macOS призначена для архітектур від Intel.
- iOS спроектована для мобільних пристроїв та компілюється для ARM-архітектур.
- Similarly, the iOS kernel has been modified somewhat to address specific features and needs of mobile systems, such as power management and aggressive memory management. Additionally, iOS has more stringent security settings than macOS.
- The iOS operating system is generally much more restricted to developers than macOS and may even be closed to developers.
- For example, iOS restricts access to POSIX and BSD APIs on iOS, whereas they are openly available to developers on macOS.

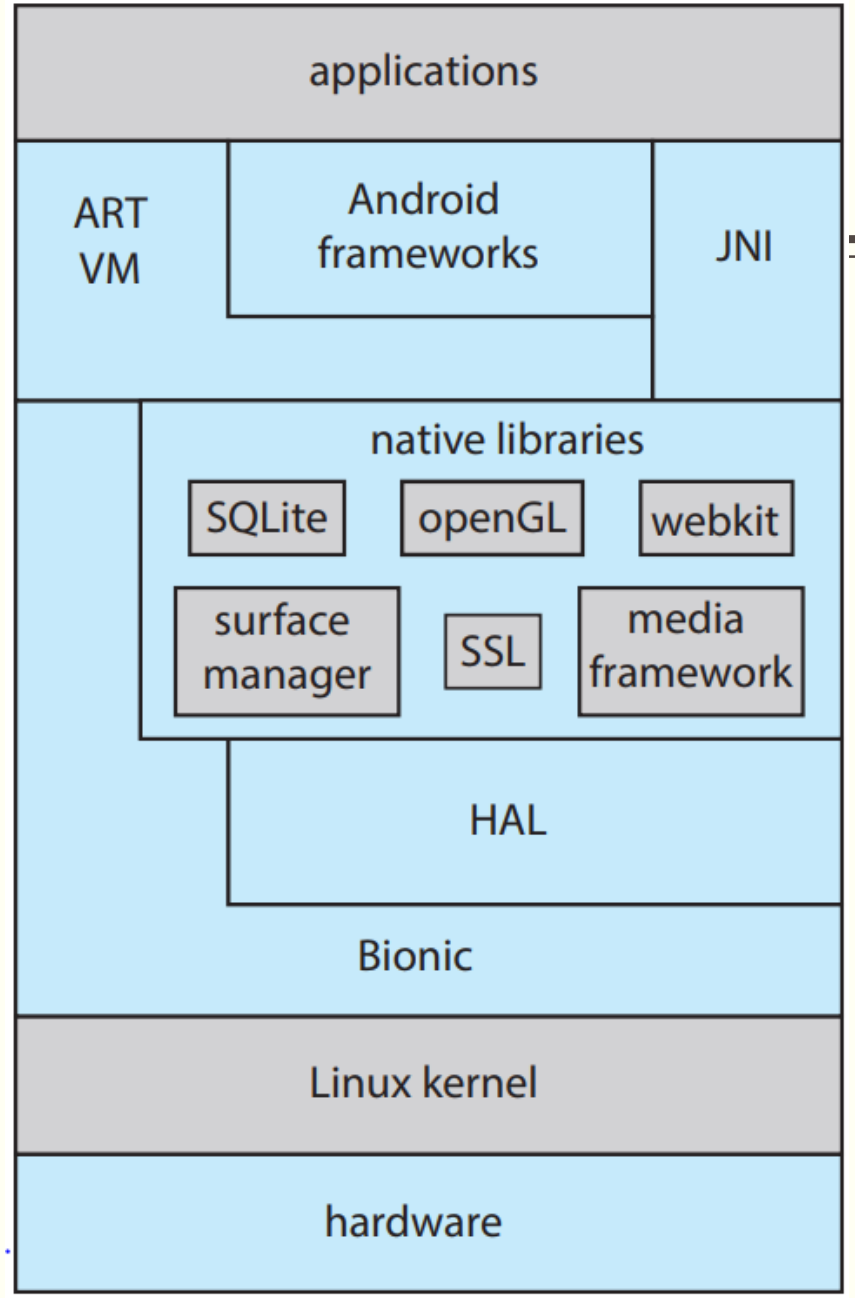
Darwin – шарувата система, яка в основному складається з мікроядра Mach та ядра BSD UNIX.

- Darwin постачає ***2*** інтерфейси системних викликів: системні виклики Mach (**traps**) та системні виклики BSD (постачають POSIX-функціональність).
- The interface to these system calls is a rich set of libraries that includes not only the standard C library but also libraries that provide networking, security, and programming language support (to name just a few).

macOS та iOS

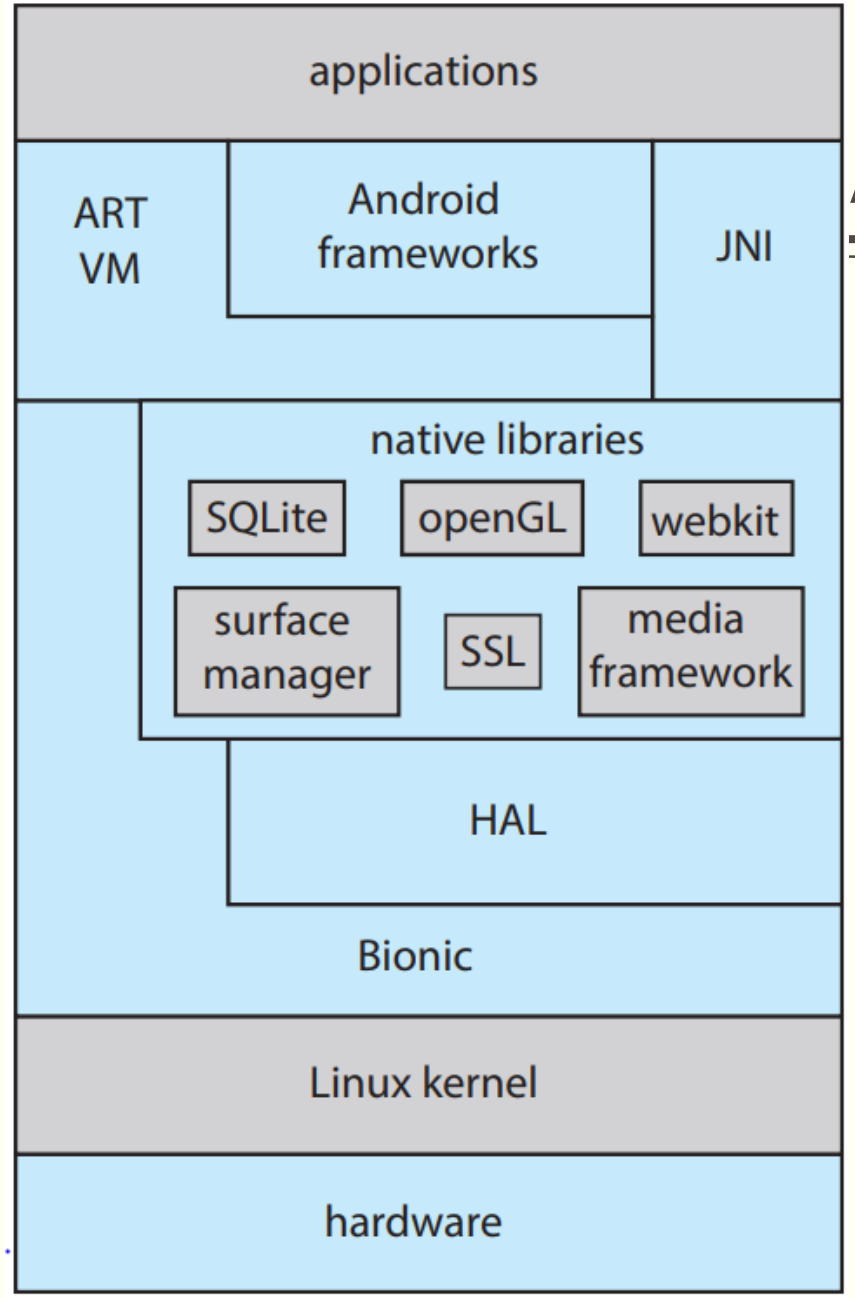


- Beneath the system-call interface, Mach provides fundamental operatingsystem services, including memory management, CPU scheduling, and interprocess communication (IPC) facilities such as message passing and remote procedure calls (RPCs).
  - Much of the functionality provided by Mach is available through **kernel abstractions**, which include tasks (a Mach process), threads, memory objects, and ports (used for IPC).
  - As an example, an application may create a new process using the BSD POSIX fork() system call. Mach will, in turn, use a task kernel abstraction to represent the process in the kernel.
  - In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which macOS refers to as **kernel extensions**, or **kexts**).
- In Section 2.8.3, we described how the overhead of message passing between different services running in user space compromises the performance of microkernels.
  - To address such performance problems, Darwin combines Mach, BSD, the I/O kit, and any kernel extensions into a single address space.
  - Thus, Mach is not a pure microkernel in the sense that various subsystems run in user space. Message passing within Mach still does occur, but no copying is necessary, as the services have access to the same address space.
- Apple has released the Darwin operating system as open source. As a result, various projects have added extra functionality to Darwin, such as the X-11 windowing system and support for additional file systems.
  - Unlike Darwin, however, the Cocoa interface, as well as other proprietary Apple frameworks available for developing macOS applications, are closed.



Android

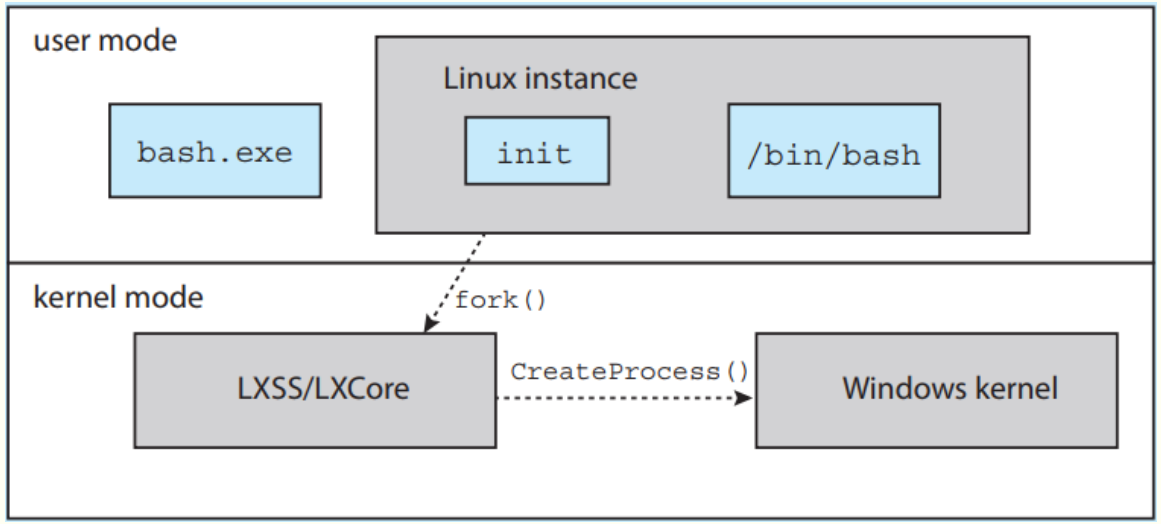
- The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers.
  - Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks supporting graphics, audio, and hardware features.
  - These features, in turn, provide a platform for developing mobile applications that run on a multitude of Android-enabled devices.
- Software designers for Android devices develop applications in the Java language, but they do not generally use the standard Java API.
  - Google has designed a separate Android API for Java development.
  - Java applications are compiled into a form that can execute on the Android RunTime ART, a virtual machine designed for Android and optimized for mobile devices with limited memory and CPU processing capabilities.
  - Java programs are first compiled to a Java bytecode .class file and then translated into an executable .dex file.
  - Whereas many Java virtual machines perform just-in-time (JIT) compilation to improve application efficiency, ART performs **ahead-of-time** (**AOT**) compilation.
  - Here, .dex files are compiled into native machine code when they are installed on a device, from which they can execute on the ART.
  - AOT compilation allows more efficient application execution as well as reduced power consumption, features that are crucial for mobile systems.



Android

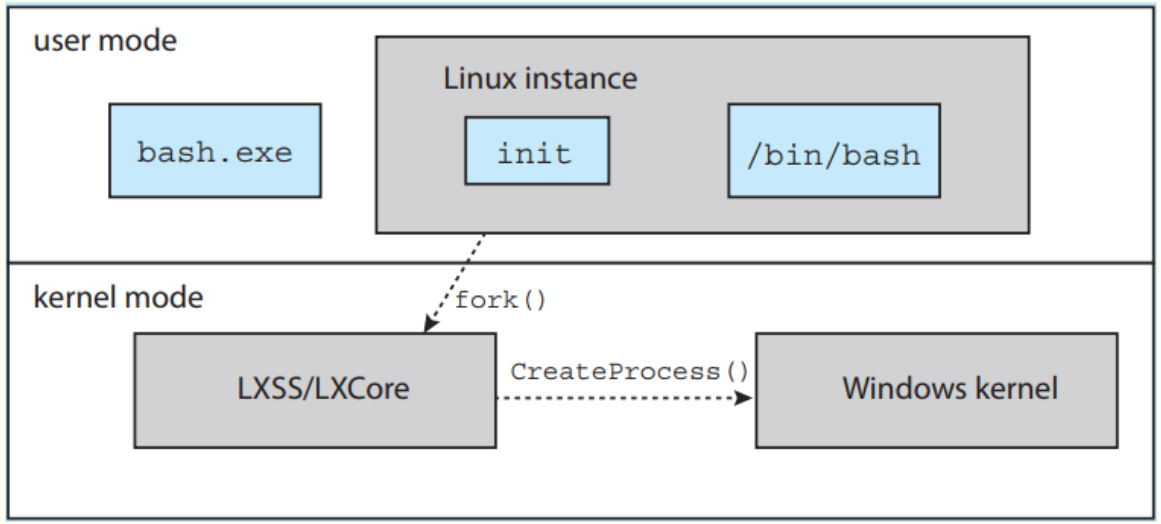
- Android developers can also write Java programs that use the Java native interface—or JNI—which allows developers to bypass the virtual machine and instead write Java programs that can access specific hardware features.
  - Programs written using JNI are generally not portable from one hardware device to another.
- The set of native libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and network support, such as secure sockets (SSLs).
- Because Android can run on an almost unlimited number of hardware devices, Google has chosen to abstract the physical hardware through the hardware abstraction layer, or HAL. By abstracting all hardware, such as the camera, GPS chip, and other sensors, the HAL provides applications with a consistent view independent of specific hardware. This feature, of course, allows developers to write programs that are portable across different hardware platforms.
- The standard C library used by Linux systems is the GNU C library (glibc). Google instead developed the **Bionic** standard C library for Android. Not only does Bionic have a smaller memory footprint than glibc, but it also has been designed for the slower CPUs that characterize mobile devices. (In addition, Bionic allows Google to bypass GPL licensing of glibc.)
- At the bottom of Android's software stack is the Linux kernel. Google has modified the Linux kernel used in Android in a variety of areas to support the special needs of mobile systems, such as power management. It has also made changes in memory management and allocation and has added a new form of IPC known as ***Binder***

WINDOWS SUBSYSTEM FOR LINUX



- Windows uses a hybrid architecture that provides subsystems to emulate different operating-system environments.
  - These user-mode subsystems communicate with the Windows kernel to provide actual services. Windows 10 adds a Windows subsystem for Linux (**WSL**), which allows native Linux applications (specified as ELF binaries) to run on Windows 10.
  - The typical operation is for a user to start the Windows application bash.exe, which presents the user with a bash shell running Linux.
  - Internally, the WSL creates a **Linux instance** consisting of the init process, which in turn creates the bash shell running the native Linux application /bin/bash. Each of these processes runs in a Windows **Pico** process.
  - This special process loads the native Linux binary into the process's own address space, thus providing an environment in which a Linux application can execute.

WINDOWS SUBSYSTEM FOR LINUX



- Pico processes communicate with the kernel services LXCore and LXSS to translate Linux system calls, if possible using native Windows system calls.
  - When the Linux application makes a system call that has no Windowsequivalent, the LXSS service must provide the equivalent functionality.
  - When there is a one-to-one relationship between the Linux and Windows system calls, LXSS forwards the Linux system call directly to the equivalent call in the Windows kernel.
  - In some situations, Linux and Windows have system calls that are similar but not identical.
  - When this occurs, LXSS will provide some of the functionality and will invoke the similar Windows system call to provide the remainder of the functionality. The Linux fork() provides an illustration of this: The Windows CreateProcess() system call is similar to fork() but does not provide exactly the same functionality.
  - When fork() is invoked in WSL, the LXSS service does some of the initial work of fork() and then calls CreateProcess() to do the remainder of the work.

Проектування та завантаження операційних систем

- If you are generating (or building) an operating system from scratch, you must follow these steps:
  - **1.** Write the operating system source code (or obtain previously written source code).
  - **2.** Configure the operating system for the system on which it will run.
  - **3.** Compile the operating system.
  - **4.** Install the operating system.
  - **5.** Boot the computer and its new operating system.
- Configuring the system involves specifying which features will be included, and this varies by operating system.
  - Typically, parameters describing how the system is configured is stored in a configuration file of some type, and once this file is created, it can be used in several ways.
- At one extreme, a system administrator can use it to modify a copy of the operating-system source code. Then the operating system is completely compiled (known as a **system build**).
  - Data declarations, initializations, and constants, along with compilation, produce an output-object version of the operating system that is tailored to the system described in the configuration file.

- At a slightly less tailored level, the system description can lead to the selection of precompiled object modules from an existing library. These modules are linked together to form the generated operating system. This process allows the library to contain the device drivers for all supported I/O devices, but only those needed are selected and linked into the operating system. Because the system is not recompiled, system generation is faster, but the resulting system may be overly general and may not support different hardware configurations. At the other extreme, it is possible to construct a system that is completely modular. Here, selection occurs at execution time rather than at compile or link time. System generation involves simply setting the parameters that describe the system configuration.

- The major differences among these approaches are the size and generality of the generated system and the ease of modifying it as the hardware configuration changes. For embedded systems, it is not uncommon to adopt the first approach and create an operating system for a specific, static hardware configuration. However, most modern operating systems that support desktop and laptop computers as well as mobile devices have adopted the second approach. That is, the operating system is still generated for a specific hardware configuration, but the use of techniques such as loadable kernel modules provides modular support for dynamic changes to the system.

Збірка ОС Linux з нуля

- **1.** Download the Linux source code from http://www.kernel.org.
  2. Configure the kernel using the “make menuconfig” command. This step generates the .config configuration file.
  3. Compile the main kernel using the “make” command. The make command compiles the kernel based on the configuration parameters identified in the .config file, producing the file vmlinuz, which is the kernel image.
  4. Compile the kernel modules using the “make modules” command. Just as with compiling the kernel, module compilation depends on the configuration parameters specified in the .config file.
  5. Use the command “make modules install” to install the kernel modules into vmlinuz.
  6. Install the new kernel on the system by entering the “make install” command.
  - When the system reboots, it will begin running this new operating system.

Завантаження ОС

- After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The process of starting a computer by loading the kernel is known as **booting** the system. On most systems, the boot process proceeds as follows:
  1. A small piece of code known as the **bootstrap program** or **boot loader** locates the kernel.
  2. The kernel is loaded into memory and started.
  3. The kernel initializes hardware.
  4. The root file system is mounted.

- Some computer systems use a multistage boot process: When the computer is first powered on, a small boot loader located in nonvolatile firmware known as **BIOS** is run. This initial boot loader usually does nothing more than load a second boot loader, which is located at a fixed disk location called the **boot block**. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it must fit in a single disk block) and knows only the address on disk and the length of the remainder of the bootstrap program.

- Many recent computer systems have replaced the BIOS-based boot process with **UEFI** (Unified Extensible Firmware Interface). UEFI has several advantages over BIOS, including better support for 64-bit systems and larger disks. Perhaps the greatest advantage is that UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.
  Whether booting from BIOS or UEFI, the bootstrap program can perform a variety of tasks. In addition to loading the file containing the kernel program into memory, it also runs diagnostics to determine the state of the machine —for example, inspecting memory and the CPUand discovering devices. If the diagnostics pass, the program can continue with the booting steps. The bootstrap can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system and mounts the root file system. It is only at this point is the system said to be **running**.

- **GRUB** is an open-source bootstrap program for Linux and UNIX systems. Boot parameters for the system are set in a GRUB configuration file, which is loaded at startup. GRUB is flexible and allows changes to be made at boot time, including modifying kernel parameters and even selecting among different kernels that can be booted. As an example, the following are kernel parameters from the special Linux file /proc/cmdline, which is used at boot time:
  BOOT IMAGE=/boot/vmlinuz-4.4.0-59-generic
  root=UUID=5f2e2232-4e47-4fe8-ae94-45ea749a5c92
  BOOT IMAGE is the name of the kernel image to be loaded into memory, and root specifies a unique identifier of the root file system.

- To save space as well as decrease boot time, the Linux kernel image is a compressed file that is extracted after it is loaded into memory. During the boot process, the boot loader typically creates a temporary RAM file system, known as initramfs. This file system contains necessary drivers and kernel modules that must be installed to support the ***real*** root file system (which is not in main memory). Once the kernel has started and the necessary drivers are installed, the kernel switches the root file system from the temporary RAM location to the appropriate root file system location. Finally, Linux creates the systemd process, the initial process in the system, and then starts other services (for example, a web server and/or database). Ultimately, the system will present the user with a login prompt.

- Варто зазначити, що механізм завантаження не є незалежним від boot loader.
  - Therefore, there are specific versions of the GRUB boot loader for BIOS and UEFI, and the firmware must know as well which specific bootloader is to be used.
  - The boot process for mobile systems is slightly different from that for traditional PCs.
  - For example, although its kernel is Linux-based, Android does not use GRUB and instead leaves it up to vendors to provide boot loaders.
  - The most common Android boot loader is LK (for “little kernel”).
  - Android systems use the same compressed kernel image as Linux, as well as an initial RAM file system.
  - However, whereas Linux discards the initramfs once all necessary drivers have been loaded, Android maintains initramfs as the root file system for the device.
  - Once the kernel has been loaded and the root file system mounted, Android starts the init process and creates a number of services before displaying the home screen.
  - Finally, boot loaders for most operating systems—including Windows, Linux, and macOS, as well as both iOS and Android—provide booting into **recovery mode** or **single-user mode** for diagnosing hardware issues, fixing corrupt file systems, and even reinstalling the operating system.
  - In addition to hardware failures, computer systems can suffer from software errors and poor operating-system performance.

ДЯКУЮ ЗА УВАГУ!

Вимоги до ОС

- Вимоги до ОС можна розбити на 2 основні групи: користувацькі та системні.
  - ***Користувачі:*** ОС повинна бути зручною, простою для вивчення, надійною, безпечною та швидкою.
  - ***Розробники:*** аналогічний набір вимогA similar set of requirements can be defined by the developers who must design, create, maintain, and operate the system.
  - The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient.
  - Again, these requirements are vague and may be interpreted in various ways.
- There is, in short, no unique solution to the problem of defining the requirements for an operating system.
  - The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments.
  - For example, the requirements for Wind River VxWorks, a realtime operating system for embedded systems, must have been substantially different from those for Windows Server, a large multiaccess operating system designed for enterprise applications