CSE308 Operating Systems

Operating System Design, Implementation and Structure

Dr S. Rajarajan
SoC
SASTRA

Design goals

- At the highest level, the design of the system will be affected by the **choice of hardware** and the **type of system**:
  - batch processing
  - time sharing
  - single user or multiuser
  - distributed
  - real time
  - general purpose
  - mobile
- The requirements can, however, be divided into two basic groups:
  - **user goals and system goals.**

- **Users want:**
  - The system should be **convenient** to use, **easy to learn** and **to use**, **reliable, safe**, and **fast**
- **System administrators want**:
  - The system should be **easy to design**, **implement, and maintain**; and it should be **flexible, reliable, error free, and efficient**
- There is **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**.

Implementation – selection of language

- Early operating systems were written in **assembly language**.
- Now, most operating systems are in a **high-level language** such as **C or C++.**
- An operating system can be written in **more than one language.**
- The **lowest levels of the kernel** might be assembly language.
- **Higher-level routines** might be in C, and system programs might be in **C or C++ or PERL or Python**, or in shell scripts.

- **Advantages of using a higher-level language**
  - The code can be **written faster,**
  - Program is more **compact**
  - Easier to **understand and debug**
- In addition, improvements in compiler technology will improve the generated code for the entire operating system by **simple recompilation.**
- An operating system is far easier to **port—to move to some other hardware—** if it is written in a higher-level language
  - For example, **MS-DOS** was written in Intel 8088 **assembly language.** Consequently, it runs natively **only on the Intel X86** family of CPUs.
  - **Linux** operating system, in contrast, is written **mostly in C** and is available natively on a **number of different CPUs**

- **Disadvantage**
  - **Reduced speed**
  - **Increased storage** requirements
- This, however, is **no longer a major issue** in today's systems
- Although an **assembly-language** can produce **efficient small routines**, for **large programs** a **modern compiler** can perform **complex analysis** and **apply sophisticated optimizations** that produce **excellent code**.
- Modern processors have deep **pipelining** and **multiple functional units** that can handle the details of **complex dependencies** much more easily

Operating-System Structure

- A **large and complex** operating system software must be **engineered carefully** if it is to function properly and be modified easily.
- A common approach is to partition the task into **small components, or modules**.
- Each of these modules should be a **well-defined portion** of the system, with carefully defined **inputs, outputs,** and **functions**.

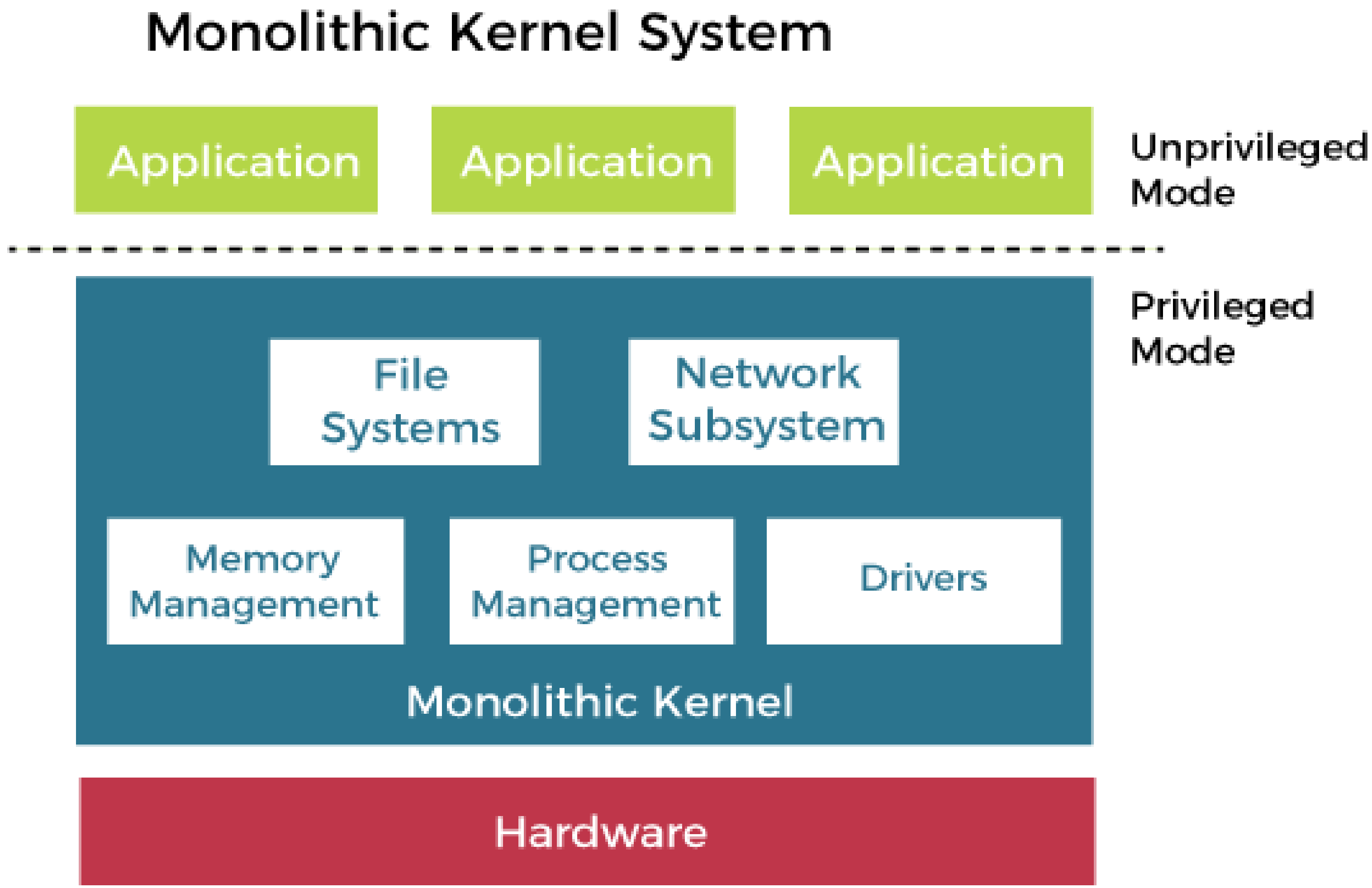
Objectives of Kernel

- **Scheduling Processes:** The kernel allocates CPU time to each process. After one process completes its execution, the kernel allocates the next one
- **Resource Allocation:** The kernel manages memory, peripheral devices, and CPU processes. It serves as a bridge between resources and processes, distributing memory and hardware component access.
- **Device Management:** The kernel oversees system devices, including I/O and storage devices. It facilitates data exchange between these devices and applications, handling information flow.
- **Interrupt Handling and System Calls:** The kernel manages task priorities, allowing high-priority tasks to take precedence. It also handles system calls, which are essentially software interrupts.
- **Memory Management:** The kernel allocates and deallocates memory for processes. It stores active processes in memory and releases memory when processes end.
- **Process Management:** The kernel is responsible for creating, executing, and terminating processes within the system. It takes charge of process management during task execution.

- There are three types of operating system structures(types of kernels):
  - Simple structure or Monolithic
  - Layered approach
  - Microkernel
  - Loadable kernel
  - Hybrid systems

Simple Structure/Monolithic Structure

- A monolithic kernel is an operating system architecture where the **entire OS** is running in **kernel space**.
- The monolithic kernel is a **static single binary file**.
- The monolithic operating system is a **very basic operating system** in which file management, memory management, device management, and process management are **directly controlled within the kernel.**
- This is an **old operating system** used to perform **small tasks** like batch processing and time-sharing tasks in banks
- **MS-DOS** is an example


Monolithic Kernel System
Application
Application
Application
Unprivileged Mode
Privileged Mode
File Systems
Network Subsystem
Memory Management
Process Management
Drivers
Monolithic Kernel
Hardware

Lack of protection in DOS

- DOS was written to provide **most functionality in the least memory space**, so it was **not carefully divided into modules**.
- In MS-DOS **application programs** are able to **access the basic I/O routines** to **write directly** to the display and disk drives.
- Such freedom leaves MS-DOS **vulnerable to errant (or malicious) programs**, causing entire system crashes when user programs fail.
- MS-DOS was also limited by the **hardware of its era.**
- Because the **Intel 8088** provides **no dual mode** and **no hardware protection**, the designers of MS-DOS had no choice but to leave the base hardware accessible

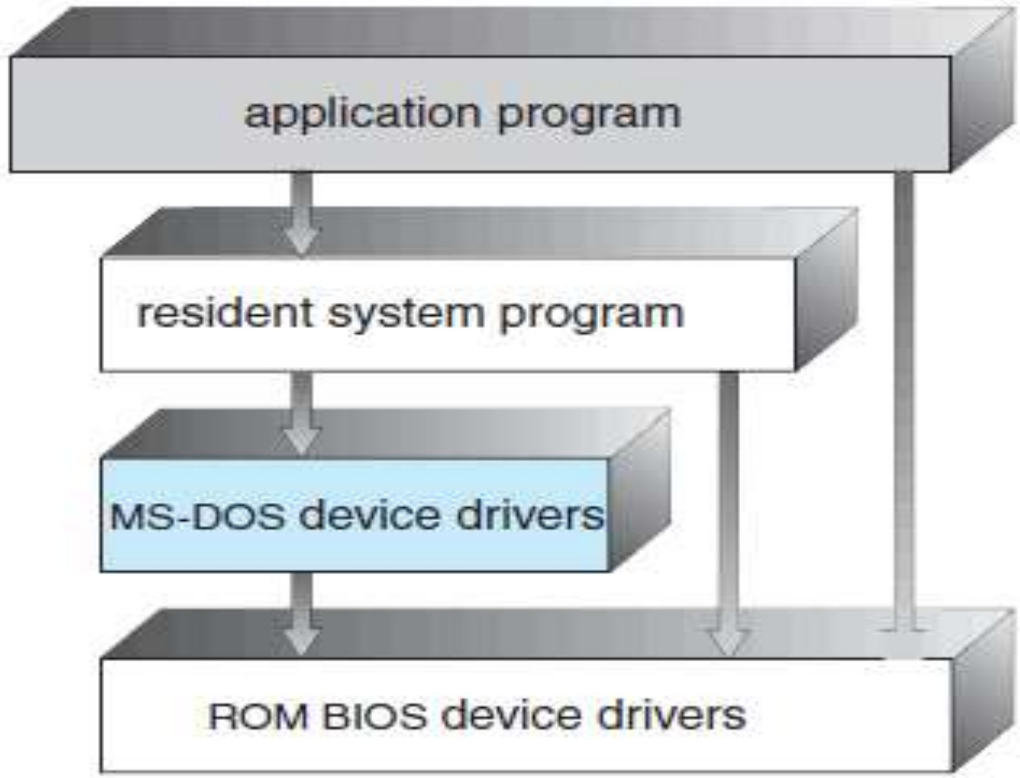


Figure 2.11 MS-DOS layer structure.

- Another example of **limited structuring** is the **original UNIX** operating system.
- Like MS-DOS, **UNIX** initially was **limited by hardware functionality.**
- It consists of two separable parts:
  - the **kernel** and the **system programs**.
- Everything **below the system-call interface** and **above the physical hardware** is the **kernel.**

- The kernel provides the **file system**, **CPU scheduling**, **memory management**, and other operating-system functions through **system calls**.
- Taken in sum, that is an **enormous amount of functionality** to be **combined into one level**.
- This **monolithic structure** was
  - difficult to **implement and maintain**.
- It had a distinct **performance advantage**:
  - there is **very little overhead** in the **system call** interface or in communication **within the kernel**.

Disadvantages

- If any service fails, then the entire system fails too.
- If any new features are added then it is necessary to modify the complete system.
- Coding and debugging in the kernel space is difficult.
- Bugs in one part of kernel space produce side effects in other parts also.
- These Kernels are huge and difficult to maintain and not always portable.

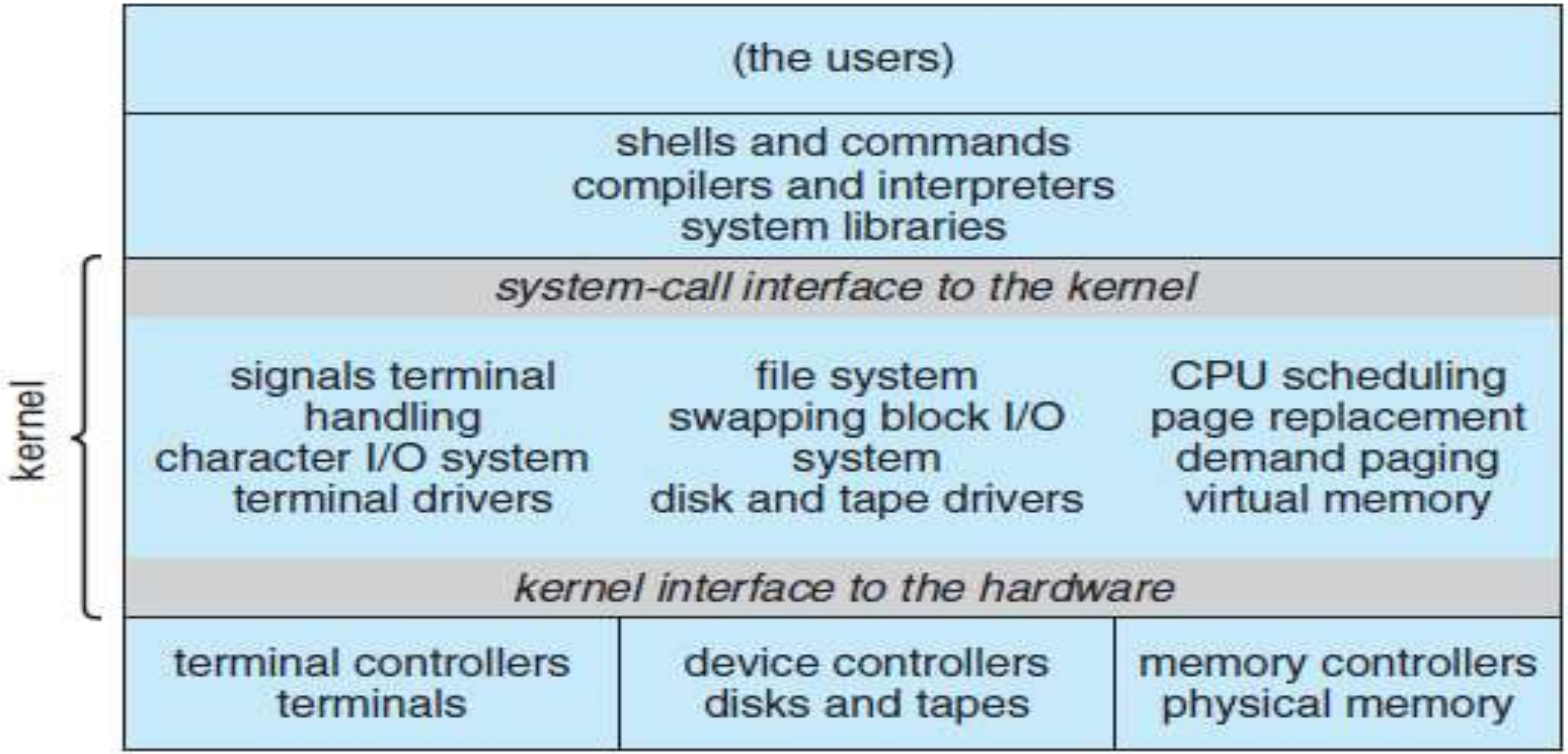
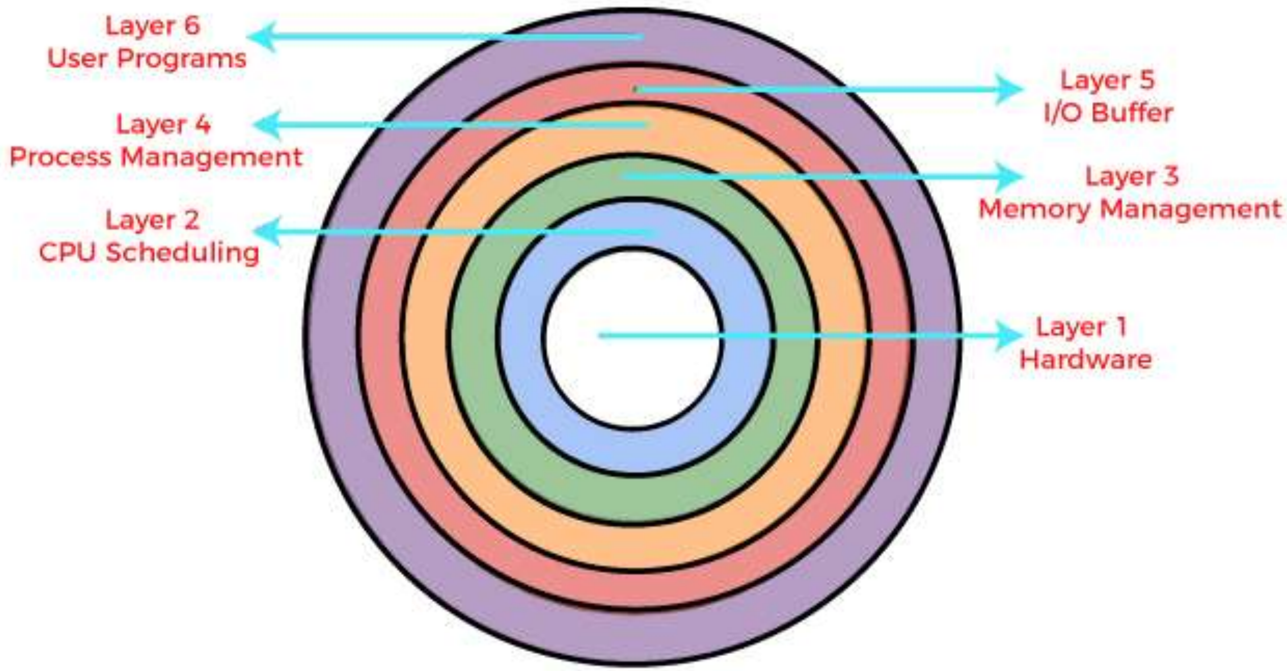


Figure 2.12 Traditional UNIX system structure.

Layered Approach

- With proper hardware support, operating systems can be broken into **smaller pieces**.
- Operating system can then retain much **greater control over the computer** and **the applications that run**
- Implementers have **more freedom** in changing the **inner workings** of the system and in creating **modular operating systems**.
- Under a **top down approach**, the overall functionality and features are determined and are separated into **components**.

- A system can be made **modular** in many ways.
- One method is the **layered approach, in which the operating system is broken into a number of layers** (levels).
- The **bottom layer** (layer 0) is the **hardware**; the **highest** (layer *N) is the* **user interface**.



- A typical operating-system layer consists of:
  - ***data structures and a set* of routines** that can be invoked by higher-level layers.
- There are some rules in the implementation of the layers as follows.
  - The **outermost layer** must be the **User Interface** layer.
  - The **innermost layer** must be the **Hardware** layer.
  - A particular layer can access operations at **all the layers present below it** but it cannot access the layers **above it (top-down**).
  - That is **layer n-1** can access all the layers from **n-2 to 0** but it cannot access the **nth layer.**

- The main **advantage** of the layered approach is **simplicity of construction** and **debugging**.
- The first layer **can be debugged** without any concern for the rest of the system
- The **major difficulty** with the layered approach involves **appropriately defining the various layers**.
- Because a layer can use only lower-level layers, **careful planning** is necessary
- For example, the **device driver for the backing store** (disk space used by virtual-memory algorithms) must be at a lower level than the **memory-management routines**, because memory management requires the ability to use the backing store.

Too many system calls

- A **final problem** with layered implementations is that they tend to be **less efficient**.
- For instance, when a user program executes an **I/O operation**, it executes a **system call** that is ***trapped*** to the **I/O layer**, which calls the **memory-management layer**, which in turn calls the **CPU-scheduling layer**, which is then passed to the hardware.
- At each layer, the **parameters may be modified**, data may need to be passed, and so on.
- Each **layer adds overhead** to the system call.
- The net result is a **system call** that **takes longer** than does one on a non-layered system.
- **Fewer layers** with **more functionality** are being preferred

- **Advantages :**
  - **Modularity :**
    . This design promotes modularity as each layer performs only the tasks it is scheduled to perform
  - **Easy debugging :**
    As the layers are discrete so it is very easy to debug. Suppose an error occurs in the CPU scheduling layer, so the developer can only search that particular layer to debug, unlike the Monolithic system in which all the services are present together.
  - **Easy update :**
    A modification made in a particular layer will not affect the other layers.
  - **No direct access to hardware :**
    The hardware layer is the innermost layer present in the design. So a user can use the services of hardware but cannot directly modify or access it, unlike the Simple system in which the user had direct access to the hardware.
  - **Abstraction :**
    Every layer is concerned with its own functions. So the functions and implementations of the other layers are abstract to it.

Microkernels

- As UNIX expanded, the **kernel became large** and **difficult** to **manage**
- In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that **modularized** the kernel using the **microkernel approach.**
- The core idea of a microkernel is enhancing reliability by **breaking the OS into smaller modules**.
- It structures the operating system by **removing all nonessential components from the kernel** and **implementing them** as system and **user-level programs.**
- The result is a **smaller kernel**.

- The user services are kept in **user address space**, and kernel services are kept under **kernel address space**, thus it **reduces the size of kernel** and **size of an operating system** as well
- Typically, microkernels provide **minimal process** and **memory management**, in addition to a **communication facility.**
- The main **function of the microkernel** is to **provide communication between the client program and the various services** that are also running in user space.
- This communication is provided through **message passing.**

- For example, if the **client program** wishes to **access a file**, it must **interact with the file server**
- The **client program** and **server never interact directly**.
- Rather, they communicate indirectly by **exchanging messages with the microkernel**

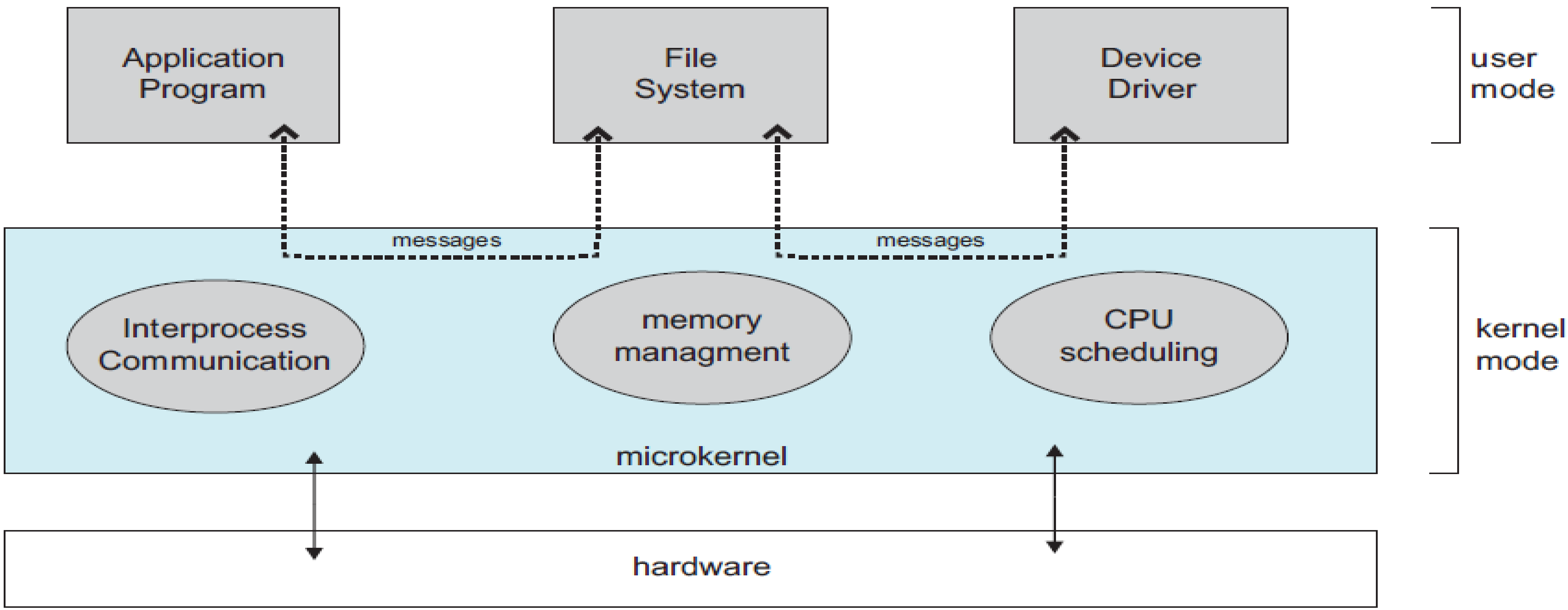


Figure 2.14 Architecture of a typical microkernel.

Advantages

- 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**.
- 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.
- **Fault tolerance:** If a service fails, the rest of the operating system remains **untouched.**

Loadable kernel modules (LKM)

- Perhaps the best current methodology for operating-system design involves using **loadable kernel modules.**
- Here, the kernel has a set of **core components** and **links in additional services via modules**, either **at boot time** or **during run time**.
- A **loadable kernel module** (**LKM**) is an object file that contains code to extend the running kernel, or so-called ***base kernel***, of an operating system.
- LKMs are typically **used to add support for new hardware** (as device drivers) and/or **file systems**, or for **adding system calls**.
- When the functionality provided by an LKM is **no longer required**, it can be **unloaded** in order **to free memory** and other resource
- This type of design is common in modern implementations of UNIX, such as **Solaris, Linux, and Mac OS X**, as well as **Windows**.

- **Without loadable kernel** modules, an operating system would have to **include all possible anticipated functionality compiled** directly into **the base kernel**.
- Much of that functionality would reside in memory without being used, **wasting memory**, and would require that users rebuild and reboot the base kernel every time they require **new functionality**
- **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**.
- The idea of the design is for the **kernel to provide core services** while **other services** are **implemented dynamically**, as the kernel is running.

- It is more **flexible** than a **layered system**, because **any module can call any other module.**
- Comparing to **microkernel approach** it is more efficient, because modules do **not need to invoke message passing** in order to communicate.

- The Solaris operating system structure is organized around a core kernel with seven types of loadable kernel modules:

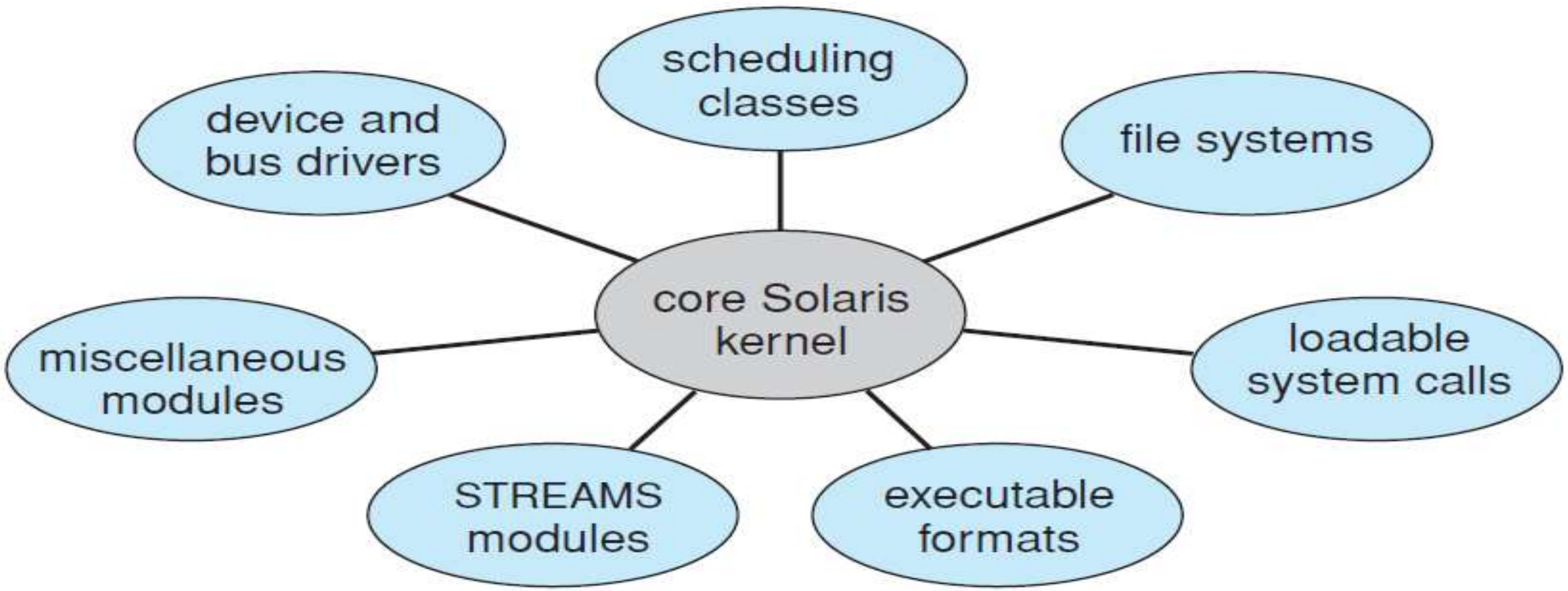


Figure 2.15 Solaris loadable modules.

Hybrid Systems

- In practice, very few operating systems adopt a single, strictly defined structure.
- Instead, they **combine different structures**, resulting in hybrid systems.
- For example, both **Linux and Solaris** are **monolithic**, because having the operating system **in a single address space** provides very efficient performance.
- However, **they are also modular**, so that **new functionality can be dynamically added** to the kernel.

- ***Windows*** systems also provide support for dynamically loadable kernel modules.
- Windows is largely monolithic as well, 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***