Operating Systems (INFR10079)

2020/2021 Semester 2

Structure

(Operating System Structure)

abarbala@inf.ed.ac.uk

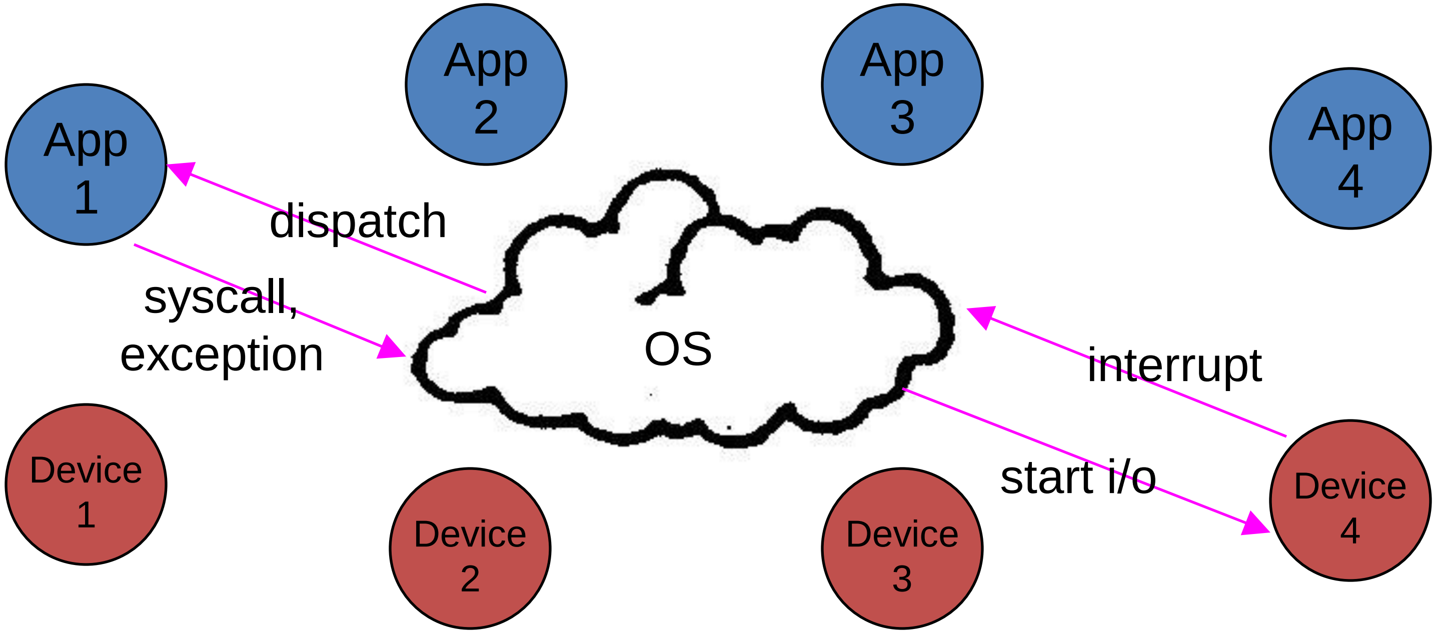
Chapter 2.4, 2.7, 2.8

Overview

- *Architecture impact*
- *Application-Operating System interaction*
- **Operating System structure**

OS Structure

- OS mediates access and abstracts away ugliness
- OS sits between **applications** and the **hardware**
  - Applications **(App)** request services
    - **Explicitly** via syscalls
    - **Implicitly** via exceptions
  - Devices **(Device)** request attention via interrupts



Operating System Design and Implementation

- Design and Implementation of OS not "solvable"
  - but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
  - **User** goals: convenient to use, easy to learn, reliable, safe, and fast
  - **System** goals: easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Affected by choice of hardware, type of system

Operating System Design and Implementation

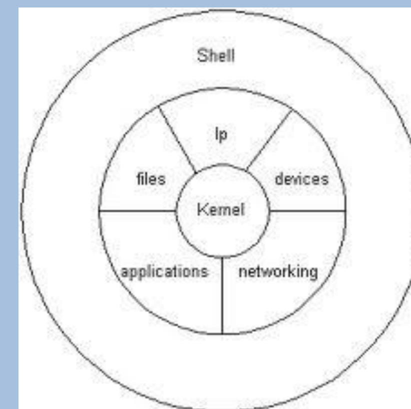
- Important **principles to separate**
- **Policy**: ***What*** will be done? (Algorithm)
- **Mechanism**: ***How*** to do it?

- Separation allows maximum flexibility
  - Policies are likely to change across places or over time
  - A general mechanism can support a wide range of policies

- Microkernel OSes are based on such principle
  - A core kernel implements the mechanisms
  - Policies are implemented outside the core kernel
    - Easily modifiable

Major OS Services

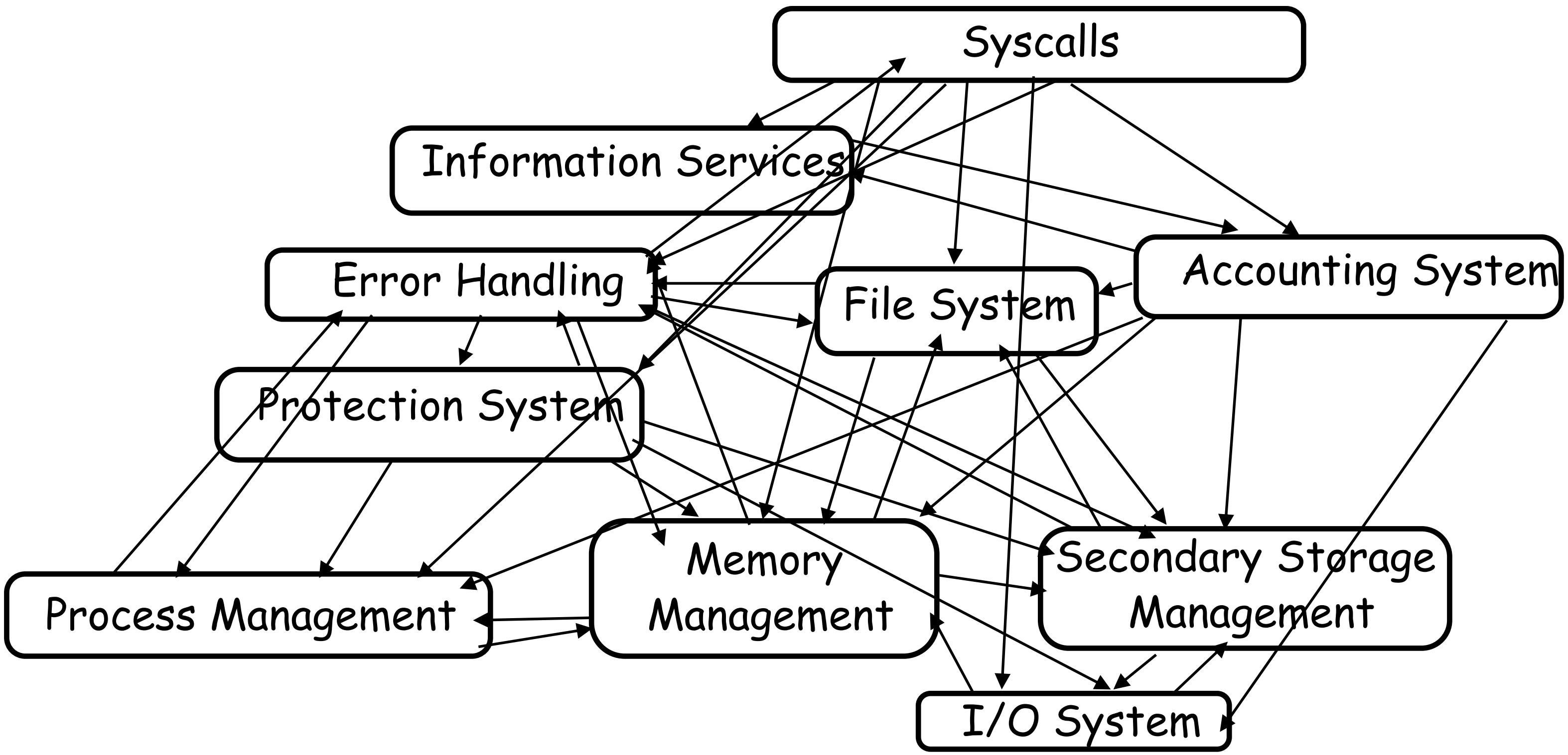
- processes
- memory
- I/O
- secondary storage
- file systems
- protection
- networking
- shells (i.e., command interpreter)
- GUI
- etc.



Systems programs – outside the kernel

OS Structure #1

- It's not always clear how to stitch OS **services** together



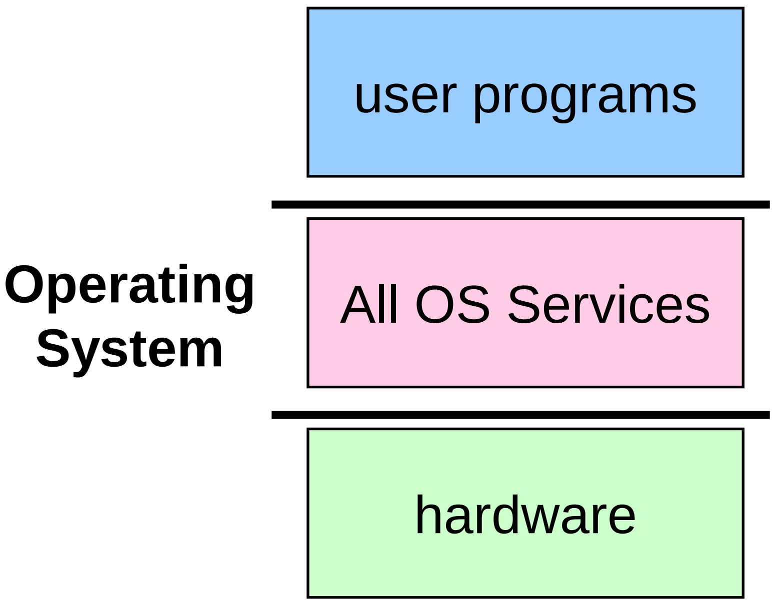
OS Structure #2

- Major issues
  - how do we organize all these?
  - what are all of the code modules, and where do they exist?
  - how do they cooperate?

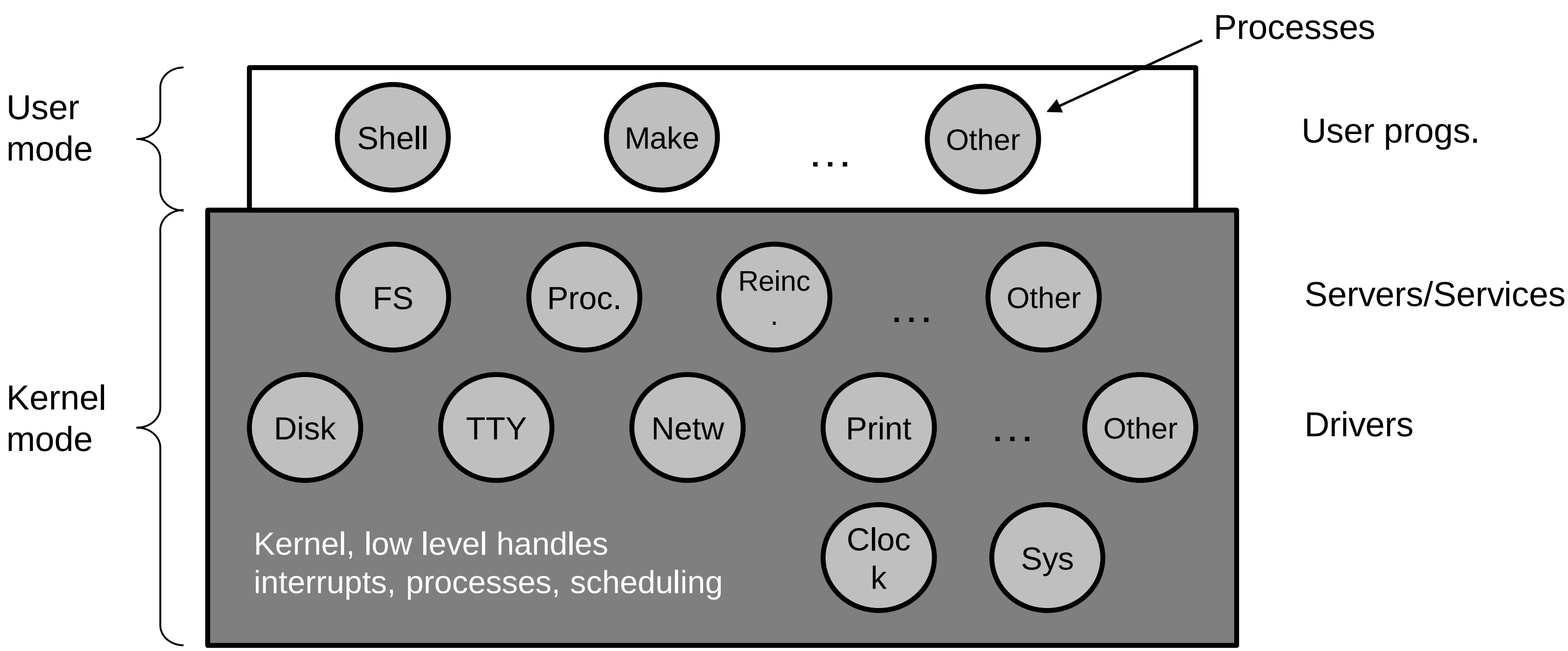
- Massive software engineering and design problem
  - design a large, complex program that
    - performs well
    - is reliable
    - is extensible
    - is backwards compatible
    - etc.

Monolithic **OS Design** #1

- Likely the earliest OS organization
- **UNIX** was built as monolithic
  - **Linux** is built as monolithic



Monolithic Example: Linux

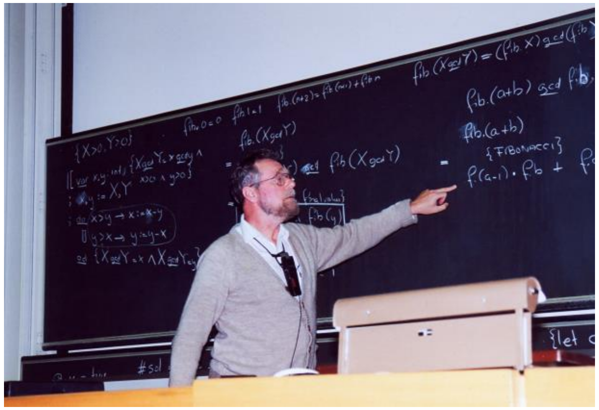


Monolithic **OS Design** #2

- Major **advantage**
  - **cost** of subsystems interactions **is low** (procedure call)
- Disadvantages
  - hard to understand
  - hard to modify
  - unreliable (no isolation between system modules)
  - hard to maintain
- What is the alternative?
  - find a way to organize OS subsystems to simplify its design and implementation

Layered **OS Design**

- The traditional approach is layering
  - implement OS as a set of layers
  - each layer presents an enhanced 'virtual machine' to the layer above
- The first description of this approach was Dijkstra's THE system
  - Layer 5: Job Managers
    - Execute users' programs
  - Layer 4: Device Managers
    - Handle devices and provide buffering
  - Layer 3: Console Manager
    - Implements virtual consoles
  - Layer 2: Page Manager
    - Implements virtual memories for each process
  - Layer 1: Kernel
    - Implements a virtual processor for each process
  - Layer 0: Hardware
- Each layer can be tested and verified independently

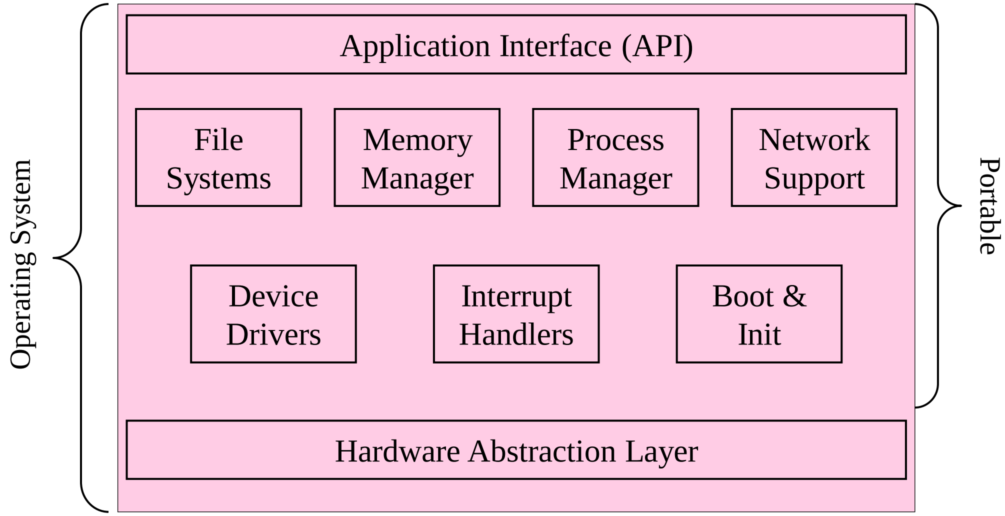


Problems with layering

- **Imposes** hierarchical structure
  - but real systems are more complex
    - File system requires virtual memory services
    - Virtual memory would like to use files for its backing store
  - **strict layering isn't flexible enough**
- Poor performance
  - each layer crossing has **overhead** associated with it
- Disjunction between model and reality
  - systems modeled as layers, but not really built that way

Hardware Abstraction Layer

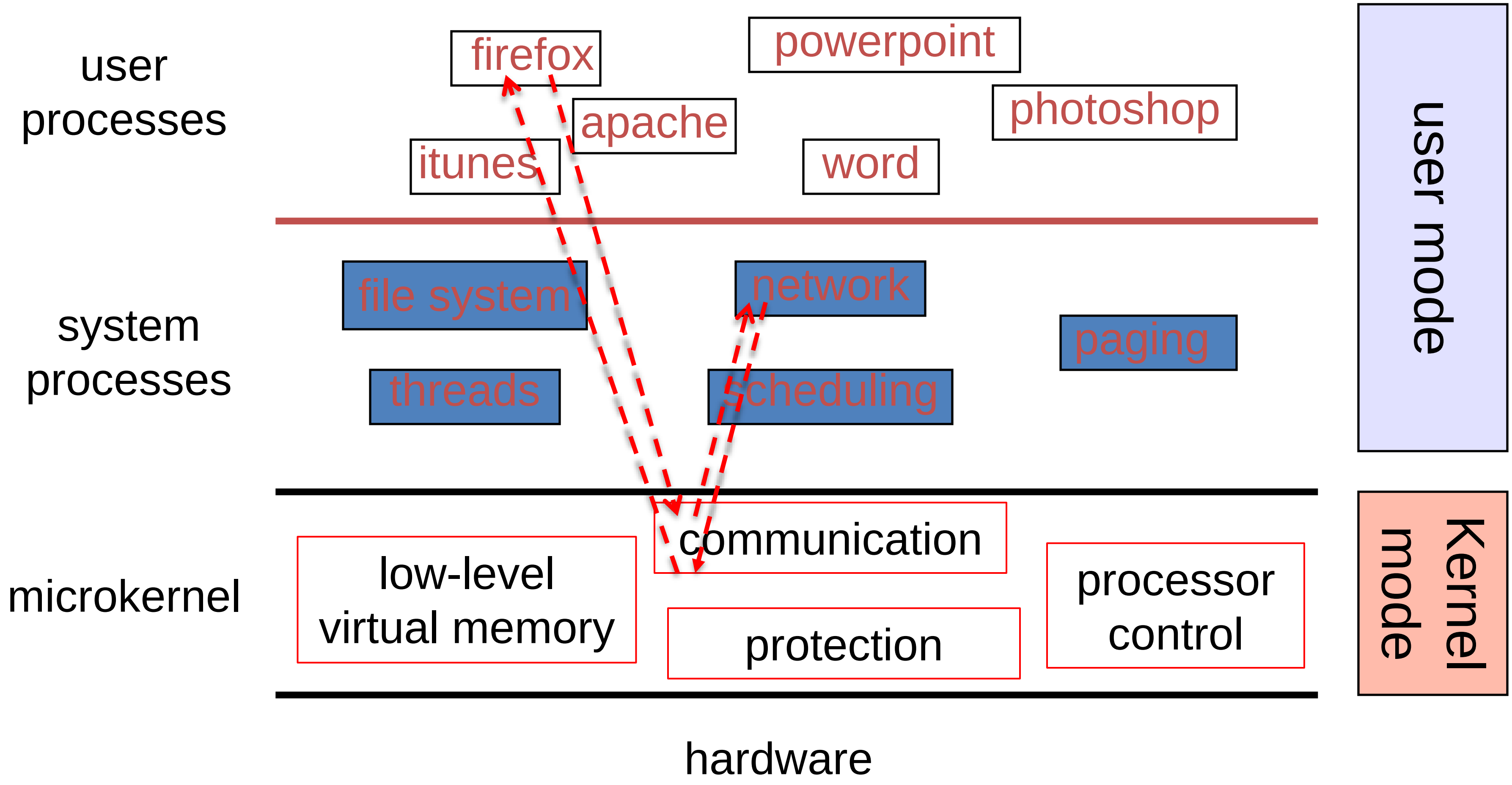
- An **example** of layering in modern operating systems
  - Windows, etc.
- Goal: separates hardware-specific routines from the **core kernel** of the OS
  - Provides portability
  - Improves readability



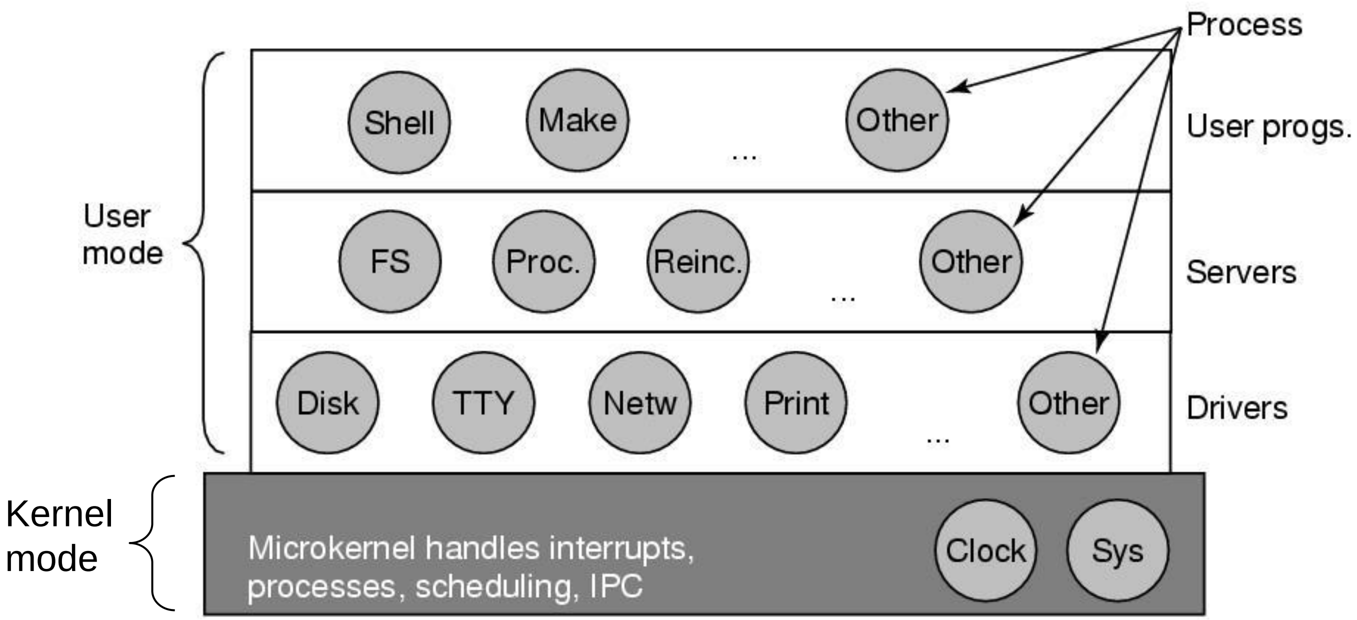
Microkernel **OS Design**

- Popular in the late 80's, early 90's
  - **recent** resurgence of popularity
- Goal
  - **minimize** what goes in kernel
  - organize rest of OS as user-level processes (services)
- This results in
  - better **reliability** (isolation between components)
  - ease of **extension and customization**
  - poor **performance** (user/kernel boundary crossings)
- First microkernel system was Hydra (CMU, 1970)
  - Follow-ons: Mach (CMU), Chorus (French UNIX-like OS), MINIX (UNIX-like OS from Amsterdam)

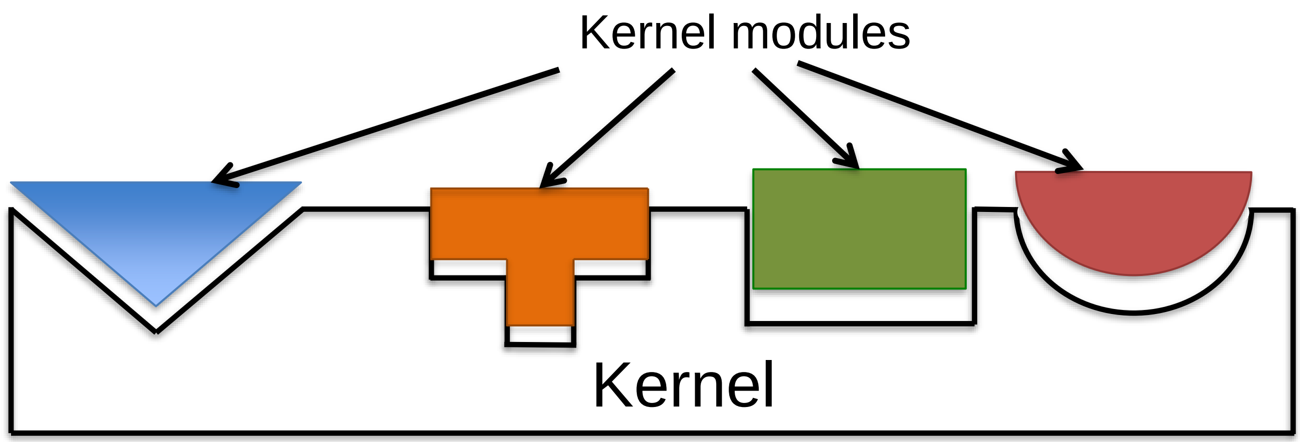
Microkernel Structure Illustrated



Microkernel Example: MINIX



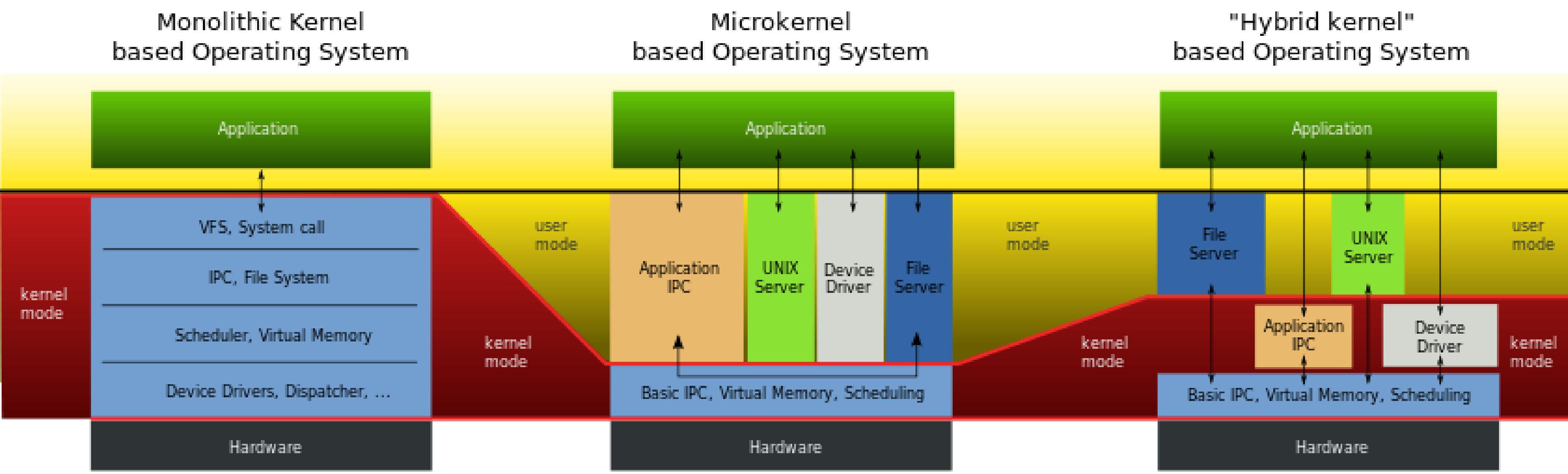
Loadable Kernel Modules



- **Core services** in the kernel, **others** dynamically loaded
- Common in modern implementations
  - **Monolithic**: load the code in kernel space (Solaris, Linux, etc.)
  - **Microkernel**: load the code in user space (any)
- Advantages
  - **Convenient**: no need for rebooting for newly added modules
  - **Efficient**: no need for message passing unlike microkernel
  - **Flexible**: any module can call any other module unlike layered model

Hybrid **OS Design**

- Many **different** approaches
  - Key idea: exploit the benefits of monolithic and microkernel designs
  - Windows, Xnu/Darwin, DragonFly BSD, …
- Extensibility via kernel modules



Picture Copyright of Wikipedia

Summary

- Fundamental distinction between user and privileged modes supported by most hardware
- OS design has been an evolutionary process of trial and error
- Successful OS designs have run the spectrum from monolithic, to layered, to micro kernels
- The role and design of an OS are still evolving
- There is no “ideal” OS structure