Operating System Structures

Chapter 2

Jobs of the OS

- The OS typically has to provide a few important things
- Support program execution (*processes*)
  - Programs must get a chance to run on a CPU (sometimes)
  - System calls to:
    - Start other programs (e.g., fork(), execl() on POSIX)
    - Check the status of itself and other programs (e.g., getpid(), getrusage(), wait())
    - Terminate itself or other programs (e.g., _exit(), kill())
- Provide memory for programs
  - Programs need to be able to access memory **without** making a system call every time.
  - System calls to get more memory or give it back (e.g., sbrk(), mmap())

Jobs of the OS

- Allocate, use and return files and other resources for programs
  - (e.g., open(), read(), close())
- Provide protection and security mechanisms
  - (e.g., chmod(), open())
- Permit *inter-process communication*
  - If the OS is doing a good job with protection
  - … then programs are going to need help to work together

Jobs of the OS

- Respond to errors
  - In hardware
  - In user programs
- Perform some accounting
- Provide some kind of user interface
  - Including a collection of *system utilities*
  - Maybe as a *command-line interface*
  - … or a *graphical interface (or, both)*

Getting Everything Started

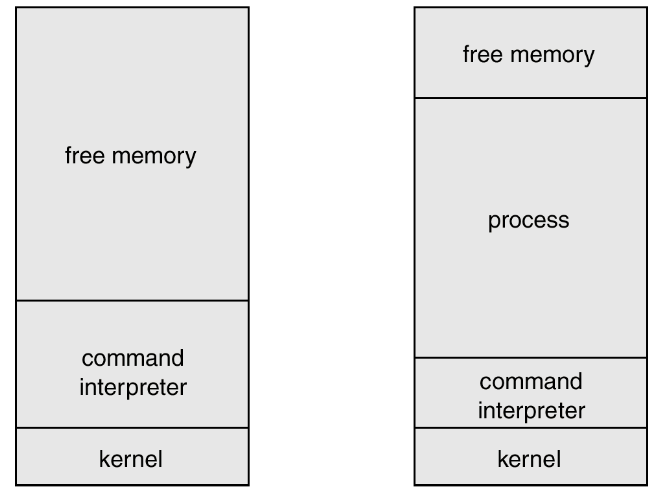
- System Boot
- Must get the kernel into memory and start running a few *system utilities / services*.
- At power on, typically:
  - Start running firmware at a known address
  - Load a *boot loader* from secondary storage
  - Bootstrap loader copies kernel into memory and starts executing it
  - Kernel initializes data structures (e.g., interrupt vectors)
  - Kernel starts running at least one system utility (e.g., Unix init or systemd on Linux)

How to Design and Build an OS

- We had to figure out:
  - What an OS should do
  - How it should be designed
- What we'd like
  - Minimize performance overhead
  - Promote portability
  - Promote extensibility
  - Minimize consequences of a failure in the OS
- OS and Hardware have grown up together

Single-Tasking

- Resources scarce and expensive
- Limited hardware support for OS
- MS-DOS Execution Environment
  - One user program at a time
  - Hardware protection not as important

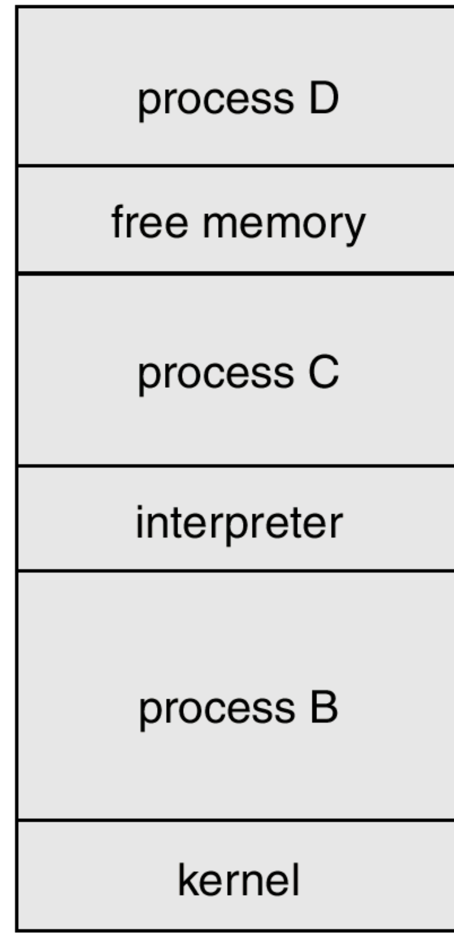


(a) Waiting for a Command

(b) Running a Program

Early Multi-Tasking

- With more resources we can:
  - Keep multiple programs (processes) running in memory
  - Let them share resources (including the CPU)
  - That's *multi-tasking*
  - Hardware protection is now important

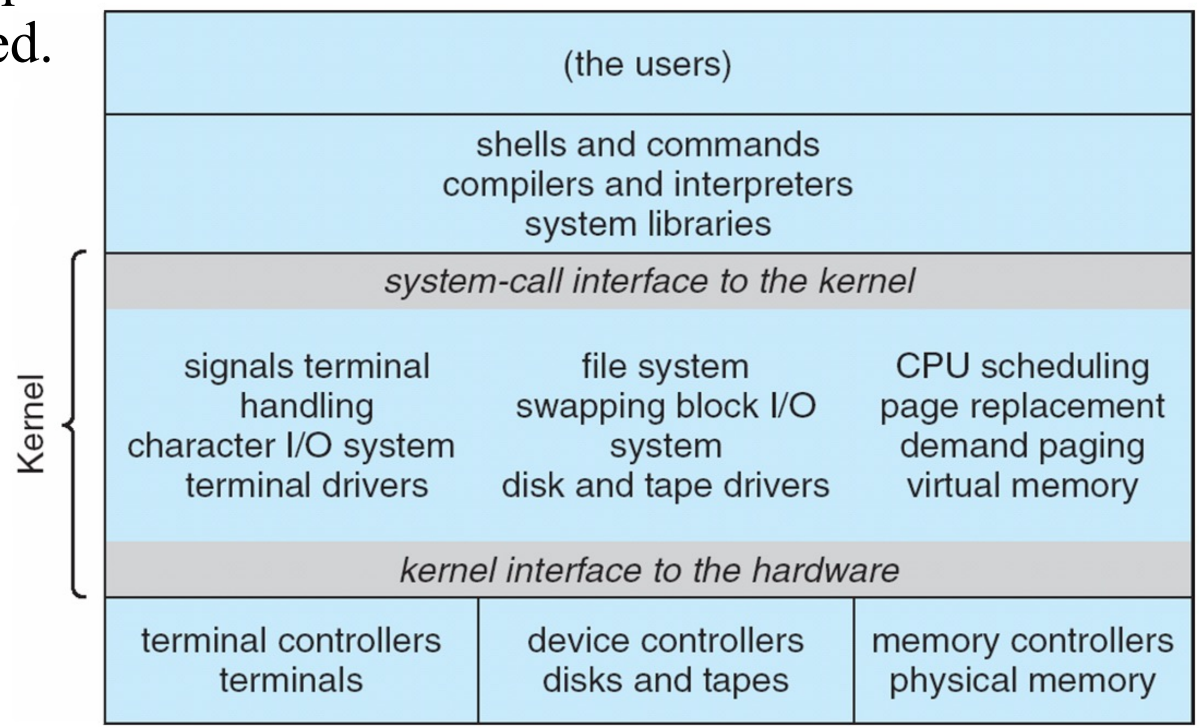


Unix System Structure

- Early OS Organization
  - Limited hardware functionality
  - Little structuring
- Two main components
  - *Kernel*
    - Everything behind the system call interface
      - File System, CPU Scheduling, Memory Management, etc.
      - All running in kernel mode
    - Mostly implemented as a collection of functions
  - *System utilities*, e.g., shell, ls, cat, init, fsck
    - To keep the system running
    - To let users interact with the system.

Unix System Structure

- An example of a *monolithic kernel*
  - All traditional OS services compiled into one big kernel
  - All running in kernel mode.
- This has some disadvantages:
  - Code organization can be unorganized.
  - Harder to modify or extend.
  - Catastrophic if there's a bug in one part of the code.

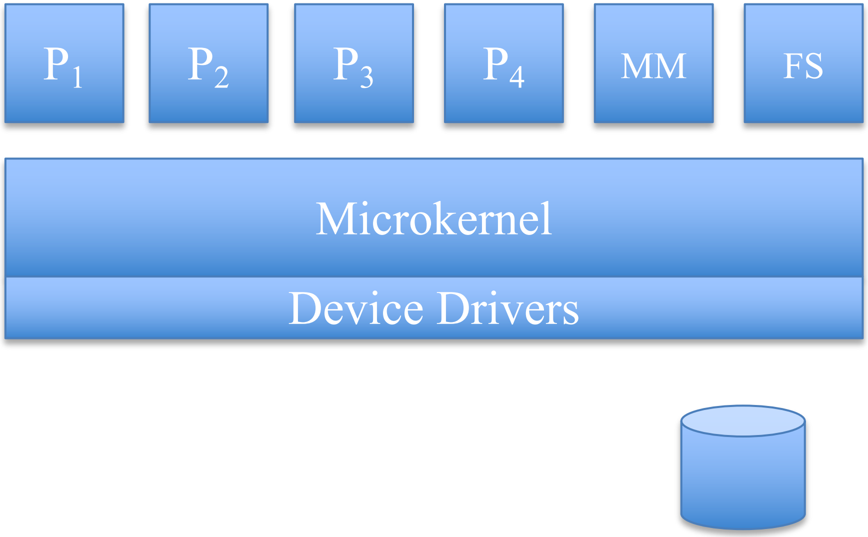


Microkernel Organization

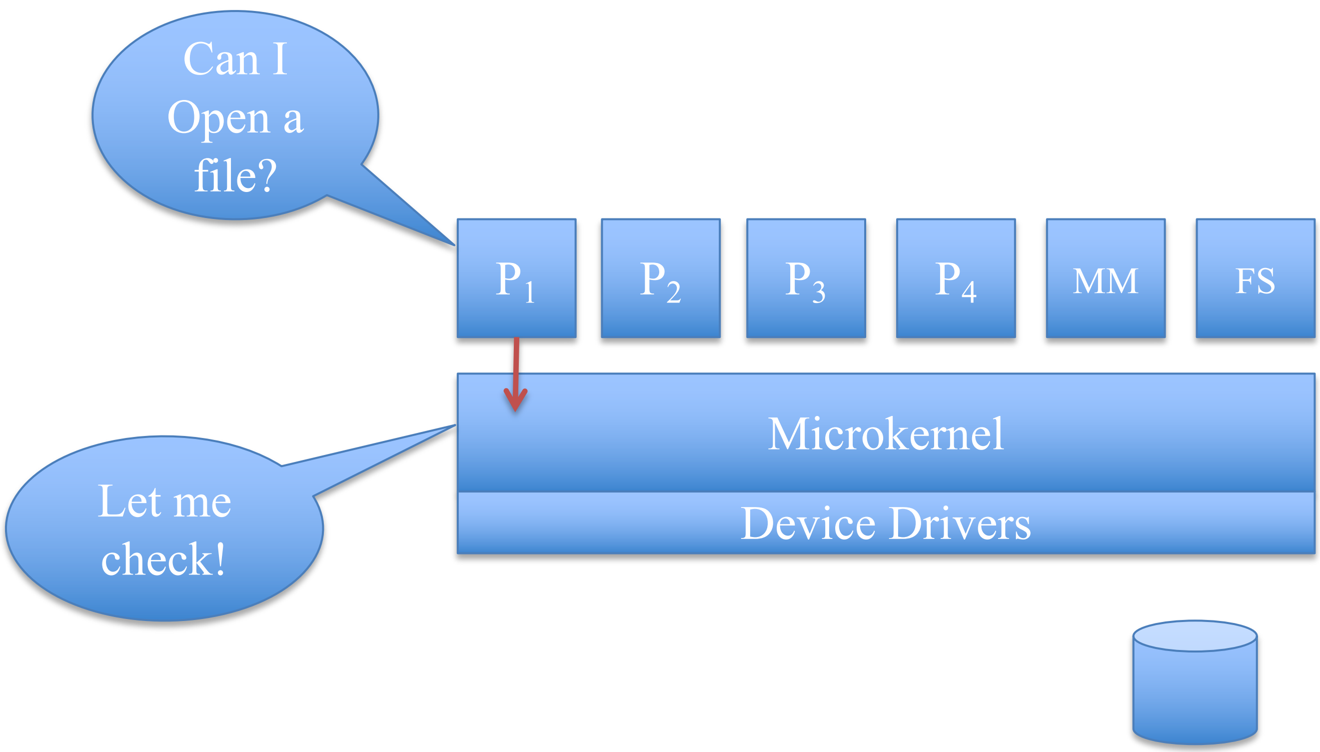
- Move traditional OS components into *user space* (i.e., out of kernel mode)
  - OS functions performed by user-mode processes
  - Do this with as many OS functions as possible
  - *e.g.,* file system
  - So, hopefully, you're left with a very small kernel
- Use of some OS features becomes inter-process communication
- Benefits
  - Easier to extend, add new features in user space
  - Easier to port to a new architecture
  - Less code in kernel mode, so more resistant to failure
  - More secure

Microkernel in Action

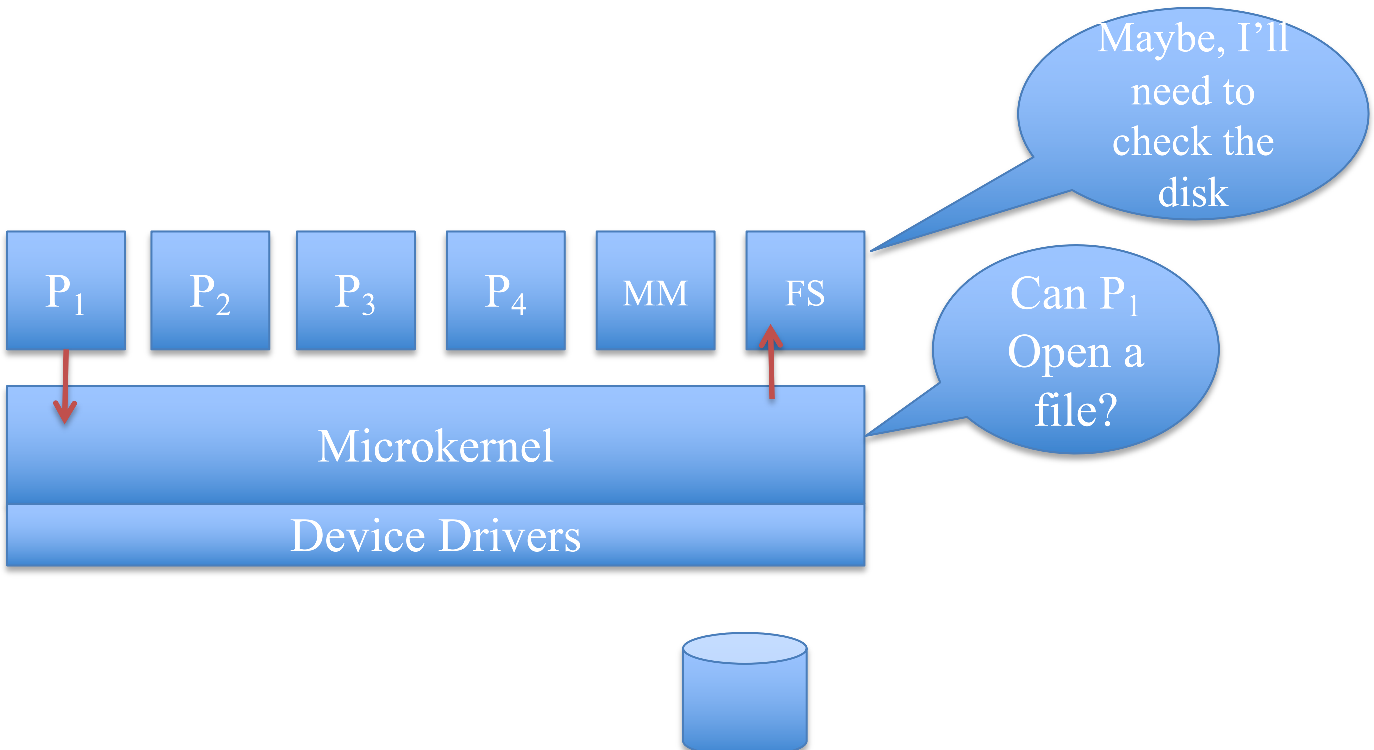
- Pretend user process P_1 needs to open a file



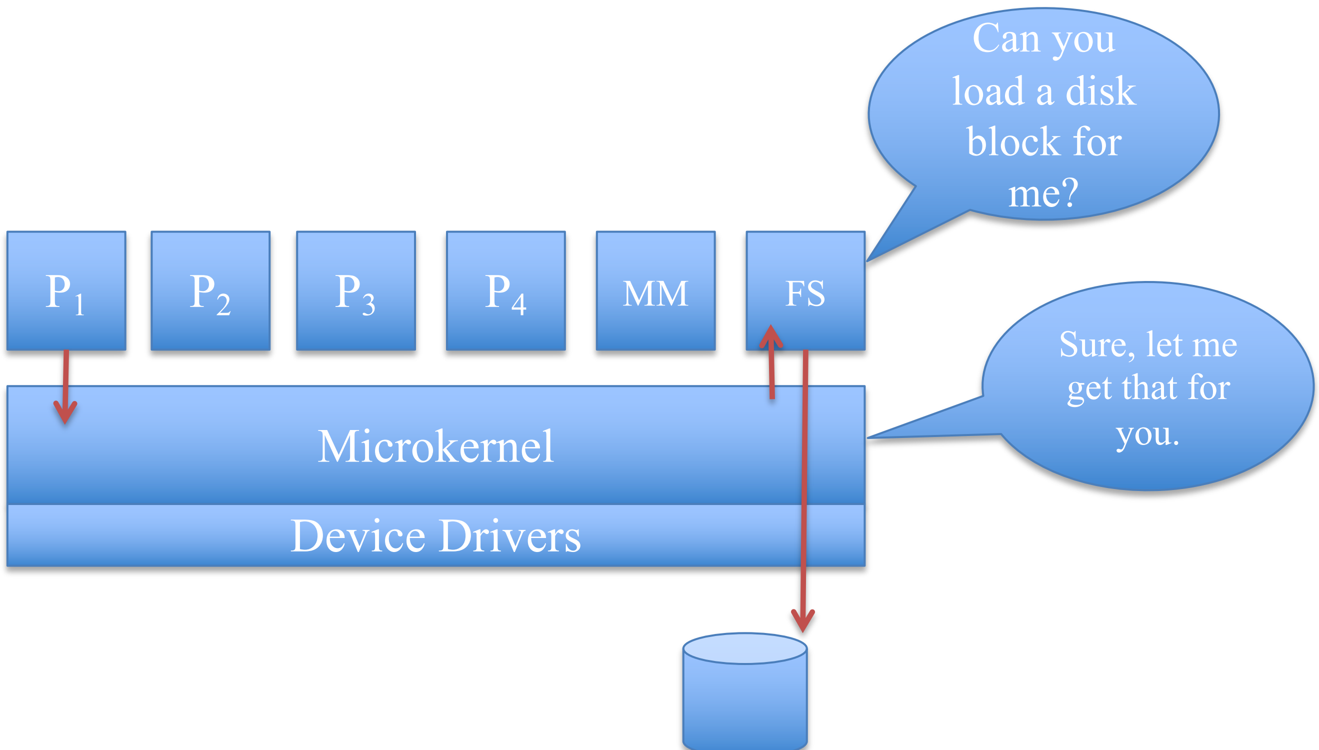
Microkernel in Action



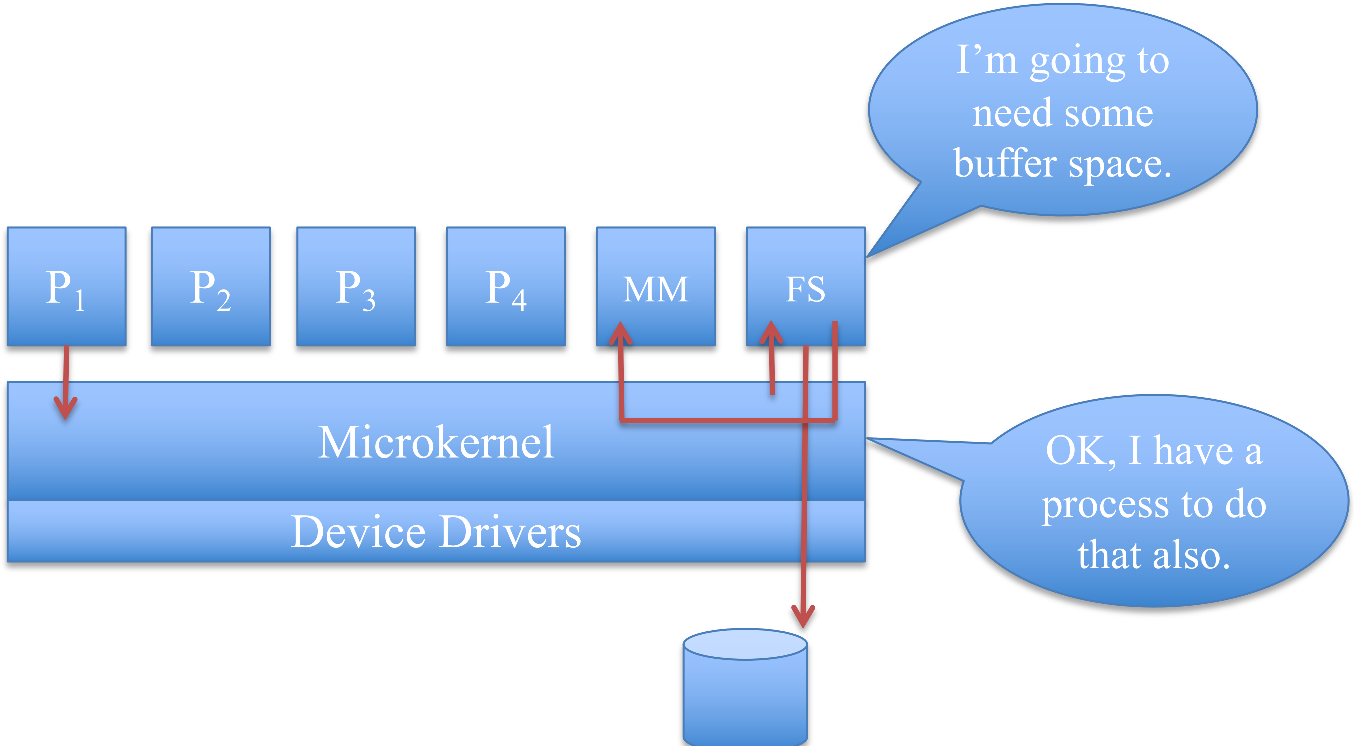
Microkernel in Action



Microkernel in Action



Microkernel in Action



Modular Organization

- Some of the advantages of Microkernel
  - Define categories OS services (e.g., a file system)
  - Each with a well-defined interface
  - e.g., device driver, file system, scheduler
- Kernel *module*
  - Object-oriented approach
  - Each type of module is like an abstract base class
  - Different subclasses, for different module implementations
  - Each module loadable during kernel execution

Microkernel Organization

- Disadvantages
  - More communication & system calls → more overhead
- Requires
  - Well-defined categories of user-mode OS services
  - Well-defined interfaces for each type of service

OS Structures

- Monolithic vs Microkernel
- Almost all modern operating systems are not one pure model, and they use a hybrid approach to combines multiple approaches to address performance, security, usability needs, including Linux, MacOSX, Windows, and Solaris.