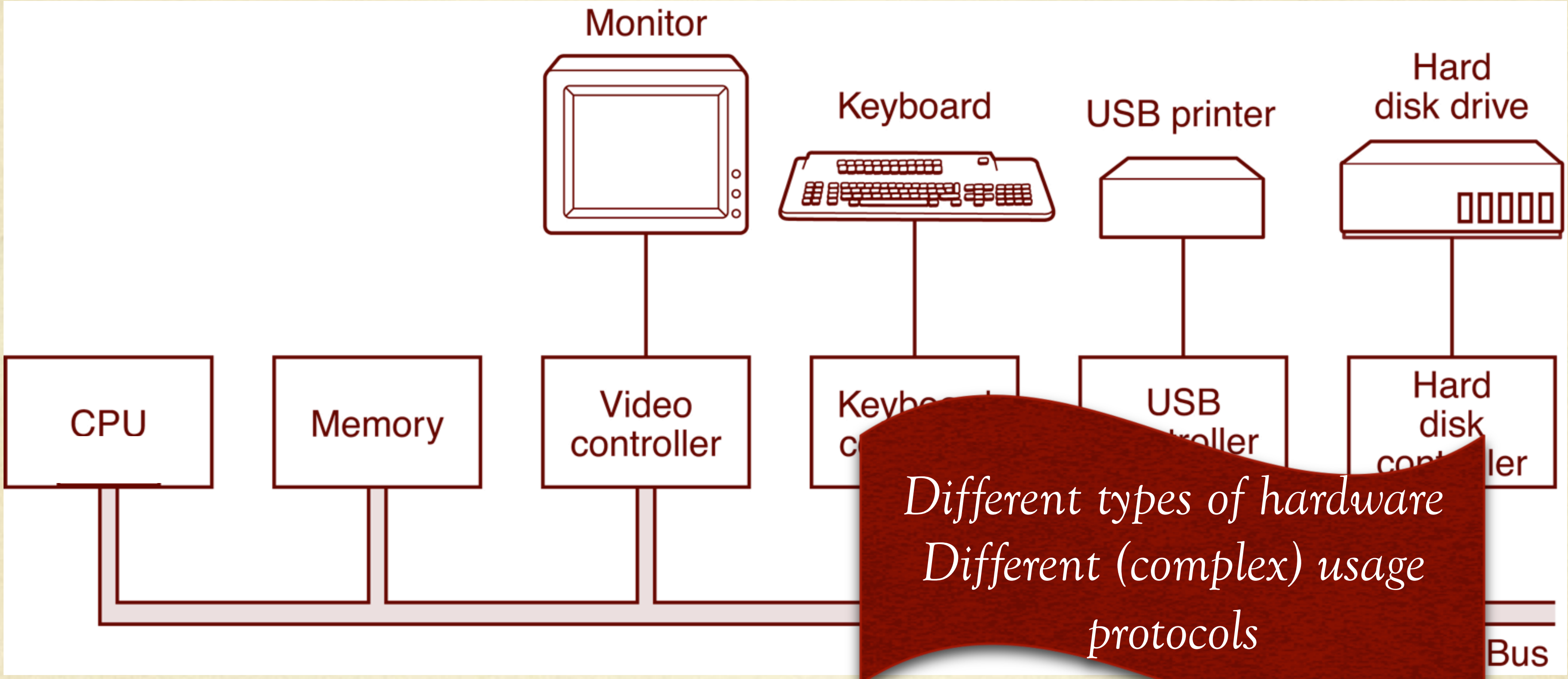
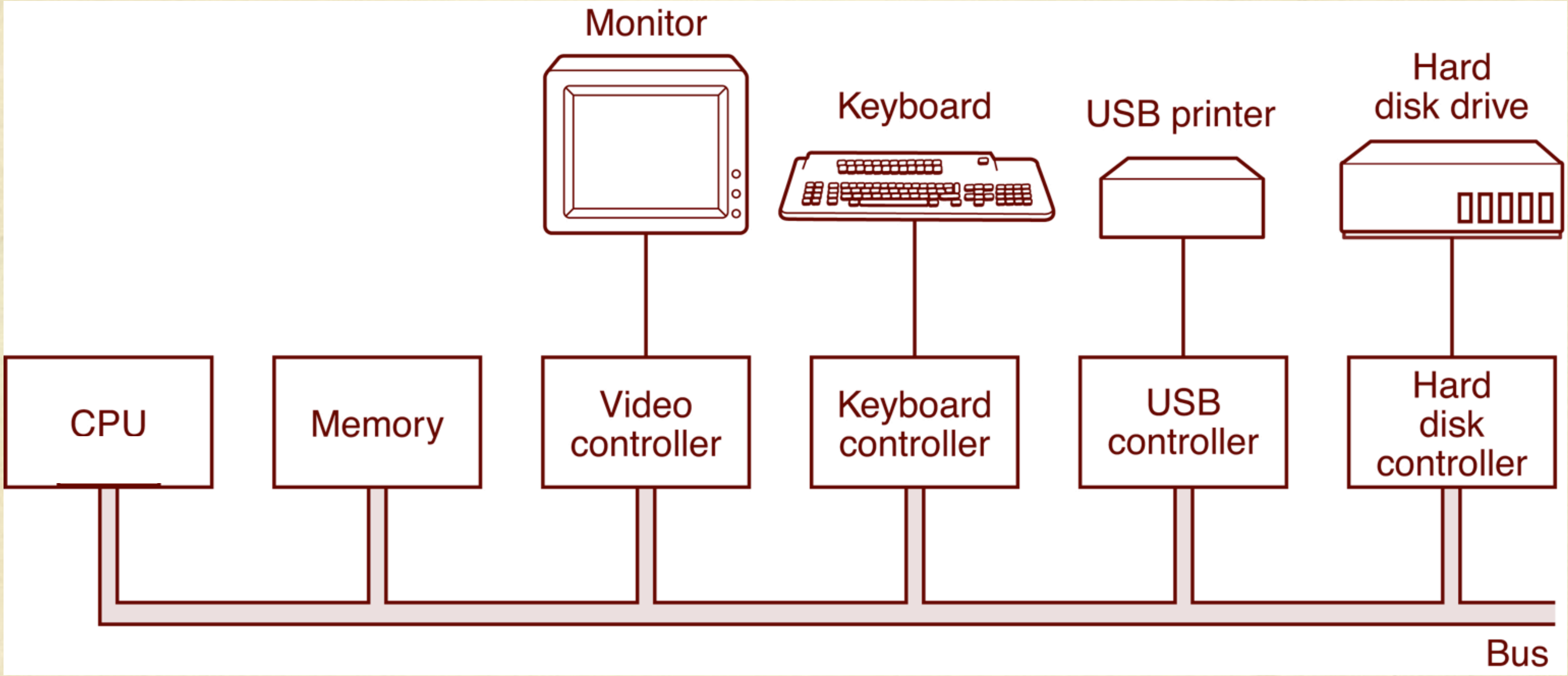
Introduction to Operating Systems



Different types of hardware
Different (complex) usage protocols

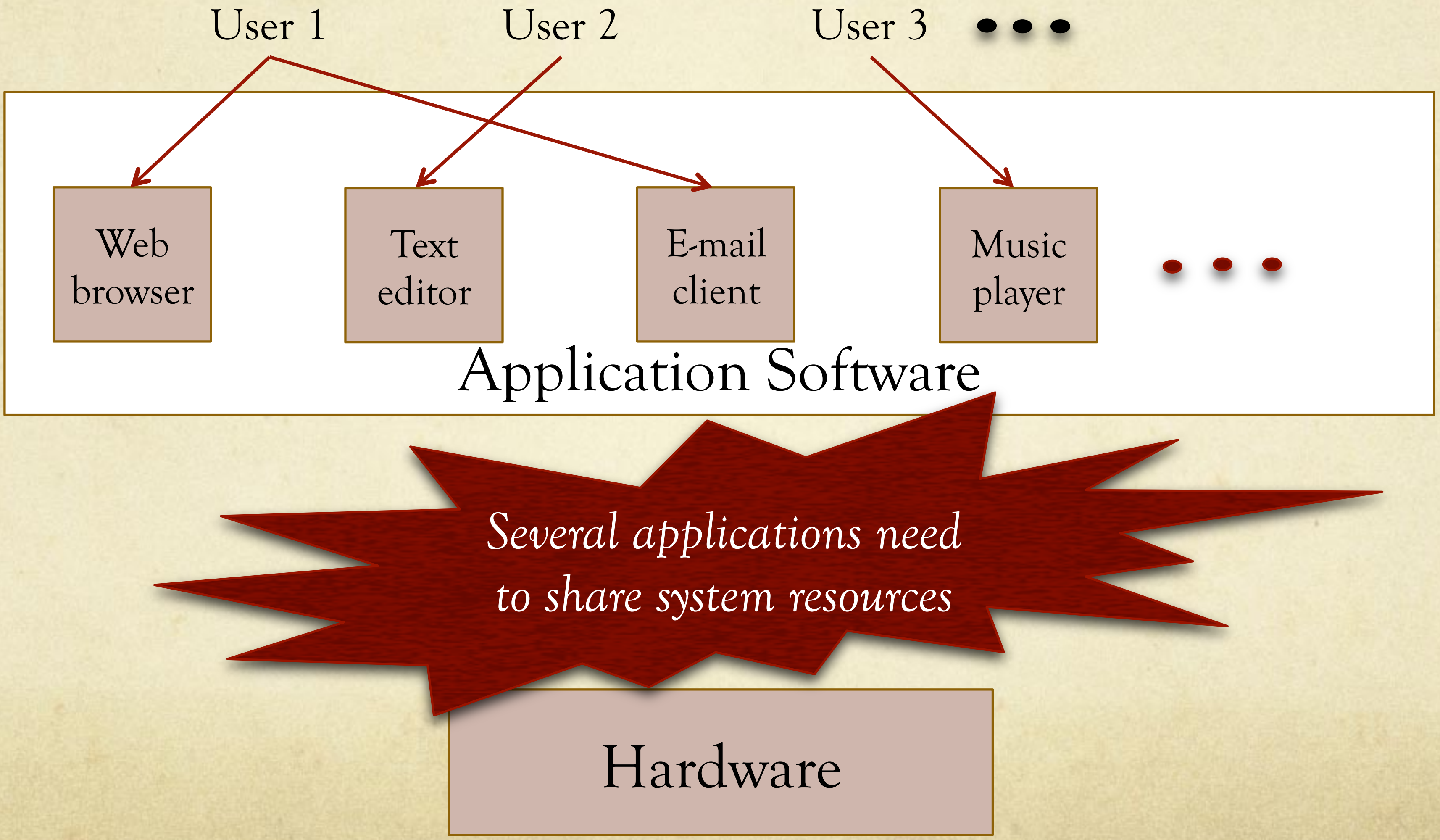
- *Low level hardware controller detail too complicated for application programmers/users*
- *Hardware state can get messed up through use of incorrect protocols*



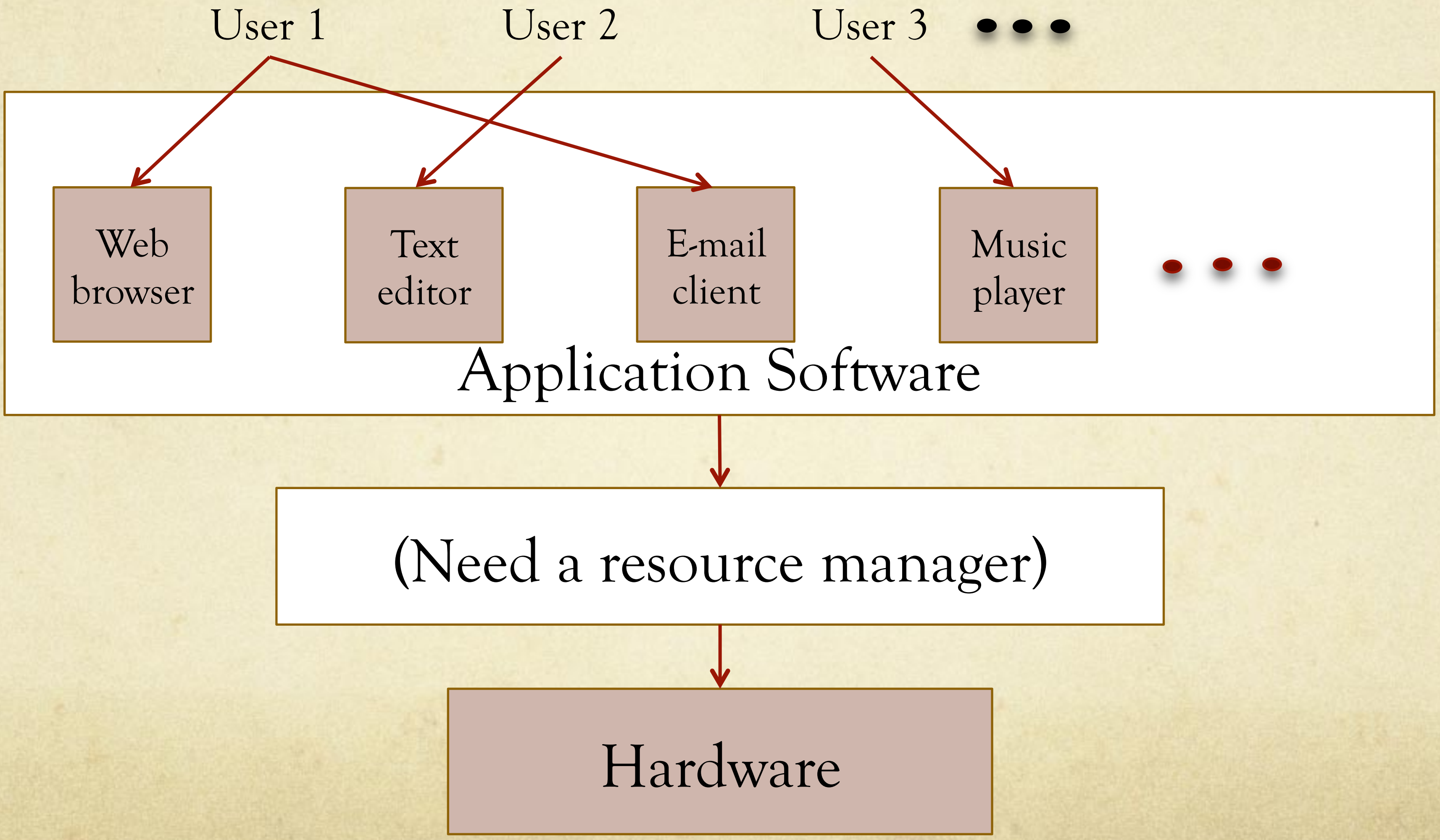
- *Need special software that*
  - *Knows how to interact with hardware controllers*
  - *Provides simpler external interface to application programmers/users*

Abstraction!

Consider another aspect



Consider another aspect



User 1 User 2 User 3 • • •

Web browser

Text editor

E-mail client

Music player

• • •

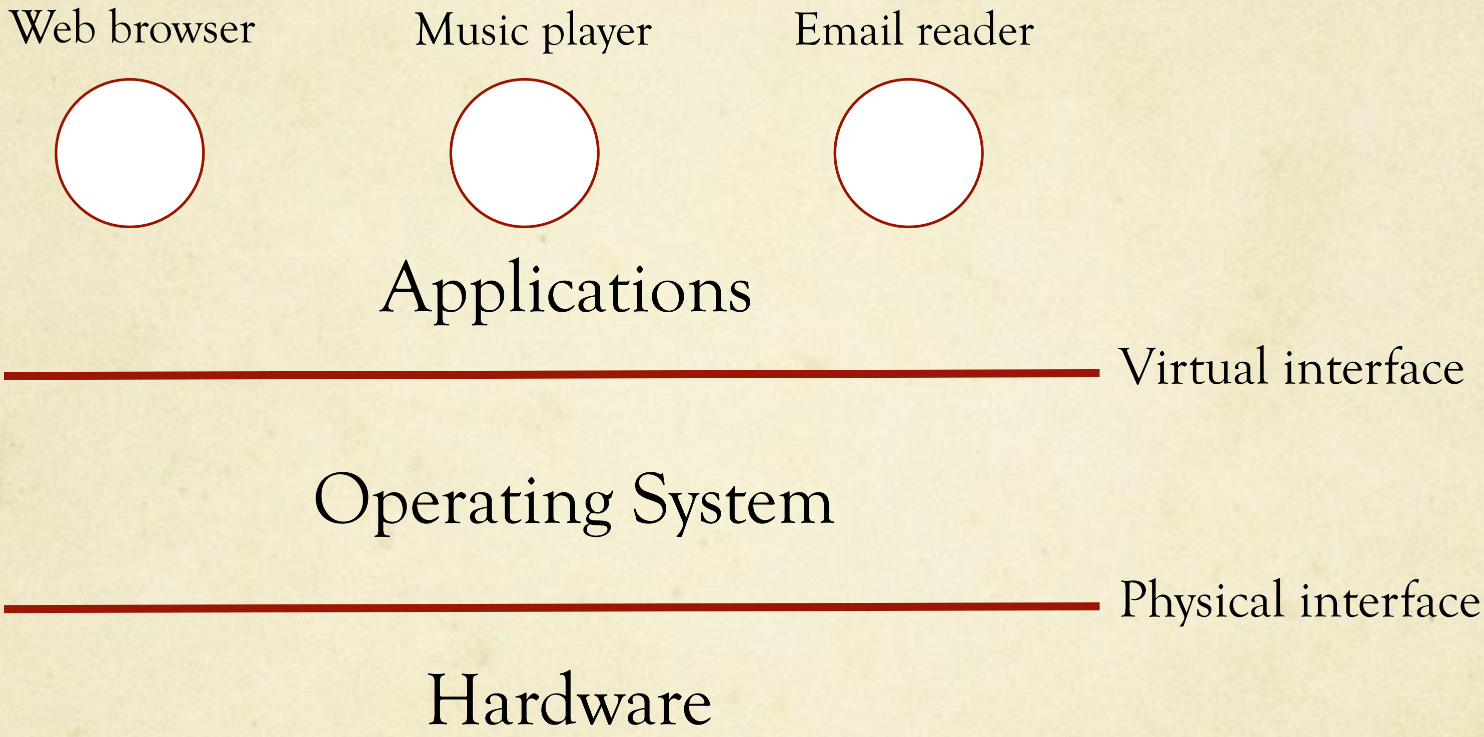
Application Software

(Need a resource manager)

Hardware

Operating System (OS)

- Software that sits b/w hardware & application (user) programs
  - Most systems have user interface layer b/w OS & applications
- Provides a *virtual interface* to underlying hardware
  - Simple interface for applications/users (hides h/w complexity)
  - Ensures safety (protects hardware, prevents & handles errors)
  - May provide multiple levels of abstraction
- Acts as a *resource manager*
  - Allows multiple applications/users to share resources
  - Ensures fair, efficient & protected access to resources



Web browser

Music player

Email reader

Applications

Virtual interface

Operating System

Physical interface

Hardware

Services provided by Operating System

- Program execution
    - Load program & data, schedule and execute program
- Memory management
    - Manage main memory; ensure programs can’t mess with other programs’ memory
- File management
    - Create, read, write files
    - Access control for files
- I/O management
    - Safe and controlled access to I/O devices

- Information maintenance
    - Get/set system time/date
- Communication services
    - Communication b/w programs
- User management
    - Authentication for access to system
- Error management
    - Detect & handle errors
- Accounting services
    - Collect statistics, monitor performance

To manage complexity…

- OS design typically separates *mechanism* from *policy*
    - I.e., separates *how* from *what/when/which*
- *Mechanism*
    - Data structures/operations used to implement abstraction/service
- *Policy*
    - Procedures/rules to guide selection of action from possible alternatives

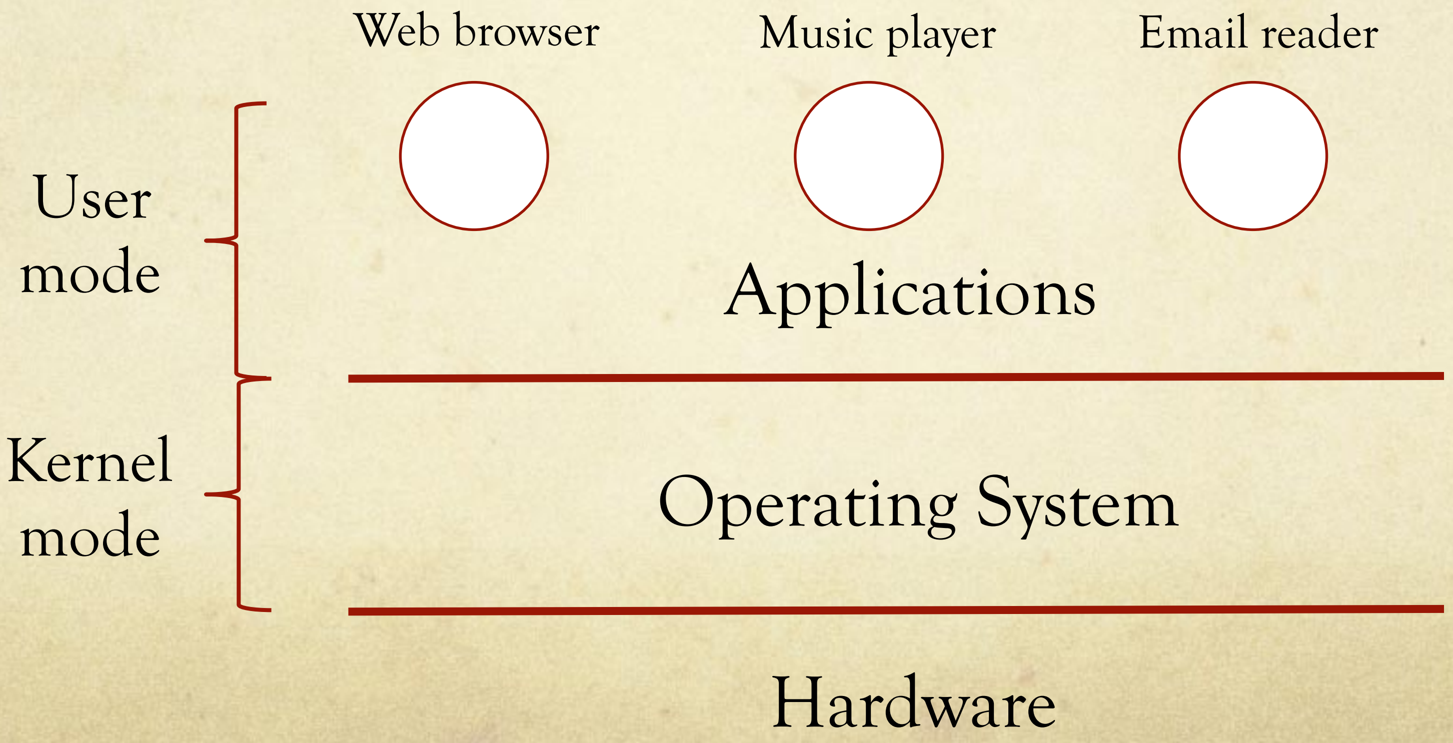
Protection

Need structures/*mechanisms* that ensure:
Protection of hardware (CPU, memory, I/O devices)
Protection between multiple applications/users

Protection

- System operation split into two *modes*
  - *User mode*
  - *Kernel mode*
- *User* mode
  - Execution on behalf of user → *protected* mode
  - No direct access to hardware
  - Can execute only *subset* of instructions
  - Can access only *restricted* memory areas

- *Kernel* (monitor/supervisor/system) mode
  - Execution on behalf of operating system → *privileged* mode
  - Complete access to hardware
  - Can execute *any* instruction
  - Can access *any* memory area



Hardware support for modes

- System maintains *mode bit* indicating current mode
- If privileged operation is attempted in user mode
  - It must be prevented from taking place
  - System must be notified
- These are achieved using an *exception*
  - *Synchronous interrupt* → caused by current instruction
- When an exception occurs
  - System enters *privileged* mode
  - Appropriate actions are taken

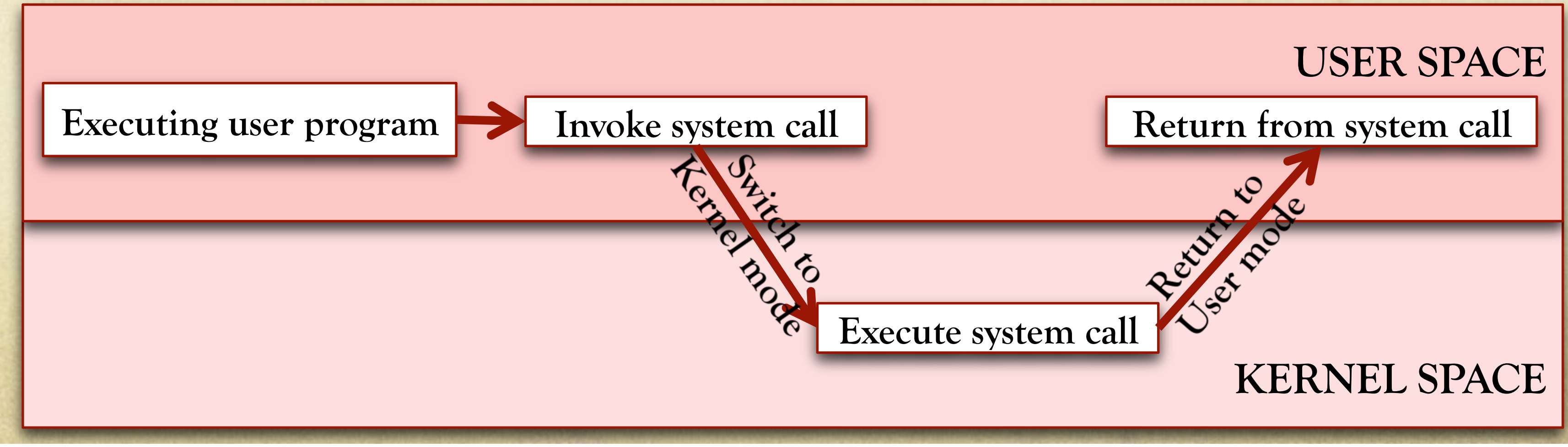
Consequence of modes

- Need special *mechanism* for applications to access OS services
- **System call** is the answer
  - *Interface* between running programs and OS
  - Provides *controlled entry* into kernel for privileged operation
  - Makes sure access is performed in specific *well defined* way

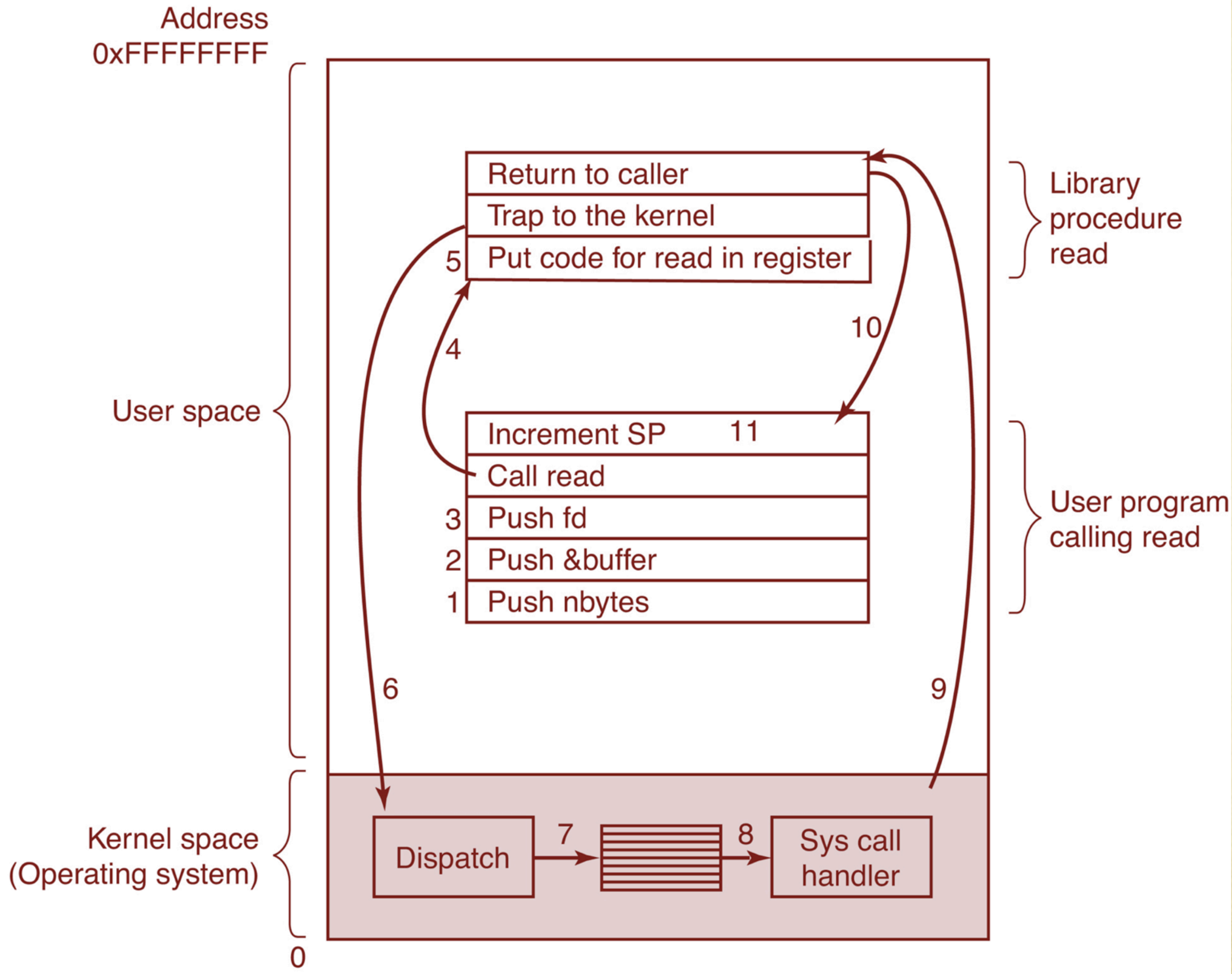
System Call

- Causes system to switch to *kernel* mode
  - → *Trap* – a kind of *synchronous interrupt* – used to achieve this
  - → **In general, any interrupt causes switch to kernel mode**

- Typically invoked using assembly language instructions
  - Systems generally provide *library* or *API* to invoke system call
  - → Library function serves as wrapper for actual system call



Example – *read* system call



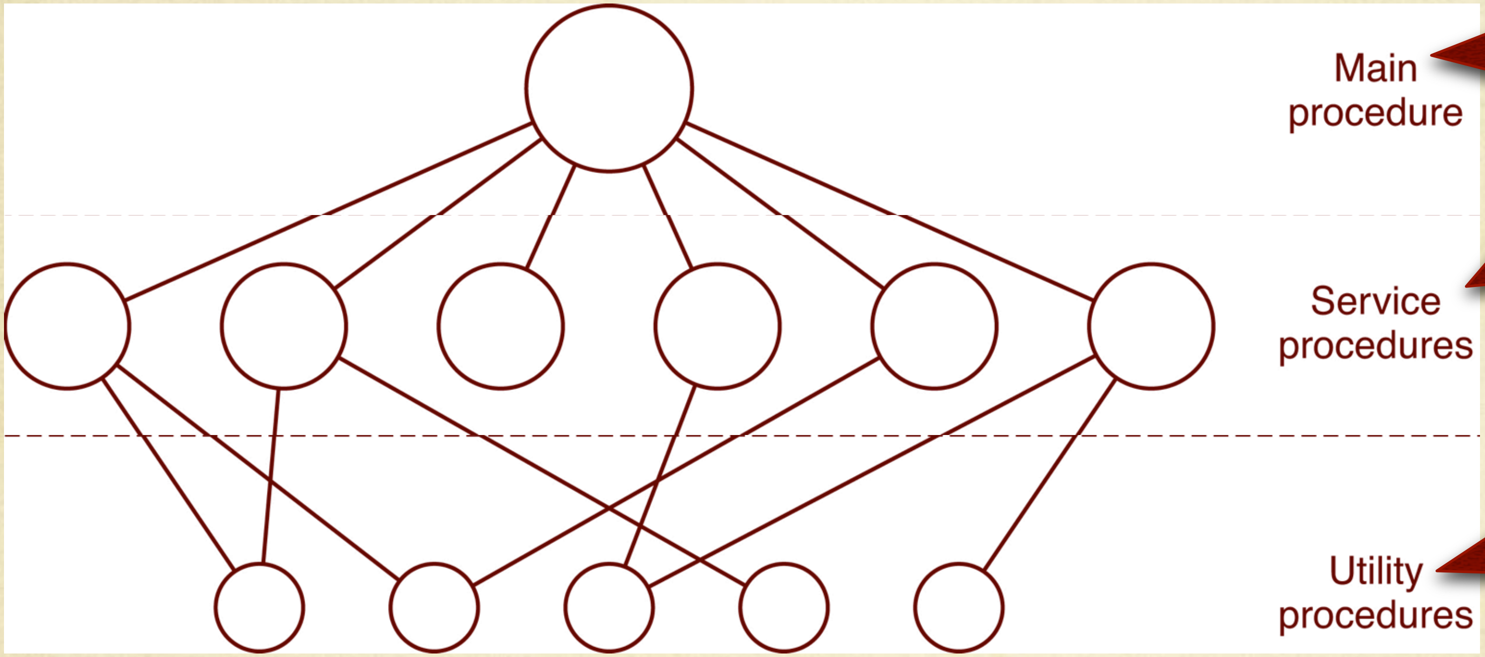
Internal structure of Operating Systems

Monolithic architecture

- Entire OS is a single program
    - Collection of procedures linked into single executable
    - Program runs fully in *kernel* mode

-
    - Sometimes called “*spaghetti nest*” approach
        - Everything tangled up with everything else

Still usually has some structure…



Entry to OS here – invokes service procs.

Carry out system calls

Help service procedures

- Examples: *Linux*, *Windows*

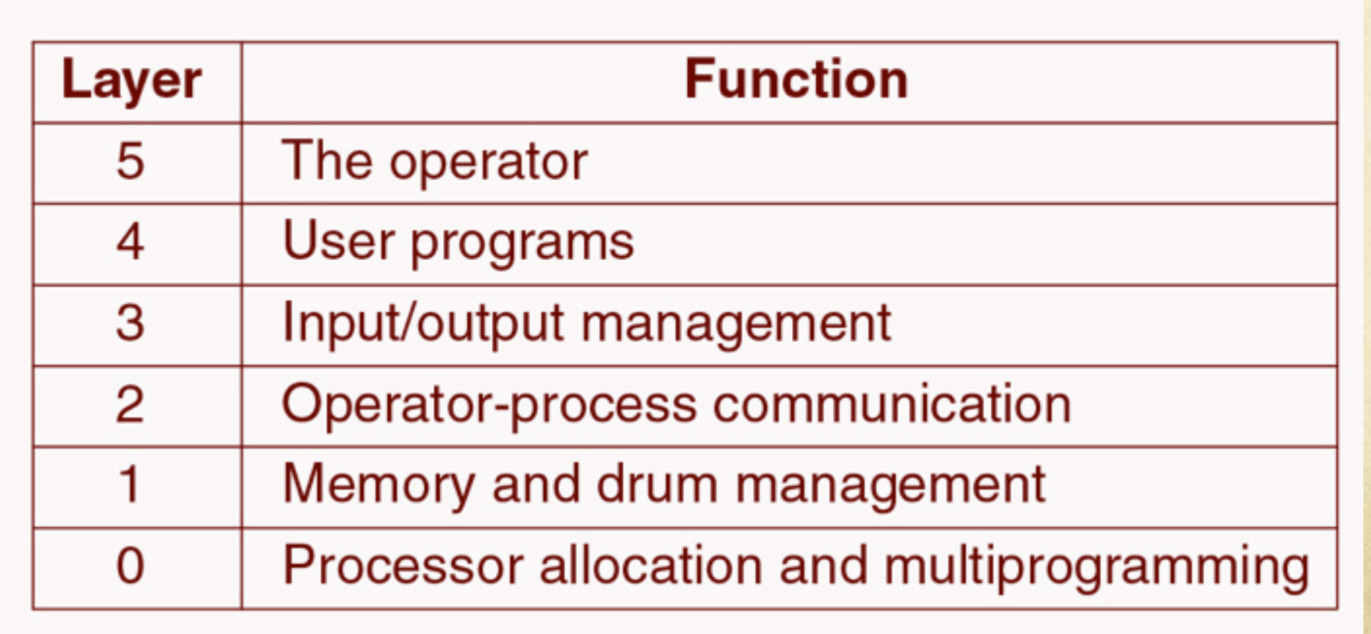
Pros & Cons

- Any procedure can call any other directly
  - → *Efficient* procedure calls
- Design, implementation, debugging etc. can be hard
- OS could become unwieldy & difficult to understand
- Error in one part of OS can bring down entire OS

Layered architecture

- Divide OS into multiple *layers*
  - Each layer responsible for certain operations/services
  - Layers independent of layers above them

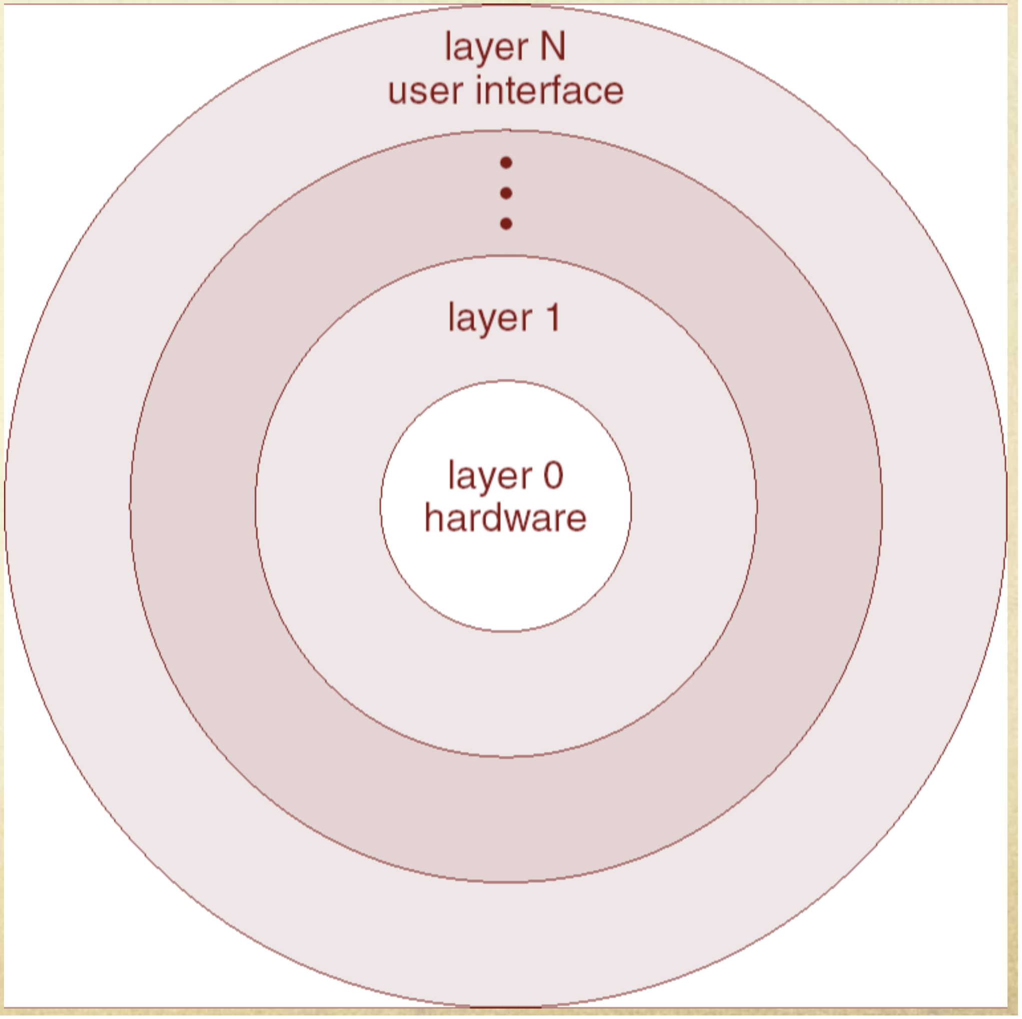
Example: *THE operating system*



Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

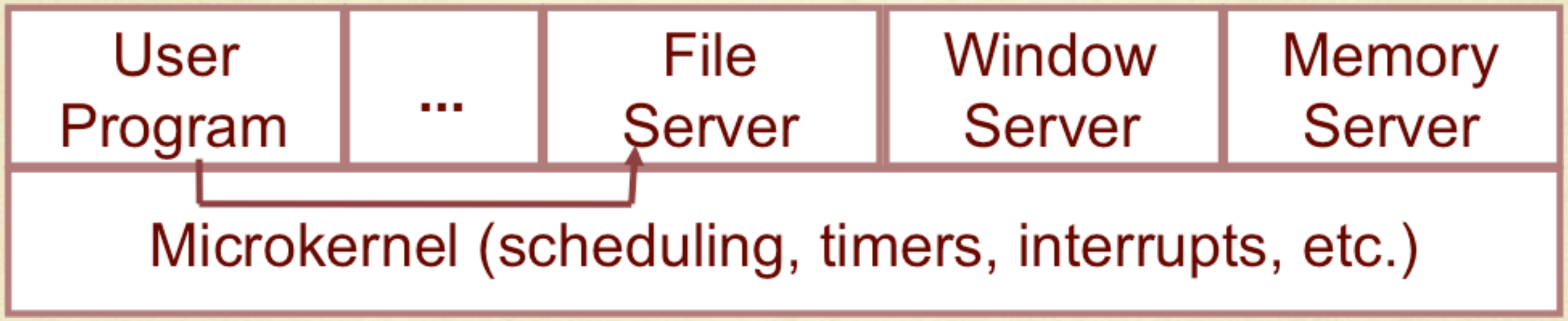
A variant of layered structure

- Similar concept, but layers represented as concentric circles
  - Inner layers have higher privilege that outer layers
- Example: *MULTICS*



Microkernel architecture

- Split OS functionality into multiple small modules
  - Core module, called *microkernel*, runs in *kernel* mode
  - All other modules run in *user* mode
  - Communication between modules using message passing



- Examples: *QNX*, *MINIX 3*

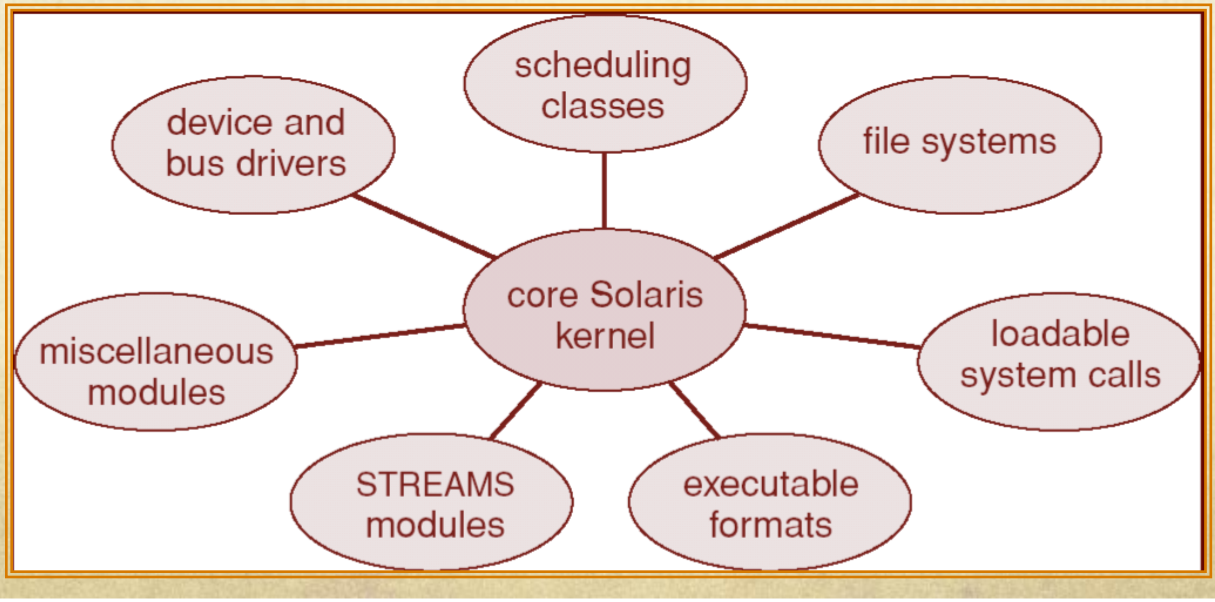
- More commonly used in embedded/real-time systems

Pros & Cons

- Easier to design, implement & debug
- More flexible & easier to extend
- More isolation of faults/errors
  - Error in one module need not bring down entire OS
- More reliable & more secure
- Significant performance overhead

Modern operating system design

- Hybrid, object-oriented approach
- Separate modules for separate functionality
- Modules loadable into kernel as needed
- Modules communicate via well defined interfaces


scheduling classes
device and bus drivers
file systems
core Solaris kernel
miscellaneous modules
loadable system calls
STREAMS modules
executable formats