Operating System objective and functions, Evolution of Operating System, Major Achievements, Development Leading to Modern Operating System, Fault Tolerance, OS Design Considerations for Multiprocessor and Multicore, Traditional Unix System, Modern Unix System, Linux.

WS 2.1-2.3 (pg.69-91),
WS 2.4-2.6 (pg.92-100),
WS 2.7- 2.10 (pg.108- 117)

OPERATING SYSTEM OBJECTIVES AND FUNCTIONS

- **Convenience**: An OS makes a computer more convenient to use
- **Efficiency**: An OS allows the computer system resources to be used in an efficient manner.
- **Ability to evolve**: An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

The Operating System as a User/Computer Interface

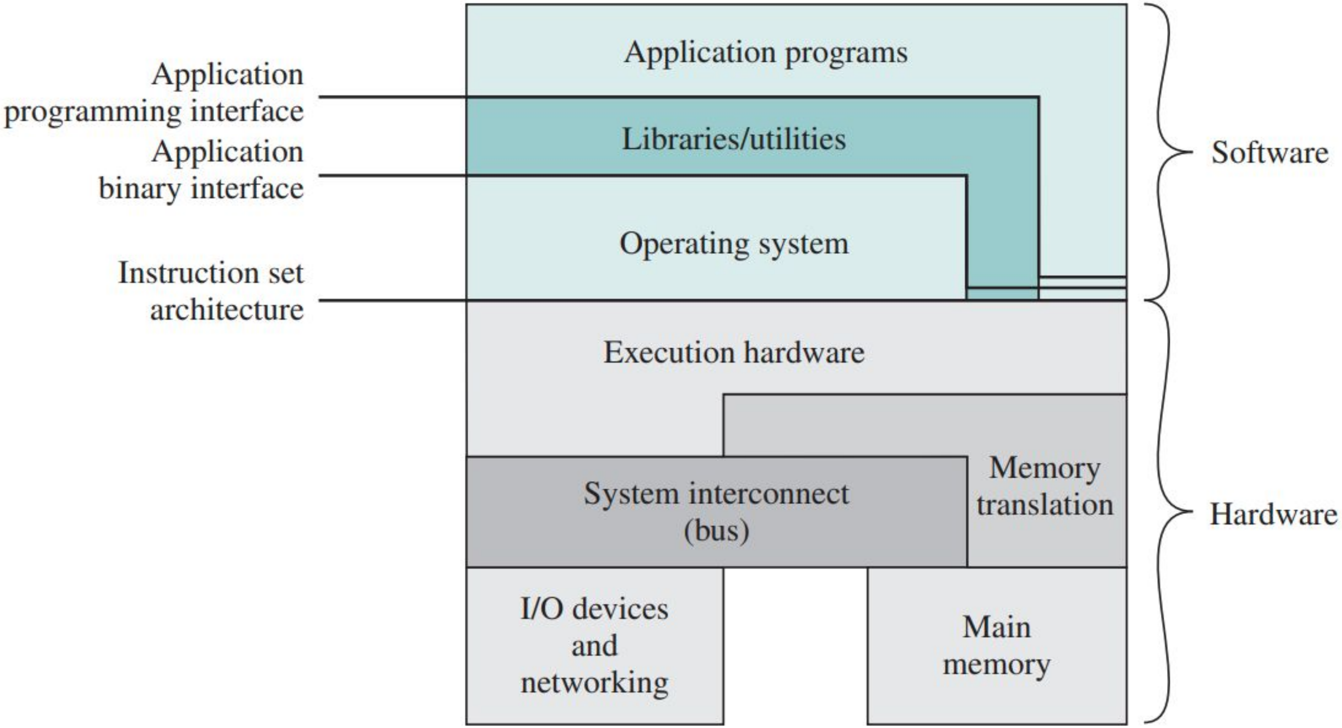

Figure 2.1 **Computer Hardware and Software Structure**

The Operating System as a User/Computer Interface

Briefly, the OS typically provides services in the following areas:

- Program development
- Program execution
- Access to I/O devices
- Controlled access to files
- System access
- Error detection and response
- Accounting
- Instruction set architecture (ISA)
- Application binary interface (ABI)
- Application programming interface (API)

The Operating System as Resource Manager

- The OS functions in the same way as ordinary computer software;
  - that is, it is a program or suite of programs executed by the processor.
- The OS frequently relinquishes control, and must depend on the processor to allow it to regain control.

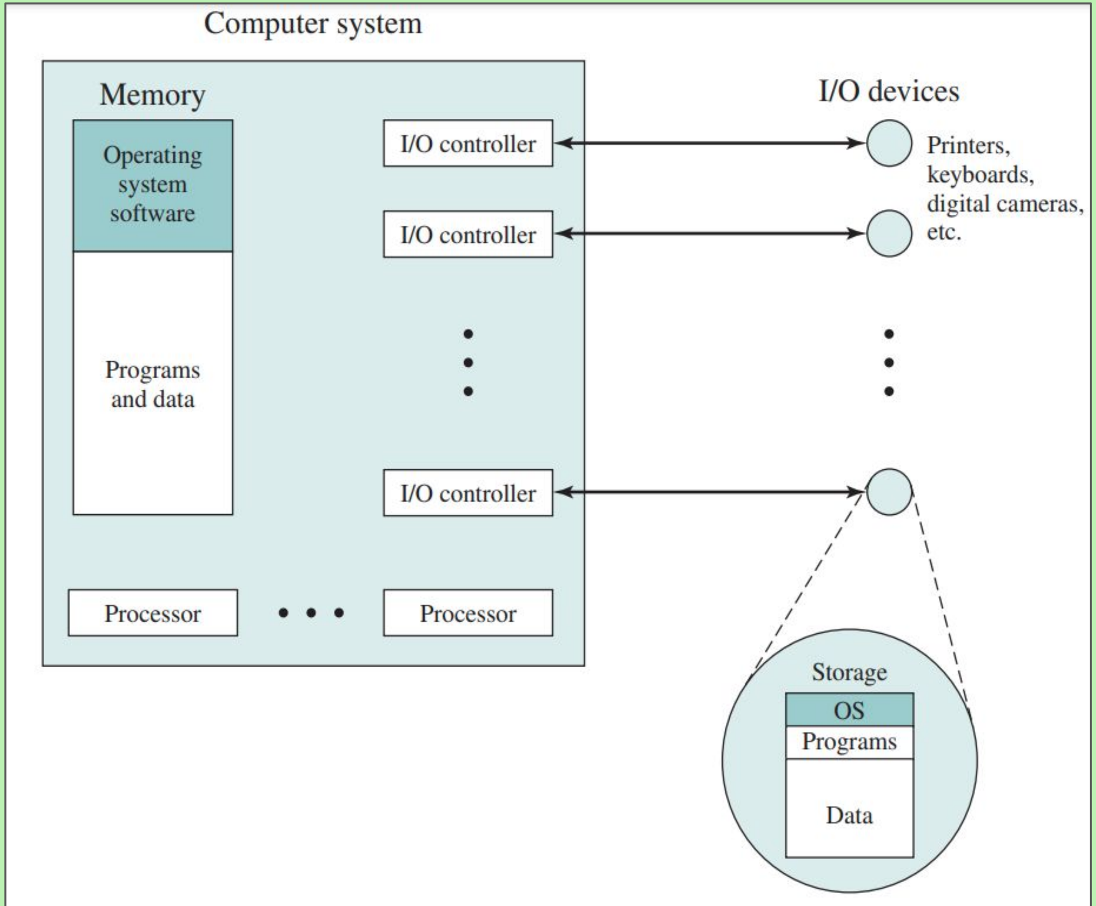

Figure 2.2 The Operating System as Resource Manager

Ease of Evolution of an Operating System

- Hardware upgrades plus new types of hardware
- New services
- Fixes

Evolution of Operating Systems

- Serial Processing
- Simple Batch Systems
- Multiprogrammed Batch Systems
- Time-Sharing Systems

Evolution of Operating Systems - Serial Processing

- late 1940s to the mid-1950s
- the programmer interacted directly with the computer hardware; there was no OS
- run from a console consisting of display lights, toggle switches, some form of input device, and a printer
- programs in machine code were loaded via the input device (e.g., a card reader).
    - If an error halted the program, the error condition was indicated by the lights
    - If the program proceeded to a normal completion, the output appeared on the printer
- two main problems:
    - Scheduling
    - Setup time

Evolution of Operating Systems - Simple Batch Systems

- The wasted time due to scheduling and setup time was unacceptable
- To improve utilization, the concept of a batch OS was developed
  - the use of a piece of software known as the monitor
- The user no longer has direct access to the processor.
  - Instead, submits the **job** (a single program) on cards or tape to an operator, who batches the jobs together sequentially and places the entire batch on an input device
- Each program is constructed to branch back to the monitor when it completes processing, at which point the monitor automatically begins loading the next program

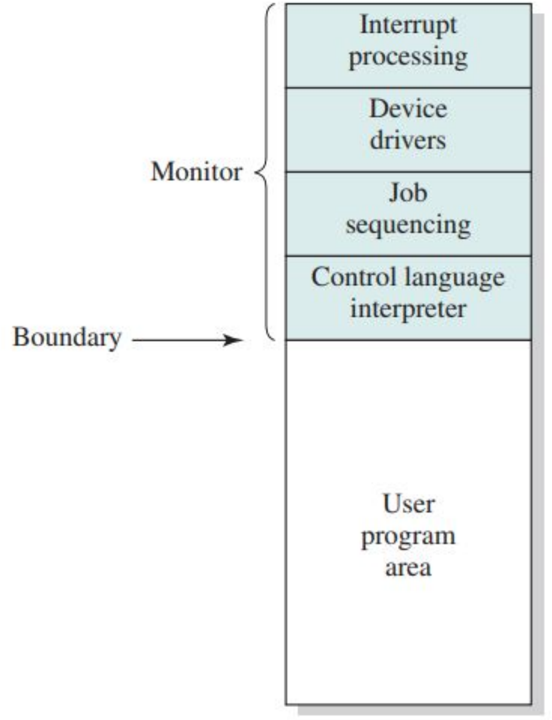

Figure 2.3 **Memory Layout for a Resident Monitor**

user mode, kernel mode

Evolution of Operating Systems - Multiprogrammed Batch Systems

- Even with the automatic job sequencing provided by a simple batch OS, the processor is often idle
    - The operator is human!

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
Total	31 μs

$$\text{Percent CPU utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

Figure 2.4 System Utilization Example

Evolution of Operating Systems - Multiprogrammed Batch Systems

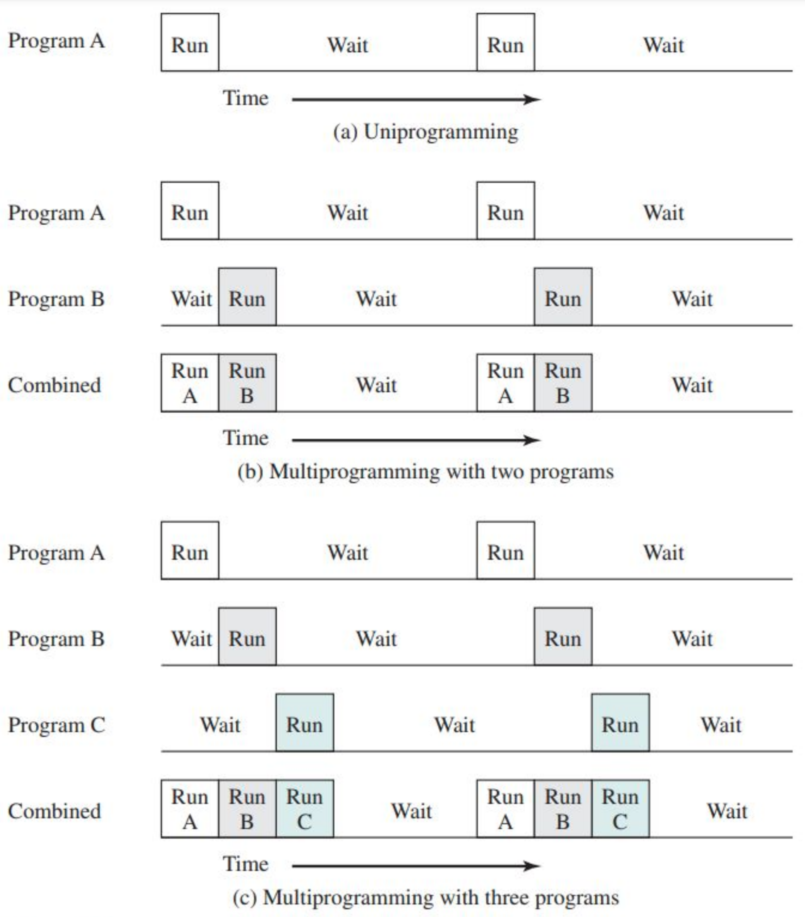

Figure 2.5 **Multiprogramming Example**

Evolution of Operating Systems - Multiprogrammed Batch Systems

Table 2.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

Evolution of Operating Systems - Multiprogrammed Batch Systems

Table 2.2 Effects of Multiprogramming on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

Evolution of Operating Systems - Multiprogrammed Batch Systems

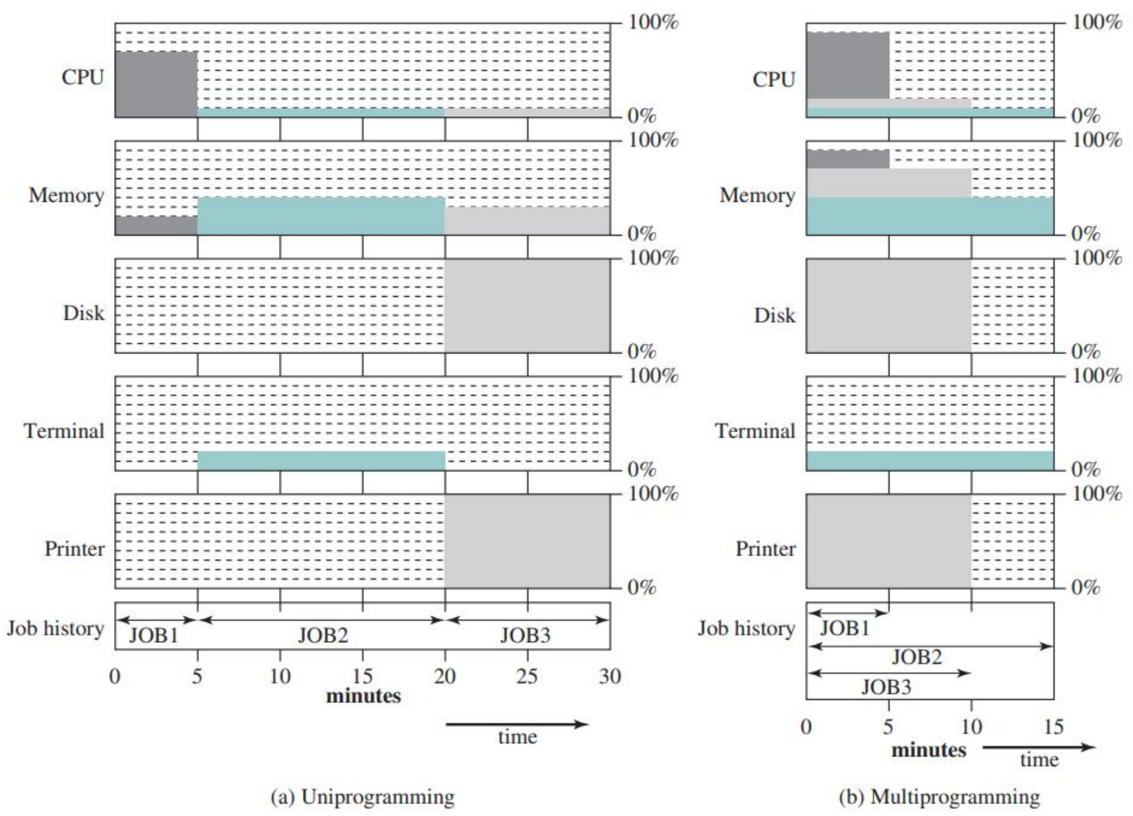

Figure 2.6 **Utilization Histograms**

Evolution of Operating Systems - Multiprogrammed Batch Systems

Suppose Three program, JOB1, JOB2 and JOB3 are submitted for execution at the same time with attribute listed below.

	JOB1	JOB2	JOB3
Type of job	Heavy compute (70% CPU used)	Heavy I/O (10% CPU used)	Heavy I/O (10% CPU used)
Duration	5 min	15 min	10 min

For simple Batch environment, these job are executed in sequence JOB1, JOB2 then JOB3. Find out CPU utilization and Throughput in case of uniprogramming and multiprogramming system

Consider a computer with 400Mbytes of available memory (not used by os). Three program, JOB1, JOB2 and JOB3 are submitted for execution at the same time with attribute listed below.

	JOB1	JOB2	JOB3
Type of job	Heavy compute (90% CPU used)	Heavy I/O (10% CPU used)	Heavy I/O (10% CPU used)
Duration	10 min	20 min	15 min
Memory required	100M	150M	125M

For simple Batch environment, these job are executed in sequence JOB1, JOB2 then JOB3. Find out CPU utilization, memory utilization and Throughput in case of uniprogramming and multiprogramming system.

Evolution of Operating Systems - Time Sharing Systems

- With the use of multiprogramming, batch processing can be quite efficient
- Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs.
  - **time sharing**, because processor time is shared among multiple users
- Both batch processing and time sharing use multiprogramming

Table 2.3 Batch Multiprogramming versus Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Evolution of Operating Systems - Time Sharing Systems

Time sharing and multiprogramming raise a host of new problems for the OS.

- If multiple jobs are in memory, then they must be protected from interfering with each other by, for example, modifying each other's data.
- With multiple interactive users, the file system must be protected so only authorized users have access to a particular file.
- The contention for resources, such as printers and mass storage devices, must be handled.

MAJOR ACHIEVEMENTS

Four major theoretical advances in the development of operating systems:

1. Processes
2. Memory management
3. Information protection and security
4. Scheduling and resource management

DEVELOPMENTS LEADING TO MODERN OPERATING SYSTEMS

Four major theoretical advances in the development of operating systems:

1. Microkernel architecture
2. Multithreading
3. Symmetric multiprocessing
4. Distributed operating systems
5. Object-oriented design

FAULT TOLERANCE

1. The reliability $R(t)$ of a system is defined as the probability of its correct operation up to time t given that the system was operating correctly at time $t = 0$.
2. Mean time to failure (**MTTF**)
3. The mean time to repair (**MTTR**) is the average time it takes to repair or replace a faulty element.
4. The **availability** of a system or service is defined as the fraction of time the system is available to service users' requests.
    a. The time during which the system is not available is called **downtime**
    b. The time during which the system is available is called uptime

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

FAULT TOLERANCE

Table 2.4 Availability Classes

Class	Availability	Annual Downtime
Continuous	1.0	0
Fault tolerant	0.99999	5 minutes
Fault resilient	0.9999	53 minutes
High availability	0.999	8.3 hours
Normal availability	0.99–0.995	44–87 hours

- The IEEE Standards Dictionary defines a fault as an **erroneous hardware or software state** resulting from component failure, operator error, physical interference from the environment, design error, program error, or data structure error.
- The standard also states that a fault **manifests** itself as (1) a defect in a hardware device or component; for example, a short circuit or broken wire, or (2) an incorrect step, process, or data definition in a computer program.

FAULT TOLERANCE

We can group faults into the following categories:

- Permanent: A fault that, after it occurs, is always present. The fault persists until the faulty component is replaced or repaired.
    - Examples include disk head crashes, software bugs, and a burnt-out communications component.
- Temporary: A fault that is not present all the time for all operating conditions. Temporary faults can be further classified as follows:
    - **Transient**: A fault that occurs only once. Examples include bit transmission errors due to an impulse noise, power supply disturbances, and radiation that alters a memory bit.
    - **Intermittent**: A fault that occurs at multiple, unpredictable times. An example of an intermittent fault is one caused by a loose connection.

FAULT TOLERANCE

In general, fault tolerance is built into a system by adding redundancy.

- Spatial (physical) redundancy
- Temporal redundancy
- Information redundancy

A number of techniques can be incorporated into OS software to support fault tolerance.

- Process isolation
- Concurrency controls
- Virtual machines
- Checkpoints and rollbacks

OS DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE

Symmetric Multiprocessor OS Considerations:

- Simultaneous concurrent processes or threads
- Scheduling
- Synchronization
- Memory management
- Reliability and fault tolerance

Multicore OS Considerations:

- Parallelism within Applications
- Virtual Machine Approach

TRADITIONAL UNIX SYSTEMS

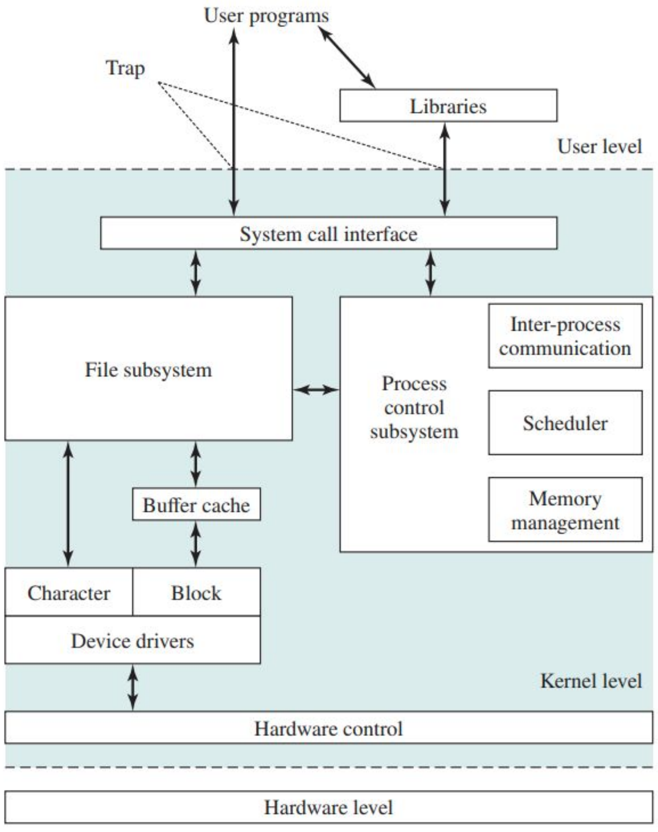

Figure 2.15 **Traditional UNIX Architecture**

MODERN UNIX SYSTEMS

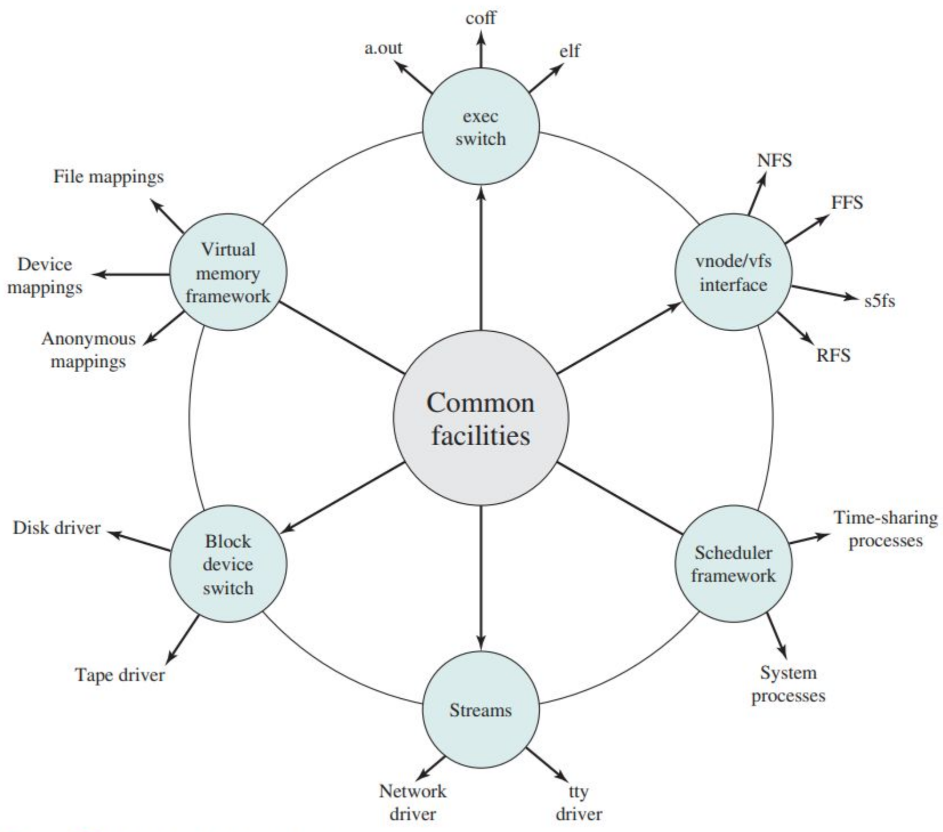

Figure 2.16 **Modern UNIX Kernel**

LINUX

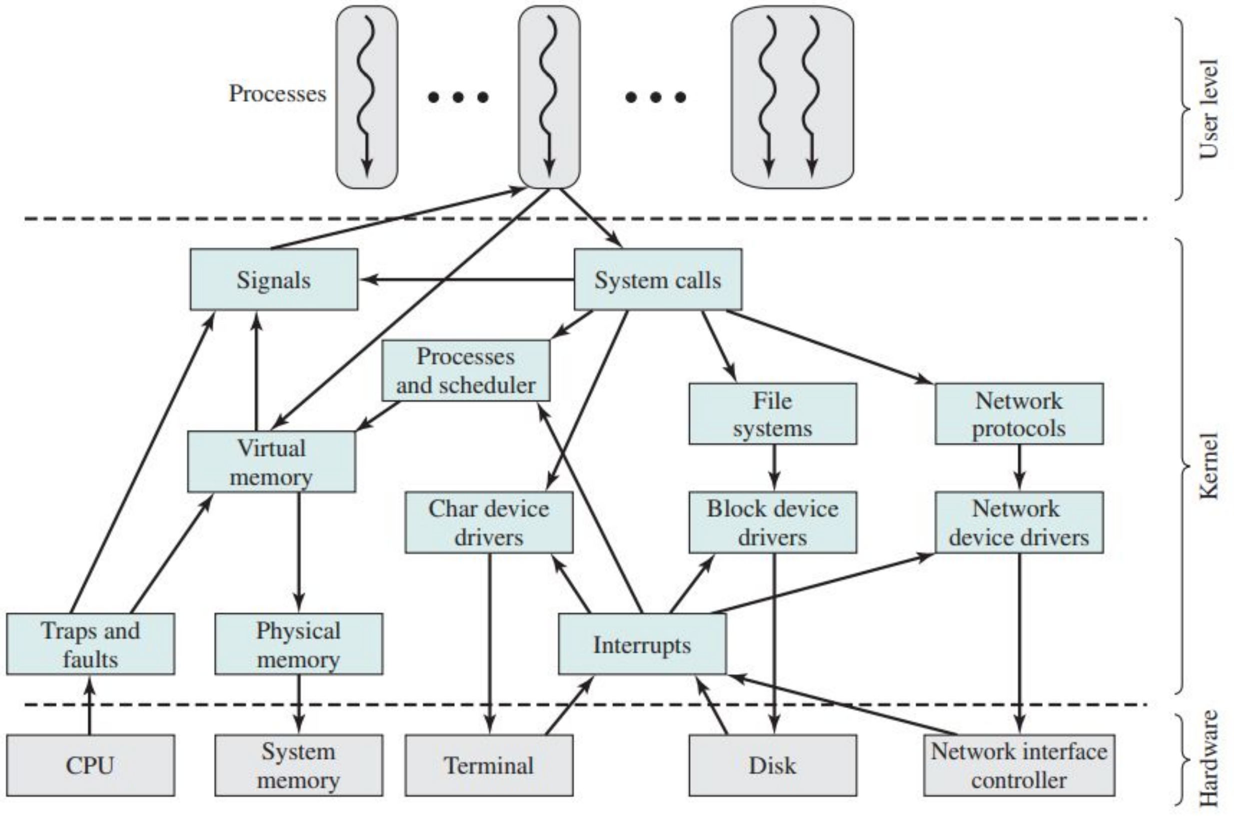

Figure 2.19 **Linux Kernel Components**

LINUX

Table 2.7 Some Linux System Calls

File System Related	
close	Close a file descriptor.
link	Make a new name for a file.
open	Open and possibly create a file or device.
read	Read from file descriptor.
write	Write to file descriptor.
Process Related	
execve	Execute program.
exit	Terminate the calling process.
getpid	Get process identification.
setuid	Set user identity of the current process.
ptrace	Provide a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers.
Scheduling Related	
sched_getparam	Set the scheduling parameters associated with the scheduling policy for the process identified by `pid`.
sched_get_priority_max	Return the maximum priority value that can be used with the scheduling algorithm identified by `policy`.
sched_setscheduler	Set both the scheduling policy (e.g., FIFO) and the associated parameters for the process `pid`.
sched_rr_get_interval	Write into the timespec structure pointed to by the parameter to the round-robin time quantum for the process `pid`.
sched_yield	A process can relinquish the processor voluntarily without blocking via this system call. The process will then be moved to the end of the queue for its static priority and a new process gets to run.

LINUX

Table 2.7 (*Continued*)

Interprocess Communication (IPC) Related	
msgrcv	A message buffer structure is allocated to receive a message. The system call then reads a message from the message queue specified by `msqid` into the newly created message buffer.
semctl	Perform the control operation specified by `cmd` on the semaphore set `semid`.
semop	Perform operations on selected members of the semaphore set `semid`.
shmat	Attach the shared memory segment identified by `semid` to the data segment of the calling process.
shmctl	Allow the user to receive information on a shared memory segment; set the owner, group, and permissions of a shared memory segment; or destroy a segment.
Socket (networking) Related	
bind	Assign the local IP address and port for a socket. Return 0 for success or −1 for error.
connect	Establish a connection between the given socket and the remote socket associated with sockaddr.
gethostname	Return local host name.
send	Send the bytes contained in buffer pointed to by *msg over the given socket.
setsockopt	Set the options on a socket.
Miscellaneous	
fsync	Copy all in-core parts of a file to disk, and wait until the device reports that all parts are on stable storage.
time	Return the time in seconds since January 1, 1970.
vhangup	Simulate a hangup on the current terminal. This call arranges for other users to have a “clean” tty at login time.

Questions

- WS Review Questions - All
- WS Problems 2.1, 2.2