Introduction to Operating Systems

# Course Overview

## Contents

## List of Figures

## List of Tables

## 1 Acknowledgments

As with any Computer Science course, Introduction to Operating Systems is constantly evolving. This evolution does not happen in a vacuum. In addition to many outside resources being consulted, we have been very fortunate to have important contributions from our TAs. Ted Cooper and Jeremiah Peschka have been particularly valuable in improving these lecture notes, as editors and contributors. Ted and Matt Spohrer developed the quiz infrastructure and questions. The quiz questions form the basis of the study questions now included with each set of notes. Ted and Jeremiah have made significant contributions and improvements to the course projects which use the xv6 teaching operating system.

As always, any omissions and/or errors are mine, notifications of such are appreciated.

## 2 Administrivia

In this class, students will study basic concepts in operating systems design and implementation. Class lectures will focus on theory and programming projects will explore translating theory into practice.

The class web page is located at `http://cs.pdx.edu/~markem/CS333` and is required reading. This course has a *survival guide* that you are required to read and follow.

In addition to lectures, there are drop-in labs, from 1:00 - 5:00 on zoom, each Saturday and also on the Sunday that assignments are due. Course staff will be present at the lab sessions to assist you in understanding course objectives and in debugging your software.

The class has a coding style which you must follow. This is documented in the survival guide. Points may be taken off for assignments with poor coding style. Turning in code that has large blocks of code commented out is poor coding style, as is inconsistent indentation, and failure to abstract duplicate code into functions, etc. We won't be overly picky, but you should have a good coding style by this point in your degree program.

For grading, each project includes a rubric that is available to you. The rubric outlines points for different aspects of the project code and report. Project grades can be challenged for 2 weeks after the scores and feedback are made available to students. The class web site lists the process for challenging a score.

### 2.1 Academic Honesty

Students are expected to understand the concept of academic integrity and the ramifications of plagiarism, cheating, etc. Contact student services or your advisor if you are unclear on these concepts. Any work submitted by a student wherein substantial parts of the work are not those of the student or a misrepresentation of one student's work as that of another are examples of academic dishonesty. Academic dishonesty will result in a minimum penalty of the loss of all points for that project or exam. All incidents of academic dishonesty will be reported to the office of the Dean of Student Life for investigation.

### 2.2 Getting Help

See the survival guide for a listing of resources. The survival guide is **required reading**.

Lab sessions are a good place to get help. All lab assistants have completed the course with high marks. For complex or difficult to debug issues, sending your code to the TAs and instructor is best, as they will be able to compile and debug your code themselves. Sending code snippets, descriptions of what you think *might* be happening, and screen shots are not useful unless accompanied by your code. *We only want you to send us necessary source code.* Follow the directions in Chapter 2 of the Survival Guide to create a tar archive that you can then email to course staff for assistance. Failure to follow all the steps outlined there could result in substantial delays in getting assistance from course staff.

## 2.3 Submitting Your Work for Grading

Each project will have two items submitted for grading via D2L:

1. The project report. The report is required to be in PDF format. Failure to submit in PDF format will result in a score of zero for the report. This is not subject to negotiation.

2. Project code. Follow the instructions in Chapter 2 of the Survival Guide. Submitting incomplete work because you did not follow the submission instructions will result in zero points for that part of the project. This is not subject to negotiation.

## 3 Learning Objectives

1. Describe the basic components of a computer system.
2. Describe the differences between SMP and DSM (NUMA) architectures in terms of memory structures.
3. Explain the rationale for the principle processor modes.
4. Explain how isolation and protection are achieved by the ISA.
5. Explain the instruction execution cycle used in this class.
6. Define and give an example of each term
    - Exception
    - Trap
    - Interrupt
    - Fault / Abort
7. Explain the role of the trap in system call design.
8. Explain how single interrupts are handled by both hardware and software.
9. Explain how nested interrupts are handled.
10. Explain the principle benefits and costs of using cache memory.
11. Explain the different types of locality, giving an illustrative pseudo-code example for each.
12. Explain cache coherency.
13. Explain serializability.
14. Explain the principle types of virtualization that will be covered in this class.
15. List the key abstractions that will be covered in this class.
16. Explain the difference between parallelism and concurrency.
17. Explain the difference between a program and a process.
18. Explain the principle roles played by the operating system.
19. Explain the principle benefits of hardware virtualization.
20. Explain what is meant by the term containerization.
21. Explain the impetus for multiprogramming.

## 4 More Than Operating Systems

This course builds on, or prepares you for, these courses

- Computer Systems Programming (CS201)
- Data Structures (CS163)
- Discrete Structures II (CS251)
- Theory of Computation (CS311)
- Algorithms and Complexity (CS350)
- Software Engineering (CS300 and CS454)
- Computer Architecture (ECE341)
- Introduction to Database Management Systems (CS486)
- Database Management System Implementation (CS487P)
- IoT Development in Rust (CS410)
- Concurrent and Parallel Programming (CS415P)

## 5 What is a Computer?

This can be a surprisingly hard question to answer. Your laptop is certainly a computer and so are most phones. Is a printer? Is a hearing aid? What about a car? I have a light switch in my house that tracks sunset and turns on the outside lights when it gets dark – it even accounts for Daylight Savings Time! Are these computers? What about "the cloud"?

One last philosophical question: is the Internet a computer (computational device) or merely a passive mechanism for moving data from point-to-point (communications channel)?

### 5.1 Hardware or Hardware Plus Software

Modern systems are quite complex. They are much more than a processor, memory, and storage. How many radios does your phone have? wifi, Bluetooth, cellular. Each of these have different requirements for frequency range and antenna design. Similarly, how many processors does your phone have? How many cores? Is "core" equivalent to "CPU"[1] or "processor"?

BTW, how are those cores constructed? As a symmetric multiprocessor (SMP) or maybe a distributed shared memory multiprocessor (DSM)? Did you know that NUMA, non-uniform memory access, is another term for DSM? Modern cores have private caches, shared caches for groups of cores and then interconnects that connect more groups of cores and main memory. For NUMA architectures, each core typically has a local memory separate from the cache.

Is the BIOS operating system software? Device firmware? What about code on embedded controllers? It is getting hard to say, anymore. Did you know that some network interfaces run Linux and so do many home routers?

[1]CPU appears to be a dying term. However, I am old, so I reserve the right to use it on occasion. Feel free to call me on it.

Intuitively, the operating system is distinct from the hardware since the same hardware can run many different operating systems. Think of a dual boot Linux/Win10 machine. We will use this intuitive approach to simplify our view of an operating system. After all, this is an introductory course.

## 5.2 Computer System Architecture

For the purposes of this class, the terms "CPU", "processor", and "core" will be used interchangeably. Where a difference is key, the context will be made clear, to avoid misunderstandings.

A key idea that all programmers must remember is that nothing happens within a computer instantaneously. Even light takes time to travel over the smallest of distances. There is always a *cost* and cost is often measured in terms of *time*, or *latency*. See Figure 5.

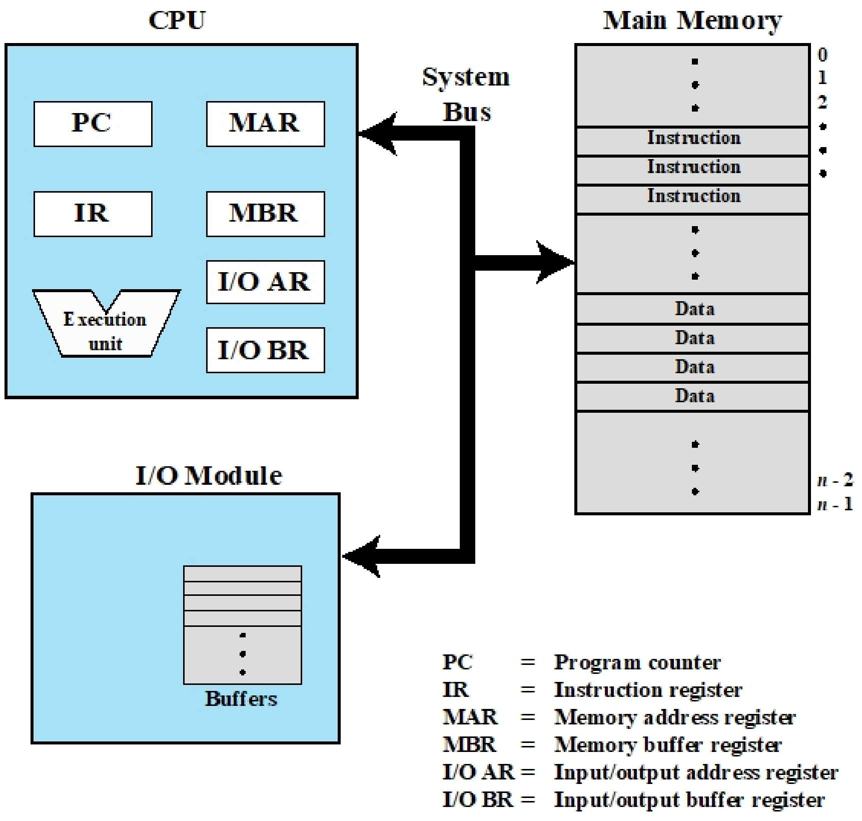

Figure 1: Primary Hardware Components

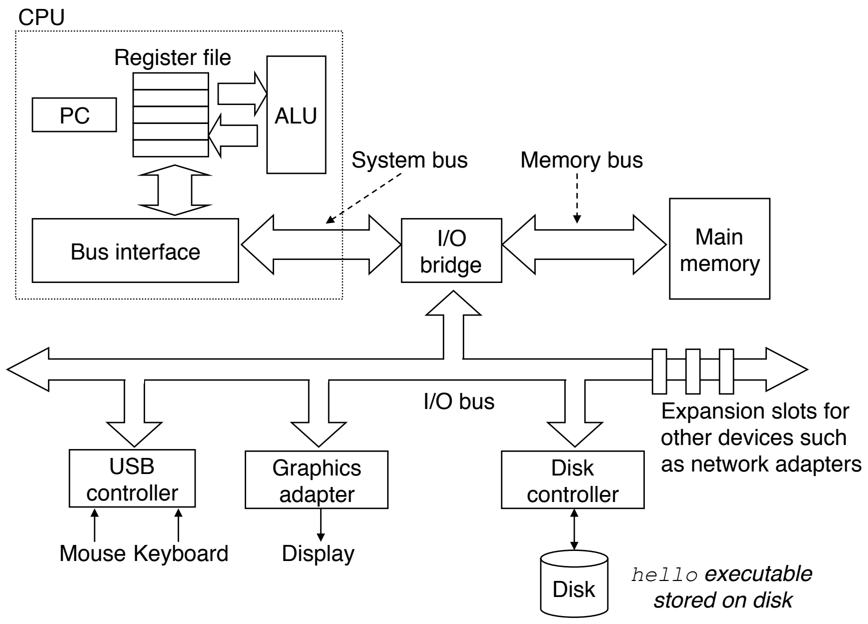

Figure 2: High-level Hardware Model

Computer systems are comprised of discrete components coupled via communication paths (buses, for the most part). In Figure 2, the hollow arrows represent *buses*. The time it takes to communicate with a device is directly proportional to the physical length of the bus. *Bridges* and *controllers* are interconnects that allow buses to be *chained* (linked together) in a hierarchy. That is, one bus can be said to "hang off" another bus through a controller that mediates the interface. *Controllers* can be simple devices or, more commonly, specialized processors which may have their own memory, storage, etc.

## 5.3 Processor Architecture / Evolution

It is rare, these days, for a computer to have only a single processor or core. Multi-core appears to be the future of processor design[2]. Alas, there is more than one type of multi-core design. Don't forget that our principle cost is time. The time it takes to complete an operation / access is called its *latency*.

In an SMP design, the latency for any memory access beyond the local cache is the same for all cores. See Fig. 3.

[2]See Moore's Law for insights. For the trivia minded, this is the same Gordon Moore whose name is on the atrium of the Engineering Building. There is some evidence that gains via Moore's Law may be over; see this article.

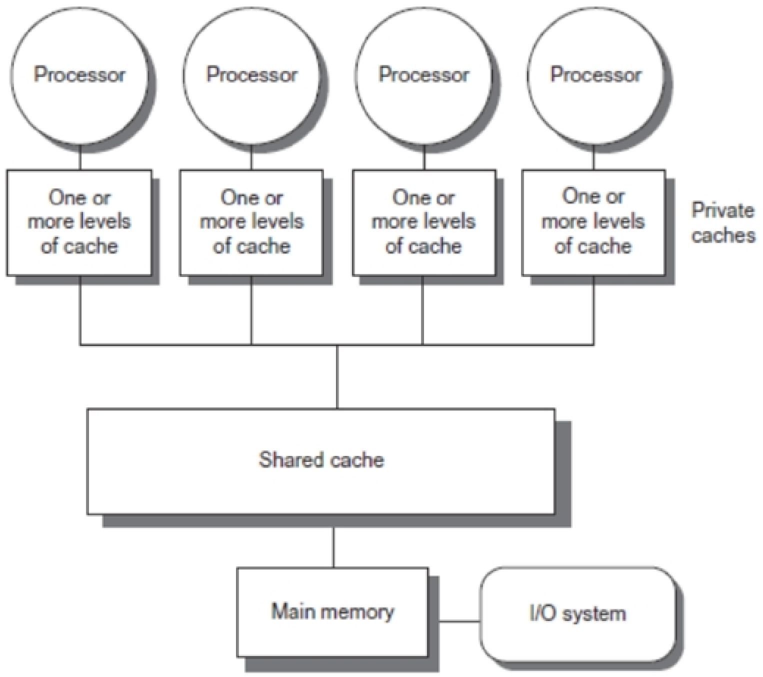
Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

Figure 3: SMP Processor

In a DSM design, each core, or group of cores, can have a local memory in addition to local caches. This means that access to memory will be *non-uniform* where the latency will depend on which remote memory is being accessed. Such a memory access model is called *NUMA* for "non-uniform memory access". See Figs. 4 and 5.

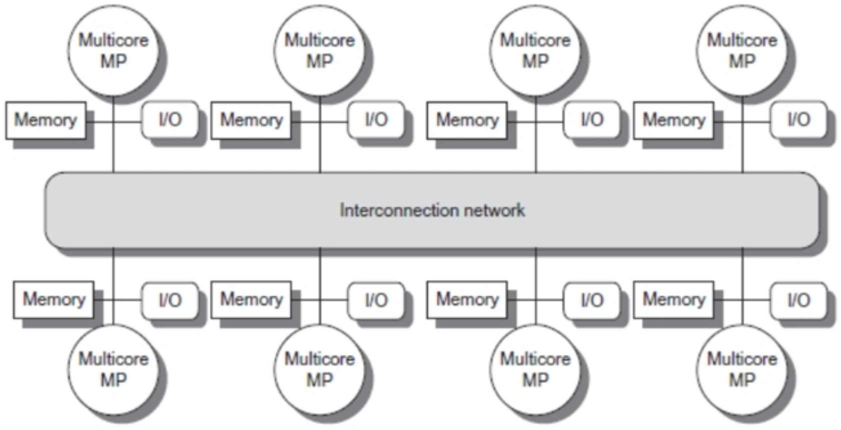

Figure 4: NUMA Architecture

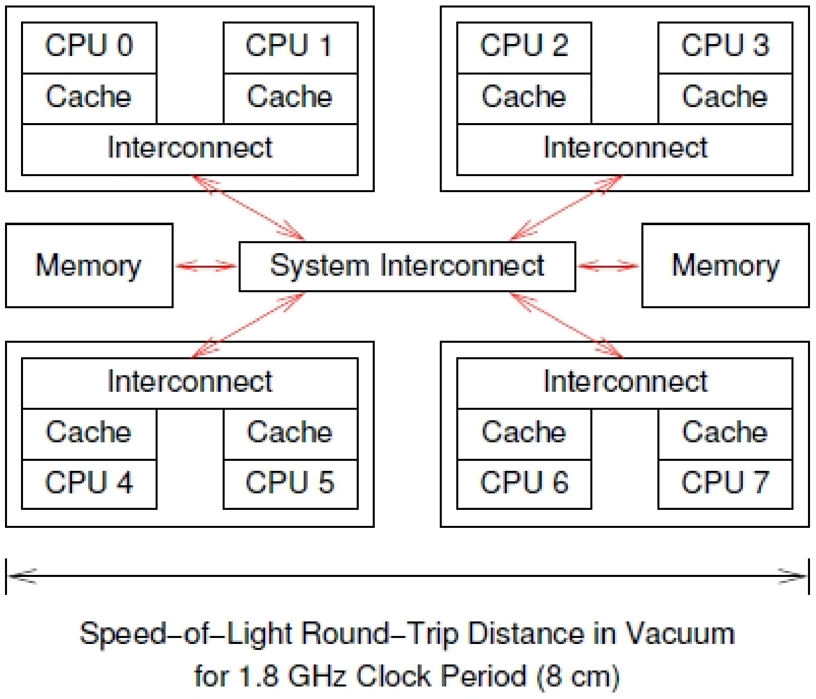

Figure 5: ,
Another View NUMA Architecture Another View

### 5.4 Modes

The operating system requires hardware assistance since when a user process is actually running in the processor, it is the instructions of the user program, not the kernel, that are executed. User processes should generally not be able to directly influence or modify other processes or the operating system. Many processor instructions can have wide-ranging implications when executed and so are naturally restricted to only the most trusted of software, e.g., the kernel. This restriction can be done using *modes*. Additionally, to get back control of the processor at regular intervals, most processors support *interrupts*, which cause code designated at system initialization to run. The main tool used for periodically granting the operating system control of the processor is the *timer interrupt*.

- Processor modes - what are they and why do they exist[3]? Figure 6 shows the Intel ring model for processor modes.

  ▷ Kernel mode. In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware. It can execute any processor instruction and reference any memory address. Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. Crashes in kernel mode are catastrophic; they will halt the entire PC.

  ▷ User mode. In User mode, the executing code has no ability to *directly* access hardware or physical memory. Code running in user mode must switch to kernel mode to access hardware or memory. This is typically done via a system call (ia an interrupt on IA32) or an internal OS process (VM translation). Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable.

  This class will not discuss the implementation of modes or the specifics of how a processor is configured for a *mode switch*. Instead, we will use terms such as "enter kernel mode", "switch to kernel mode", "switch to user mode", and "return to user mode".

- Privileged instructions. Privileged instructions are sometimes called *protected operations*.

  ▷ A machine code instruction that may only be executed when the processor is running in kernel, or supervisor, mode. Privileged instructions include operations such as I/O and memory management.

  ▷ A class of instructions, usually including storage protection setting, interrupt handling, timer control, input/output, and special processor status-setting instructions, that can be executed only when the computer is in a special privileged mode that is generally available to an operating system, or executive, but not to user programs.

  Note to students: these two bullets can be interpreted as being different explanations for the same basic idea.

- Isolation and protection. By using modes, the system is able to *isolate* certain instructions to use only in the operating system. Through this isolation, the system provides *protection* against malicious use of privileged instructions. This is one example of isolation and protection, but as you'll see, these concepts apply beyond the execution of instructions.

[3]See Coding Horror

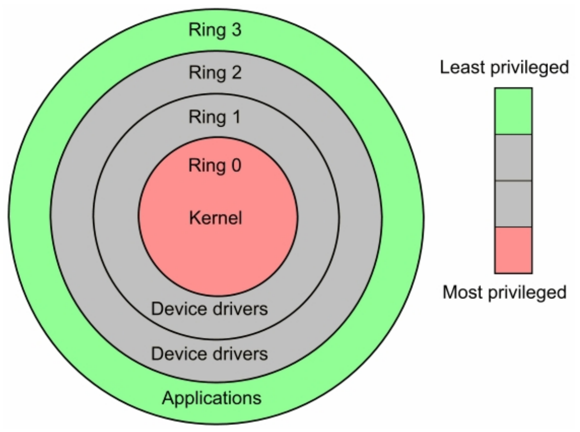

Figure 6: Intel Ring Model of CPU Modes

What, you mean after all this there are more modes that don't relate to kernel and user? For the difference between real and protected modes for Intel processors, see this article from sodocumentation.net.

## 5.5 Processor Types

1. Microprocessor. Fastest general purpose processor. May be organized as one core per chip or multiple cores per chip. Systems may also have more than one chip (single or multi-core).
2. Graphical Processing Units (GPUs). Provide efficient computation on arrays of data using Single-Instruction Multiple Data (SIMD) techniques. No longer used just for rendering advanced graphics. Also used for
   (a) General numerical processing
   (b) Games rendering and computations
   (c) Computations on large data sets
   (d) Cryptographic computations
3. Digital Signal Processors (DSPs).
   (a) Commonly used in embedded devices such as modems and radios
   (b) Becoming more important for the hand-held market
   (c) Encoding and decoding speech and video (codecs)
   (d) Various uses in encryption and security

## 5.6 Instruction Execution

Instruction execution is multi-step (see Figure 7). It is important to note that this *instruction cycle* is implemented in hardware and from the standpoint of concurrency, cannot be interrupted, and is therefore *atomic*. This is because the last step checks for and executes pending interrupts, and they are not processed during the other steps. This means that each execution of the four steps is atomic from the standpoint of concurrency. This becomes very important later.

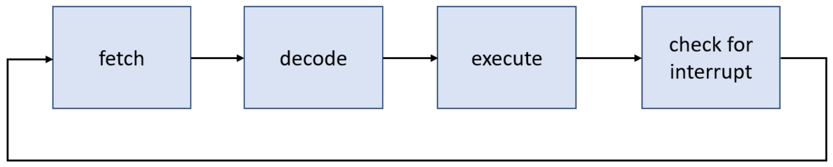

Figure 7: CPU Execution Cycle

| | |
|---|---|
| Fetch | The instruction pointer (IP), also called the program counter (PC), has the address in main memory for the next instruction to be executed. This instruction must be read from main memory and placed where the processor can find it – typically in an instruction register (IR) that is invisible outside of the processor. Note that this means that there is *at least one memory reference per assembly instruction*. Note that the presence of one or more caches may mean that the address reference can be satisfied without the latency of a main memory fetch. |
| Decode | Within a processor, there is a *micro-architecture* that executes the instruction. The decode phase identifies these components and may also cause data (instruction arguments) to be fetched from main memory. The same caveat as for "fetch" applies here. |
| Execute | The instruction is executed and the result stored in a register or main memory location (for a store). Additionally, for some instructions, *condition codes* are set. The same caveat as for "fetch" applies here. |
| Check for Interrupt | Determine if the processor must suspend execution of the current process in order to respond to some event outside of the control of the process. |

Table 1: Steps of the Instruction Execution Cycle

### 5.7 Interrupts, Faults, Exceptions

According to Bryant and O'Hallaron[4], exceptions allow a system to respond to changes in system state that are not captured by internal program variables and are not necessarily related to the execution of the current process. The term *exceptional flow control* is used as a category for many concepts, including exceptions, which they further break down into *interrupts*, *traps*, *faults*, and *aborts*.

[4]See Chapter 8 of *Computer Systems: A Programmer's Perspective*, 3rd ed., Bryant and O'Hallaron, Prentice Hall, 2015.

| | |
|---|---|
| Exception | An abrupt change in the control flow in response to a change in the state of the processor. When noticed by the processor an *exception handler* is *dispatched* to handle the specific exception type through the use of an *interrupt dispatch table* (IDT), sometimes called an *interrupt vector table* (IVT)[5]. The operating system configures the system at boot time to execute specific code in response to each exception, or interrupt, number. |
| Interrupt | An interrupt is an *asynchronous* notification to the processor, often by I/O devices. Using the interrupt number, an *interrupt handler* is dispatched to handle the event[6]. |
| Trap | Traps are a *synchronous* notification to the processor. They are deliberately caused by process execution. The most common type of trap is the *system call*[7]. |
| Fault / Abort | A fault may or may not be recoverable[8]. A fault that cannot be recovered from is termed an abort[9]. |

Table 2: Types of Interrupts

Interrupts provide a clean mechanism by which other system components (I/O, memory) may interrupt the regular sequence of instruction execution of the processor. The principle purpose of interrupts is to provide a mechanism for improved processor utilization[9].

Interrupts can be placed into a relatively few categories based on the entity that caused[10] the interrupt to occur.

| | |
|---|---|
| **Program** | Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's *allowed* address space (memory region). |
| **Timer** | Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis. |
| **I/O** | Generated by an I/O controller (hardware), to signal normal completion of an operation or to signal a variety of error conditions. |
| **Hardware Failure** | Generated by a failure, such as power failure or memory parity error. |

Table 3: Interrupt Categories

[5]Intel has used both terms over time to mean subtly different things. RTRM– read the reference manual.

[6]The *timer interrupt* will be discussed at length, especially as to how it enables the *context switch* process. Later you will learn about the very useful *page fault* and how it improves memory utilization.

[7]Much of the course work with xv6 will involve writing new system calls. For xv6, the mode is changed to kernel using the Intel *int* instruction is the file `usys.S` and the mode it changed back to user with the Intel instruction *iret* in `trapasm.S`. The *function dispatch table* for system calls for xv6 is located in `syscall.c`. A function dispatch table is an array of *function pointers*, as described in CS 201.

[8]We will discuss *page faults* at length later in the term.

[9]We will see more on this idea when we discuss scheduling.

[10]There are many ways to cause an interrupt to occur (post), including special instructions, hardware issues, invalid memory access, etc.

### 5.7.1 Single Interrupts

An interrupt suspends the executing process and causes other code to be executed[11]. This is called *handling the interrupt* and the code responding to the interrupt is called the *interrupt handler*[12]. To accommodate (seemingly) immediate processing of an interrupt, the instruction cycle for the processor has an added stage, *check for interrupt*. The processor and operating system cooperate to ensure that the interrupted process cannot tell that an interrupt has occurred and will be able to resume processing as though the interrupt never happened.

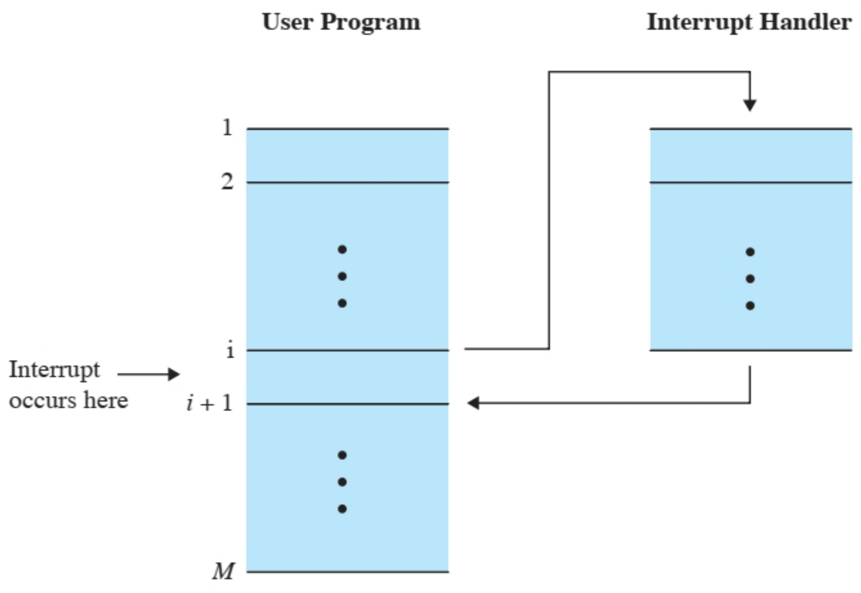

Figure 8: Transfer of Control via Interrupts

Processing for interrupts is quite complex[13]. It requires the cooperation of both hardware and kernel software. Figure 8 shows a simple example.

[11]This requires complex, processor-specific, setup at boot time. We will not cover boot issues in this class.
[12]Also called the *interrupt service routine* (ISR).
[13]See Wikipedia.

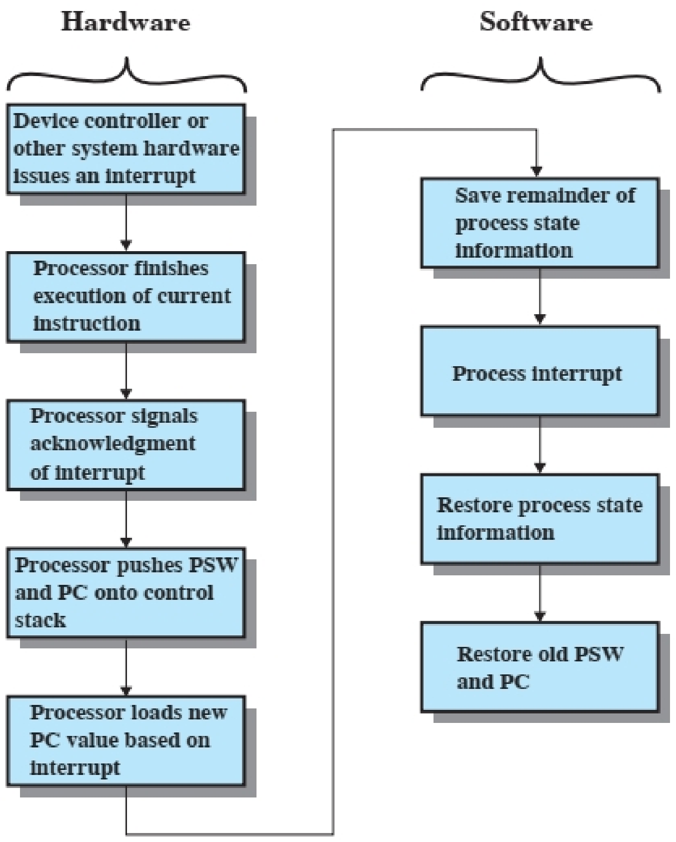
Hardware
Software
Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC

Figure 9: Simple Interrupt Processing

Figure 9 shows the steps for processing an interrupt. This is a simple example and a much more complex approach is often required[14]. The processing steps are

1. The device posts (issues) an interrupt signal to the processor.
2. The processor finishes executing the current instruction and then checks for an interrupt. Note that the currently executing instruction is not suspended, but allowed to complete.
3. The processor sends an acknowledgment to the device that posted the interrupt. This may or may not allow the device to continue processing, independent of the processor. However, many modern systems require that an *end-of-interrupt* (EOI) message[15] be sent to the device (or a specialized interrupt processor) at the end of interrupt processing.
4. The processor prepares to transfer control to the interrupt service routine. Much of this preparation involves creating a special interrupt stack frame where key processor state is stored.
5. The processor then transfers control of the processor to the memory location where the interrupt handler is located. This is conceptually similar to a function call. From this point forward, interrupt processing is completed via software.
6. The operating system must now save enough process state[16] for the currently executing process to be resumed as though no interrupt occurred.
7. The interrupt handler may now process the interrupt! If necessary, an EOI message will be generated at the end of this step.
8. The context of the interrupted process may now be restored.
9. The contents of the interrupt stack frame are now restored.

### 5.7.2 Multiple Interrupts

There are two basic approaches to dealing with multiple interrupts; that is, dealing with interrupt processing that is then interrupted by an interrupt[17]. This is termed "nested interrupts".

- Disable interrupts. To disable an interrupt is to ignore any new interrupt signal until interrupt processing is explicitly turned back on. The interrupt is typically not lost and remains *pending* until a handler processes the interrupt.
- Use priorities. All possible interrupts are numbered and the numbering constitutes a priority. An interrupt with a higher priority (however that may be implemented) can interrupt the currently executing interrupt handler (also called an *interrupt service routine (ISR)*). An interrupt at the same[18] or lower priority is *delayed*, or left pending, until its turn to be executed.

[14]One of the reasons that graduate school exists.
[15]The xv6 operating system uses EOI messaging for interrupt processing.
[16]We will learn more about saving a process context when we cover scheduling.
[17]This can get quite complex and is not studied in depth in this course.
[18]We would, of course, prefer a total ordering, but that is not required. A total ordering is much easier to process as, at some point, tasks within the same priority set will have to be (perhaps arbitrarily) ordered for execution. This can be very difficult to get correct.

There is a maximum number of interrupts that can be pending (queued) and any additional interrupts may be lost. It is also possible that some interrupts are not queued, due to design considerations. Such interrupts are said to be *lost*.

A lost interrupt can create havoc within a system. A lost timer interrupt will affect the accuracy of the system clock – losing several could cause unacceptable clock drift. To account for such circumstances, a software routine can be periodically invoked to synchronize the local time with a source that is known to be correct. The Network Time Protocol (NTP) is a networking protocol for clock synchronization between computer systems over variable-latency data networks[19]. NTP is intended to synchronize all participating computers to within a few milliseconds of Coordinated Universal Time (UTC)[20].

## 5.8 Memory Hierarchy

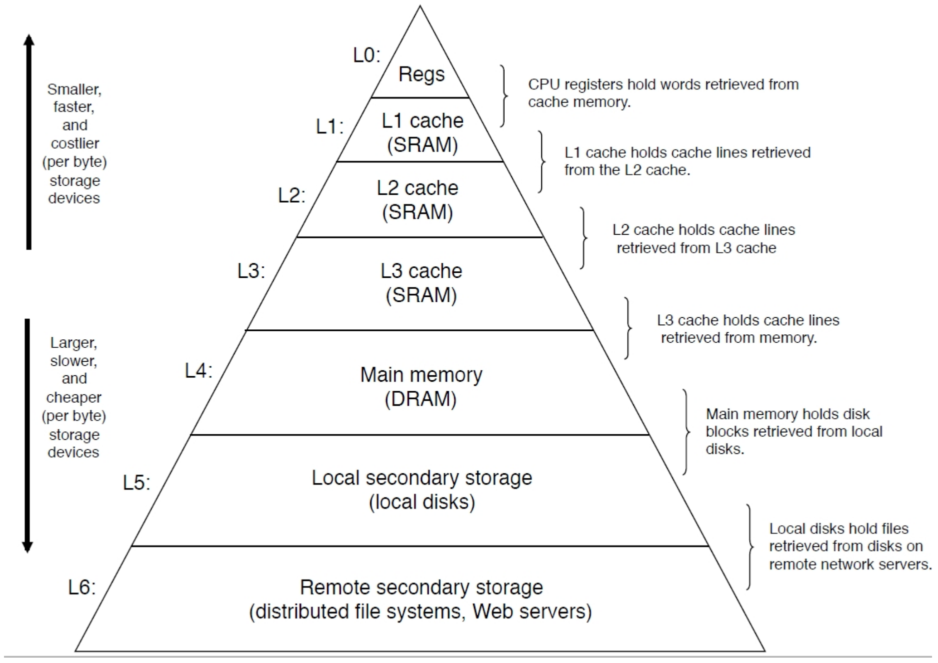
Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

Figure 10: Memory Hierarchy

## 5.9 Cache Memory

A memory cache is invisible to the programmer[21]. Intelligent use of the cache, via hardware design and good programming technique, results in improved performance. Cache theory relies on the *Principle of Locality*.

- Memory references by the processor tend to cluster
- Access to data in the *working set* typically satisfied high in the memory pyramid

[19]See Wikipedia.

[20]See Wikipedia.

[21]Caches and caching was studied in your CS 201 class and will be studied in detail in your architecture class. In this course, we will only discuss in detail a cache used for virtual memory translation – the Translation Lookaside Buffer (TLB).

- Can be applied across more than one level of a memory hierarchy
- A critical factor in *paging*

There are two main types of locality: *spatial* and *temporal*.

- Spatial locality. If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In this case, it is common to attempt to guess the size and shape of the area around the current reference for which it is worthwhile to prepare faster access for subsequent reference. Moving data that we expect to need soon to a faster memory is known as prefetching.
- Temporal locality. If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future. There is a temporal proximity between the adjacent references to the same memory location. In this case, it is common to make efforts to store a copy of the referenced data in faster storage and keep the copy around to reduce the latency of subsequent references. Temporal locality is a special case of spatial locality, namely when the prospective location is identical to the present location.

Components of cache design can impact programs, including operating systems.

- Associativity and Replacement Policy

  ▷ Tags, sets, lines blocks, and all that.

  ▷ When you need to evict an item from a cache, how do you go about selecting a victim for replacement?

  ▷ Different approaches have different impacts and require different complexity of hardware design.

- Coherence

  ▷ All reads of a memory location by any processor must return the most recently written value.

  ▷ Writes to the same location by any two processors are seen in the same order by all processors.

- Consistency

  ▷ When a written value will be returned by a read.

  ▷ If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A.

- False Sharing. This occurs primarily when a cache line contains both data being referenced and data not being referenced by this processor. It is particularly bad when one processor wants part of the line at the same time that another processor wants a different part of the line. This means that the cache line must be shared, in some way, between the caches of two different processors. Let's just say that this can be very bad for performance and also very hard to get right for the memory hierarchy designers[22].

[22]See Wikipedia.

- Contention
  - ▷ Data
  - ▷ Lock

Two terms that are related to this concept, but are much more broad are

1. Serializability. In concurrency control of a transaction[23], both centralized and distributed, a transaction schedule (ordering) is serializable if its outcome (the post-transaction state) is equal to the outcome of its transactions executed serially, i.e. without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transaction executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control. As such it is supported in all general purpose database systems. Strong strict two-phase locking (SS2PL)[24] is a popular serializability mechanism utilized in most of the database systems (in various variants) since their early days in the 1970s[25].

2. Linearizability. In concurrent programming, an operation (or set of operations) is linearizable if it consists of an ordered list of invocation and response events (callbacks), that may be extended by adding response events such that:

   (a) The extended list can be re-expressed as a sequential history (is serializable), and

   (b) Any response that precedes an invocation in the original list also precedes that invocation in the sequential history.

   Informally, this means that the unmodified list of events is linearizable if and only if its invocations were serializable, but some of the responses of the serial schedule have yet to return.

For this class, *serializability is important* but linearizability is not.

## 6 Virtualization and Abstraction

The operating system is an abstract virtual machine in which programs may execute.

What are the differences between a virtualization and an abstraction?

- Virtualization.
  1. In computing, virtualization means to create a virtual version of a device or resource, such as a server, storage device, network or even an operating system where the framework divides the resource into one or more execution environments. See this article.
  2. In computing, virtualization refers to the act of creating a virtual (rather than actual) version of something, including virtual computer hardware platforms, storage devices, and computer network resources[26].

[23]E.g., of databases, transaction processing (transaction management), and various transactional applications (e.g., transactional memory and software transactional memory)

[24]See Wikipedia.

[25]See Wikipedia.

[26]See Wikipedia.

Note that these two are essentially the same but the sources in the footnotes have different supplemental information.

- Abstraction[27]. An entity that is purely a software construct, such as a process or thread of control. An abstraction has no corresponding physical world entity.

Like the program and process, it is mostly harmless to interchange virtualization and abstraction because the correct term is almost always clear from the context in which it is being used.

## 6.1 Key Virtualizations

- Processor Virtualization. A *virtual processor* is an idealized processor that each computer process believes it has exclusively. The term is also used in other parts of computer science to mean subtly difference concepts.
- Virtual Memory. *Virtual memory* is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine" which creates the illusion of a very large, contiguous and dedicated memory for each process[28].
- Storage / Persistence

## 6.2 Key Abstractions

- Process
- Thread (of execution)
- File system

    Directories

    Files

# 7 Concurrency and Parallelism

- Parallel means to happen at the same time. A computer can execute as many processes in parallel as it has physical processors.
- For this class, concurrent is defined as to *happen at the same time or appear to happen at the same time*. Concurrency can happen with a single processor, whereas parallelism cannot. The *appear* part is done via *time sharing* of the physical processors.

Some possible new issues for programmers

- Reentrancy. A thread of control is called reentrant if multiple invocations can safely run concurrently. The concept applies even on a single processor system, where a reentrant procedure can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution. The interruption could be caused by an internal action such as a jump or call, or by an external action such as

[27]The term "abstraction" has different definitions in different parts of computer science. Do not confuse them!
[28]See Wikipedia.

an interrupt or signal. The previous invocations may resume correct execution before the reentered invocation completes, unlike recursion, where the previous invocations may only resume correct execution once the reentered invocation completes[29].

## 8 Programs and Processes

For this course, the terms "operating system" and "kernel" are synonymous.

The terms *program* and *process* are often used interchangeably. However, they each have very specific meanings. Since the meanings largely do not overlap, it is fine to talk about *when a program is executing* or *when my program does X*. Using "program" in situations where "process" is the correct term is almost always obvious by context and so little or no confusion usually results. However, the idea of a process being a file on disk that contains executable code plus data would be very, very wrong[30].

- Program. A computer program is a passive collection of instructions. Is the source code the program? What about the compiled but unlinked object code?
  - ▷ Source code
  - ▷ Object file
    - – Static linking
    - – Dynamic linking
  - ▷ Executable file
    - – ELF (Executable and Linkable Format). Linux
    - – PE (Portable Executable). Microsoft.
    - – a.out. Original UNIX executable format. Trivia: why does the gcc compiler same the executable file "a.out" if no output file name is specified?
    - – COFF. Replaced a.out and was subsequently replaced by ELF.
- Process. A process is not a file format. Nor is it merely a program. In computing, a process is the instance of a computer program that is being executed. A process contains the executable code from the program file on disk, data, some of which is from the program file, and all state necessary for the process to execute. Each instance of a program in execution is a separate process. A process always has at least one *thread of control*.

## 9 Operating System Design

An operating system can be viewed as a program that controls the execution of other programs. These other programs are called "user programs" or "commands". For example, the shell you use when you log in to a Linux system is a user program.

An operating system can be viewed as having three primary objectives.

[29]See Wikipedia.

[30]See Chapter 7 of *Computer Systems: A Programmer's Perspective*, 3rd ed., Bryant and O'Hallaron, Prentice Hall, 2015. This is the course text for CS 201, Computer Systems Programming.

| | |
|---|---|
| Convenience | The OS makes the physical computer easier to use. Without the operating system, users would have to program *bare hardware*, which is both cumbersome and error-prone. |
| Efficiency | The OS can be optimized for efficiency across several axes. |
| Evolution | The OS should be constructed so as to permit efficient development and integration of new functionality. |

Table 4: Primary OS Objectives

## 9.1 OS as User/Computer Interface

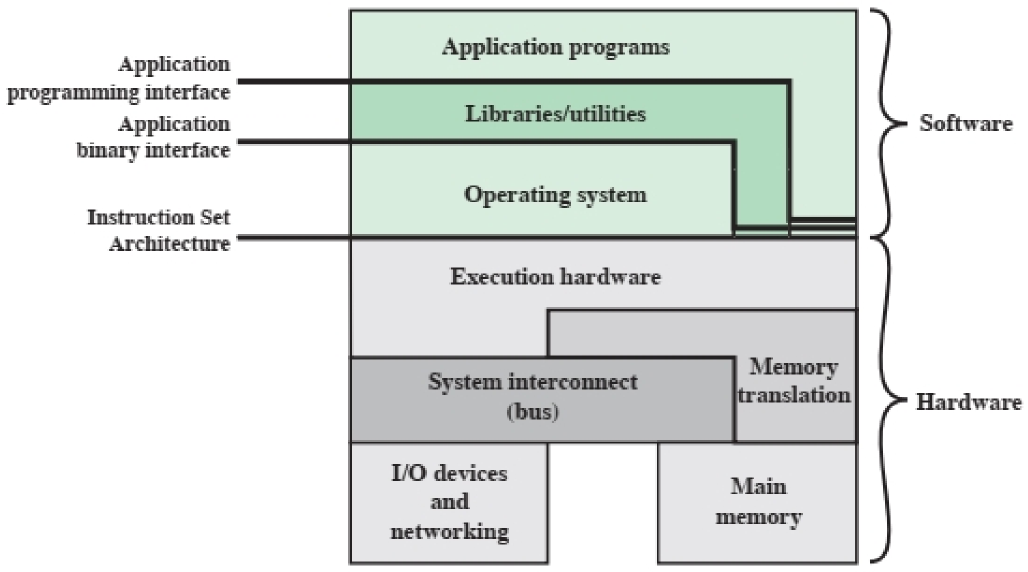
Application programming interface
Application binary interface
Instruction Set Architecture
Application programs
Libraries/utilities
Operating system
Execution hardware
System interconnect (bus)
Memory translation
I/O devices and networking
Main memory
Software
Hardware

Figure 11: A Layered View of Software and Hardware

As can be seen in Figure 11, application programs and libraries cannot access the hardware directly. The OS is the *mediator* of access, usually providing access through one or more *system calls.*

This layering allow the OS to provide services in several areas.

1. Program development. Editors, compilers, linkers, loaders, etc. for a wide variety of computer languages and executable file formats.
2. Program execution
3. Access to I/O devices
4. Controlled access to files and directories, via a file system (persistence) abstraction
5. System access
6. Error detection and response

7. Accounting
8. Instruction set architecture (ISA)
9. Application binary interface (ABI)
10. Application programming interface (API)

### 9.2 OS as Resource Manager

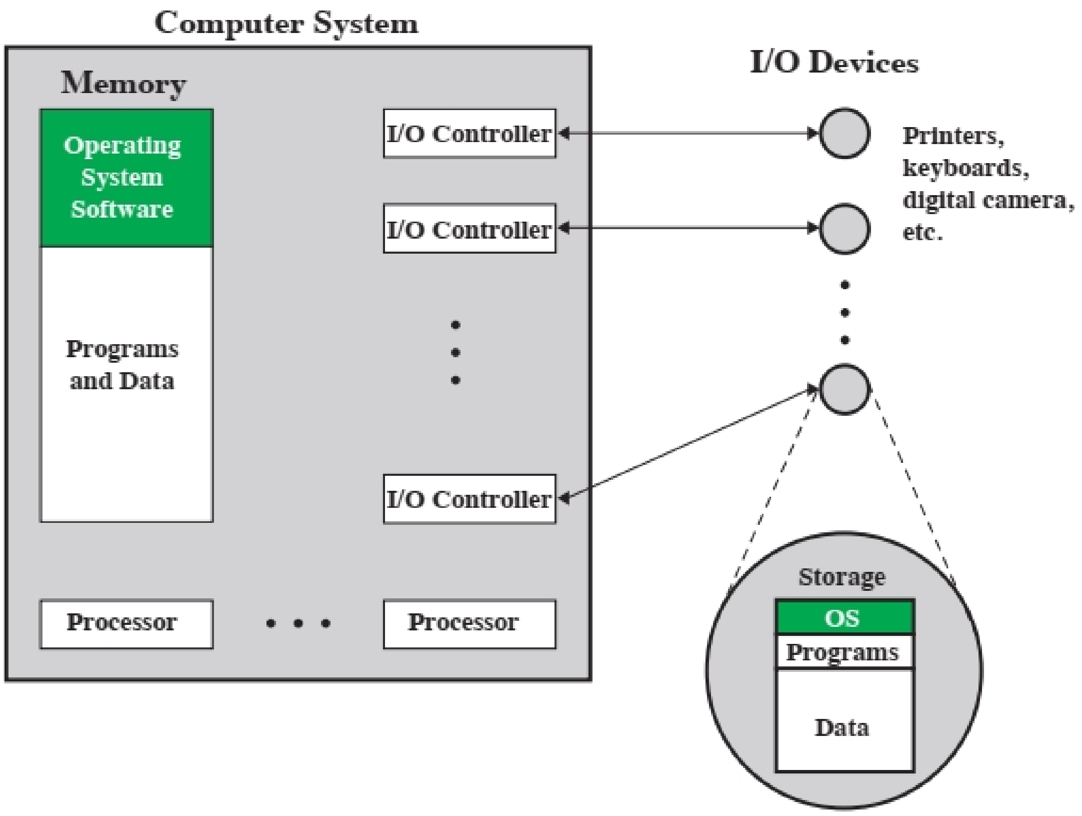

Figure 12: OS as Resource Manager

### 9.3 OS as Virtual Machine

In computing, virtualization refers to the act of creating a virtual (rather than actual) version of something, including virtual computer hardware platforms, storage devices, and computer network resources.

Virtualization began in the 1960s, as a method of logically dividing the system resources provided by mainframe computers between different applications. Since then, the meaning of the term has broadened.

A virtual computer system is known as a “virtual machine” (VM): a tightly isolated software container with an operating system and application inside[31]. Each self-contained VM is completely

[31]See VMWare

independent. Putting multiple VMs on a single computer enables several operating systems and applications to run on just one physical server, or “host.”

A thin layer of software called a “hypervisor” decouples the virtual machines from the host and dynamically allocates computing resources to each virtual machine as needed.

### 9.3.1 Key Properties

VMs have the following characteristics, which offer several benefits.

1. Partitioning
   (a) Run multiple operating systems on one physical machine.
   (b) Divide system resources between virtual machines.
2. Isolation
   (a) Provide fault and security isolation at the hardware level.
   (b) Preserve performance with advanced resource controls.
3. Encapsulation
   (a) Save the entire state of a virtual machine to files.
   (b) Move and copy virtual machines as easily as moving and copying files.
4. Hardware Independence
   (a) Provision or migrate any virtual machine to any physical server.

### 9.3.2 Hardware Virtualization

Hardware virtualization[32] or platform virtualization refers to the creation of a virtual machine that acts like a real computer with an operating system. Software executed on these virtual machines is separated from the underlying hardware resources. For example, a computer that is running Microsoft Windows may host a virtual machine that looks like a computer with the Ubuntu Linux operating system; Ubuntu-based software can be run on the virtual machine.

In hardware virtualization, the host machine is the machine that is used by the virtualization and the guest machine is the virtual machine. The words host and guest are used to distinguish the software that runs on the physical machine from the software that runs on the virtual machine. The software or firmware that creates a virtual machine on the host hardware is called a hypervisor or virtual machine monitor.

Different types of hardware virtualization include:

1. Full virtualization. An almost complete simulation of the actual hardware to allow software environments, including a guest operating system and its apps, to run unmodified.
2. Paravirtualization. The guest apps are executed in their own isolated domains, as if they are running on a separate system, but a hardware environment is not simulated. Guest programs need to be specifically modified to run in this environment.

[32]This section is a copy of the corresponding page at Wikipedia.

Hardware-assisted virtualization is a way of improving overall efficiency of virtualization. It involves processors that provide support for virtualization in hardware, and other hardware components that help improve the performance of a guest environment.

Hardware virtualization can be viewed as part of an overall trend in enterprise IT that includes autonomic computing, a scenario in which the IT environment will be able to manage itself based on perceived activity, and utility computing, in which computer processing power is seen as a utility that clients can pay for only as needed. The usual goal of virtualization is to centralize administrative tasks while improving scalability and overall hardware-resource utilization. With virtualization, several operating systems can be run in parallel on a single central processing unit (processor). This parallelism tends to reduce overhead costs and differs from multitasking, which involves running several programs on the same OS.

Hardware virtualization is not the same as hardware emulation. In hardware emulation, a piece of hardware imitates another, while in hardware virtualization, a hypervisor (a piece of software) imitates a particular piece of computer hardware or the entire computer. Furthermore, a hypervisor is not the same as an emulator; both are computer programs that imitate hardware, but their domain of use in language differs.

**Snapshots** A snapshot is the specific state of a virtual machine, and generally its storage devices, at an exact point in time. A snapshot enables the virtual machine's state at the time of the snapshot to be restored later, effectively undoing any changes that occurred afterwards. This capability is useful as a backup technique, for example, prior to performing a risky operation.

Virtual machines frequently use virtual disks for their storage; in a very simple example, a 10-gigabyte hard disk drive is simulated with a 10-gigabyte flat file. Any requests by the VM for a location on its physical disk are transparently translated into an operation on the corresponding file. Once such a translation layer is present, however, it is possible to intercept the operations and send them to different files, depending on various criteria. Every time a snapshot is taken, a new file is created, and used as an *overlay*[33] for its predecessors. New data is written to the topmost overlay; reading existing data, however, needs the overlay hierarchy to be scanned, resulting in accessing the most recent version. Thus, the entire stack of snapshots is virtually a single coherent disk; in that sense, creating snapshots works similarly to the incremental backup technique.

Other components of a virtual machine can also be included in a snapshot, such as the contents of its random-access memory (RAM), BIOS settings, or its configuration settings. "Save state" feature in video game console emulators is an example of such snapshots.

Restoring a snapshot consists of discarding or disregarding all overlay layers that are added after that snapshot, and directing all new changes to a new overlay.

**Migration** The snapshots described above can be moved to another host machine with its own hypervisor; when the VM is temporarily stopped, snapshotted, moved, and then resumed on the new host, this is known as migration. If the older snapshots are kept in sync regularly, this operation can be quite fast, and allow the VM to provide uninterrupted service while its prior physical host is, for example, taken down for physical maintenance.

**Failover** Similar to the migration mechanism described above, failover allows the VM to continue operations if the host fails. Generally it occurs if the migration has stopped working. However, in

[33]We will not discuss this technique in this course.

this case, the VM continues operation from the last-known coherent state, rather than the current state, based on whatever materials the backup server was last provided with.

**Nested Virtualization** Nested virtualization refers to the ability of running a virtual machine within another, having this general concept extendable to an arbitrary depth. In other words, nested virtualization refers to running one or more hypervisors inside another hypervisor. Nature of a nested guest virtual machine does not need not be homogeneous with its host virtual machine; for example, application virtualization can be deployed within a virtual machine created by using hardware virtualization.

Nested virtualization becomes more necessary as widespread operating systems gain built-in hypervisor functionality, which in a virtualized environment can be used only if the surrounding hypervisor supports nested virtualization; for example, Windows 7 is capable of running Windows XP applications inside a built-in virtual machine. Furthermore, moving already existing virtualized environments into a cloud, following the Infrastructure as a Service (IaaS) approach, is much more complicated if the destination IaaS platform does not support nested virtualization.

The way nested virtualization can be implemented on a particular computer architecture depends on supported hardware-assisted virtualization capabilities. If a particular architecture does not provide hardware support required for nested virtualization, various software techniques are employed to enable it. Over time, more architectures gain required hardware support; for example, since the Haswell microarchitecture (announced in 2013), Intel started to include VMCS shadowing as a technology that accelerates nested virtualization.

### 9.3.3 Desktop Virtualization

Desktop virtualization is the concept of separating the logical desktop from the physical machine.

One form of desktop virtualization, virtual desktop infrastructure (VDI), can be thought of as a more advanced form of hardware virtualization. Rather than interacting with a host computer directly via a keyboard, mouse, and monitor, the user interacts with the host computer using another desktop computer or a mobile device by means of a network connection, such as a LAN, Wireless LAN or even the Internet. In addition, the host computer in this scenario becomes a server computer capable of hosting multiple virtual machines at the same time for multiple users.

As organizations continue to virtualize and converge their data center environment, client architectures also continue to evolve in order to take advantage of the predictability, continuity, and quality of service delivered by their converged infrastructure. For example, companies like HP and IBM provide a hybrid VDI model with a range of virtualization software and delivery models to improve upon the limitations of distributed client computing. Selected client environments move workloads from PCs and other devices to data center servers, creating well-managed virtual clients, with applications and client operating environments hosted on servers and storage in the data center. For users, this means they can access their desktop from any location, without being tied to a single client device. Since the resources are centralized, users moving between work locations can still access the same client environment with their applications and data. For IT administrators, this means a more centralized, efficient client environment that is easier to maintain and able to more quickly respond to the changing needs of the user and business. Another form, session virtualization, allows multiple users to connect and log into a shared but powerful computer over the network and use it simultaneously. Each is given a desktop and a personal folder in which they store their files. With multiseat configuration, session virtualization can be accomplished using a single PC with multiple monitors, keyboards, and mice connected.

Thin clients, which are seen in desktop virtualization, are simple and/or cheap computers that are primarily designed to connect to the network. They may lack significant hard disk storage space, RAM or even processing power, but many organizations are beginning to look at the cost benefits of eliminating "thick client" desktops that are packed with software and making more strategic investments. Desktop virtualization simplifies software versioning and patch management, where the new image is simply updated on the server, and the desktop gets the updated version when it reboots. It also enables centralized control over what applications the user is allowed to have access to on the workstation.

Moving virtualized desktops into the cloud creates hosted virtual desktops (HVDs), in which the desktop images are centrally managed and maintained by a specialist hosting firm. Benefits include scalability and the reduction of capital expenditure, which is replaced by a monthly operational cost.

### 9.3.4 Containerization

Operating-system-level virtualization, also known as containerization, refers to an operating system feature in which the kernel allows the existence of multiple isolated user-space instances. Such instances, called containers, partitions, virtual environments (VEs) or jails (FreeBSD jail or chroot jail), may look like real computers from the point of view of programs running in them. A computer program running on an ordinary operating system can see all resources (connected devices, files and folders, network shares, processor power, quantifiable hardware capabilities) of that computer. However, programs running inside a container can only see the container's contents and devices assigned to the container.

Containerization started gaining prominence in 2014, with the introduction of Docker.

### 9.3.5 Other Types of Virtualization

1. Software.
   (a) Application virtualization and workspace virtualization: isolating individual apps from the underlying OS and other apps; closely associated with the concept of portable applications
   (b) Service virtualization: emulating the behavior of specific components in heterogeneous component-based applications such as API-driven applications, cloud-based applications and service-oriented architectures
2. Memory.
   (a) Memory virtualization: aggregating random-access memory (RAM) resources from networked systems into a single memory pool
   (b) Virtual memory: giving an app the impression that it has contiguous working memory, isolating it from the underlying physical memory implementation
3. Storage
   (a) Storage virtualization: the process of completely abstracting logical storage from physical storage
   (b) Distributed file system: any file system that allows access to files from multiple hosts sharing via a computer network

   (c) Virtual file system: an abstraction layer on top of a more concrete file system, allowing client applications to access different types of concrete file systems in a uniform way
   (d) Storage hypervisor: the software that manages storage virtualization and combines physical storage resources into one or more flexible pools of logical storage
   (e) Virtual disk: a computer program that emulates a disk drive such as a hard disk drive or optical disk drive (see comparison of disc image software)

4. Data
   (a) Data virtualization: the presentation of data as an abstract layer, independent of underlying database systems, structures and storage
   (b) Database virtualization: the decoupling of the database layer, which lies between the storage and application layers within the application stack overall

5. Network
   (a) Network virtualization: creation of a virtualized network addressing space within or across network subnets
   (b) Virtual private network (VPN): a network protocol that replaces the actual wire or other physical media in a network with an abstract layer, allowing a network to be created over the Internet

## 9.4 OS as Evolutionary Entity

1. Hardware upgrades
2. New hardware
3. New services, internal and at the OS boundary. Linux module concept
4. Fixes / Patches

## 9.5 OS Performance

| **Term** | **Description** |
|---|---|
| Overhead | The additional cost, in terms of latency, of adding additional features to the operating system. |
| Efficiency | An algorithm or implementation is efficient if it achieves the minimal overhead. |
| Fairness | How should different system users be treated? Should all users and resources be treated equally or should some get preferential treatment? If there is a bias in treatment, how does the system minimize the negative aspects? |
| Response Time | The time taken from when a task arrives in the ready, or runnable, state until it is scheduled on a processor. |
| Throughput | The rate at which the system completes tasks. |
| Predictability | The degree of consistency in system metrics over time. For example, response time. |

# 10 OS Evolution

## 10.1 Serial and Simple Batch

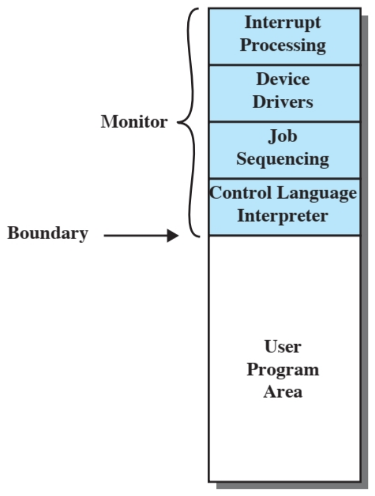

Figure 13: Memory Layout for a Resident Monitor

## 10.2 Multiprogrammed Batch

Remember that these systems are single processor. Some supercomputers of the day had more than one processor, but they were the exception.

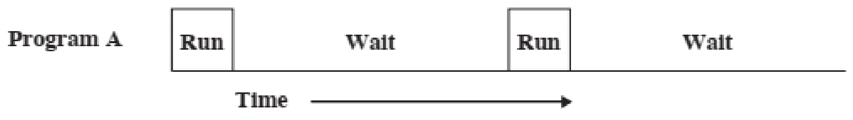

Figure 14: Processor Utilization Uniprogramming

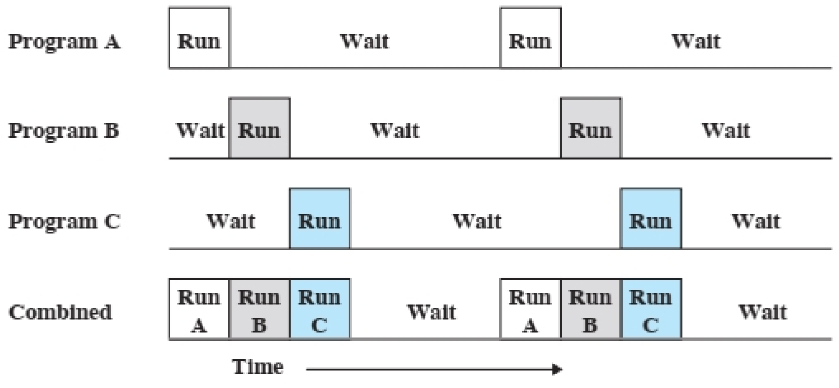

Figure 15: Processor Utilization Multiprogramming

**Example**

| | **Job 1** | **Job 2** | **Job 3** |
|---|---|---|---|
| **Type of Job** | Heavy Compute | Heavy I/O | Heavy I/O |
| **Duration** | 5 min | 15 min | 10 min |
| **Memory** | 50 MB | 100 MB | 75 MB |
| **Disk?** | No | No | Yes |
| **Terminal?** | No | Yes | No |
| **Printer?** | No | No | Yes |

Table 6: Sample Program Execution Attributes

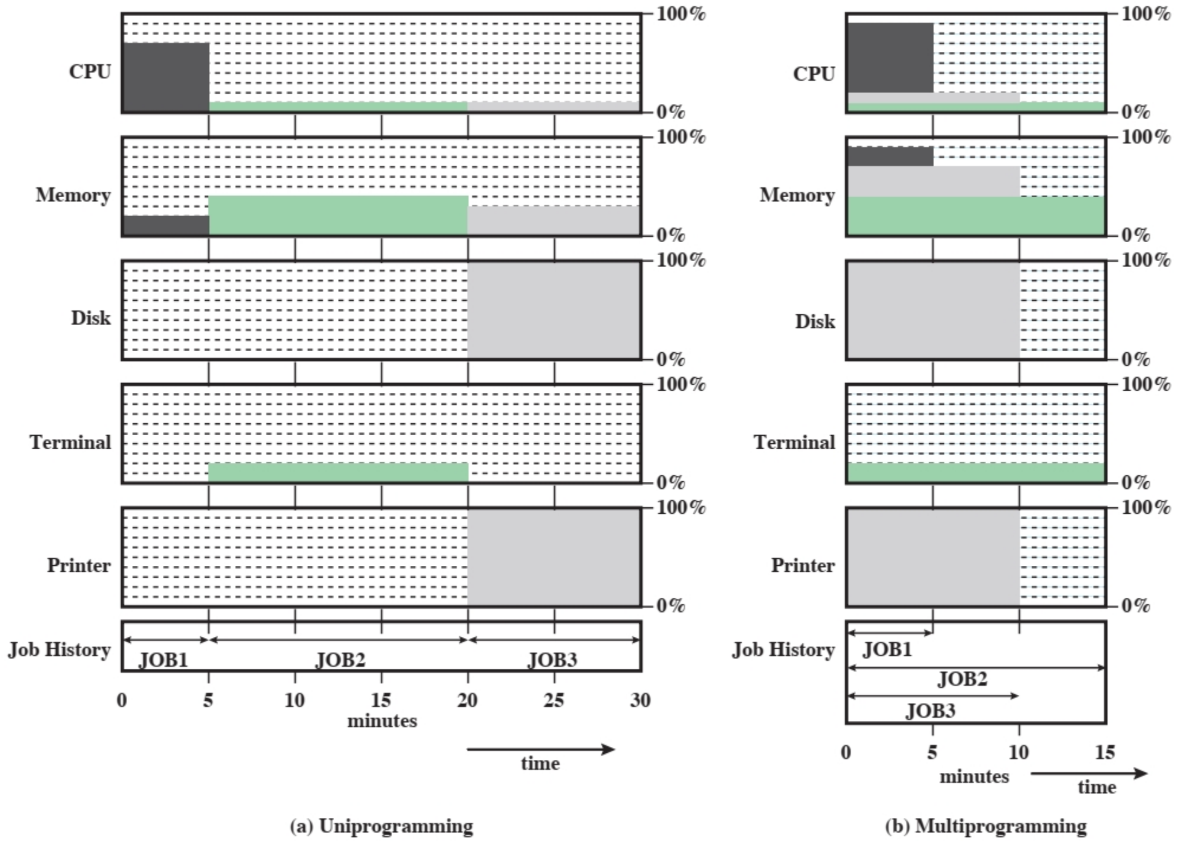

Figure 16: Utilization Histograms

| | **Uniprogramming** | **Multiprograming** |
|---|---|---|
| **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 |

Table 7: Effects of Multiprogramming on Resource Utilization

## 10.3 Time-Sharing

Designed for environments where user interaction is desirable. The invention of the *programmable interrupt controller* (PIC) and the resultant timer interrupt[34] allowed for the concept of a *time slice* wherein at regular intervals, the currently executing process could be interrupted and a new process allowed to run on the processor.

Multiprogramming allows for *schedule compression* as shown in Figure 16. Compression is possible because, at times, a process may be scheduled into the processor but unable to make progress as it is waiting on an outside event (e.g., I/O). As a side-effect of making a system call, process $P_A$ could be removed from the processor in favor of a new process. An example is when a process performs operations on a file. I/O devices are *very, very slow* compared to the processor. As a result, letting the process remain in the processor, when all it can do is wait for I/O completion, is wasteful of the processor resource(s). By preempting the waiting process, one or more processes can perform meaningful computations during the time that $P_A$ would otherwise waste processor time.

## 10.4 Programming Interface

The operating system can perform any operation allowed by the hardware and provide many services. Many of the services it provides are because if a user program had to provide the service directly, the potential for negative impact on other users of the system would be high. Since the OS is *trusted*[35], it provides common and protected services to user processes and these services are considered trusted and optimal. Your process doesn't really print to the screen; it requests that the operating system do so for it.

The set of privileged services provided to a process by the operating system is encapsulated in its *system calls*. System calls are specific entry points into the kernel to request specific protected services. On Linux, system calls are documented in section 2 of the on-line manual. The command "man 2 intro" will give you an overview.

You should be familiar with C library functions. These functions provide common services to C programs; for example, `strncpy()`. Many C library routines invoke the services of the underlying operating system via system calls. For example, printf() invokes the write() system call on Linux, while another OS, such as OpenVMS[36], may invoke an entirely different system call.

[34] It really wasn't in the original concept of a processor. It was added later.
[35] Think about what it would mean to have an untrusted OS
[36] See OpenVMS.com.

### 10.4.1 Single UNIX Specification

The Single UNIX Specification is the standard against which the core interfaces of a UNIX OS are measured. The UNIX standard includes a rich feature set, and its core volumes are simultaneously the IEEE Portable Operating System Interface (POSIX) standard and the ISO/IEC 9945 standard. The specification encompasses the base operating system environment, networking services, windowing system services, and internationalization aspects and programming languages. The latest version of the certification standard is UNIX V7, aligned with the Single UNIX Specification Version 4, 2018 Edition[37].

### 10.4.2 Linux Standards Base (LSB)

The Linux Standard Base (LSB) is a joint project by several Linux distributions under the organizational structure of the Linux Foundation to standardize the software system structure, including the Filesystem Hierarchy Standard used in the Linux kernel. The LSB is based on the POSIX specification, the Single UNIX Specification, and several other open standards, but extends them in certain areas.

According to the LSB:

> The goal of the LSB is to develop and promote a set of open standards that will increase compatibility among Linux distributions and enable software applications to run on any compliant system even in binary form. In addition, the LSB will help coordinate efforts to recruit software vendors to port and write products for Linux Operating Systems.

### 10.4.3 The C Library

The header files and library facilities that make up the GNU C Library are a superset of those specified by the ISO C standard. The library facilities specified by the POSIX standards are a superset of those required by ISO C; POSIX specifies additional features for ISO C functions, as well as specifying new additional functions. In general, the additional requirements and functionality defined by the POSIX standards are aimed at providing lower-level support for a particular kind of operating system environment, rather than general programming language support which can run in many diverse operating system environments.

The GNU C Library Reference Manual is key reading for C programmers using the GNU C compiler, linker, assembler, etc. Chapters 13 and 14 are low-level I/O.

For information regarding the ANSI C standard library, see C standard library. There are many editions of ANSI C. They are referred to by the year of publication: C89, C90, C95, C99, C11, C18.

## 10.5 Major Ideas

### 10.5.1 The Process

In computing, a *process* is an instance of a computer program that is being executed by one or many threads. It contains the program code and its activity, including runtime data. Depending on the operating system (OS), a process may be made up of multiple threads of execution (multithreading) that execute instructions concurrently.

[37] See The Open Group and Wikipedia.

While a computer program is a passive collection of instructions, a process is the *actual execution* of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often results in more than one process being executed.

Multitasking is a method to allow multiple processes to share processors and other system resources. Each processor (core) executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish (preemption). Depending on the operating system implementation, switches could be performed when tasks initiate and wait for completion of input/output operations, when a task voluntarily yields the processor, on hardware interrupts, and when the operating system scheduler decides that a process has used its fair share of processor time (e.g, by the Completely Fair Scheduler of the Linux kernel).

A common form of multitasking is provided by processor time-sharing[38]. Preemption has an important side effect for interactive process that are given higher priority than processor bound processes: when this is the case, users are immediately assigned computing resources at the simple pressing of a key or when moving a mouse. Furthermore, applications like video and music reproduction are given some kind of real-time priority, preempting any other lower priority process. In time-sharing systems, context switches are performed rapidly, which makes it seem like multiple processes are being executed simultaneously on the same processor. This appearance of simultaneous execution of multiple processes is called *concurrency.*

For security and reliability, most modern operating systems prevent direct communication between independent processes, providing strictly mediated and controlled inter-process communication functionality.

### 10.5.2 Virtual Memory

- Process Isolation for protection and security
- Contiguous address space makes programming *much* easier
- Dynamic Allocation
- Separation of *idealized model* and the *messy real world* of physical memory

### 10.5.3 Persistence

In a sense, answers the question *What happens to my data when the power goes away?*

Data is said to *persist* once it is written to a storage device that is *non-volatile*, meaning that the data will still be available even if the device loses power, etc. While the memory hierarchy contains many levels, it is only the bottom levels that persist data. For example, a disk drive, SSD, tape drive, etc.

## 10.6 Modern OS Design

- Concurrent processes or threads (KLTs)
- Scheduling

[38]A method for interleaving the execution of users processes and threads, and even of independent kernel tasks – although the latter feature is feasible only in preemptive kernels such as Linux.

- Synchronization
- Memory Management
- Parallelism within Applications
- Multiprocessor and multicore
- Virtual Machine

## 10.7 Windows

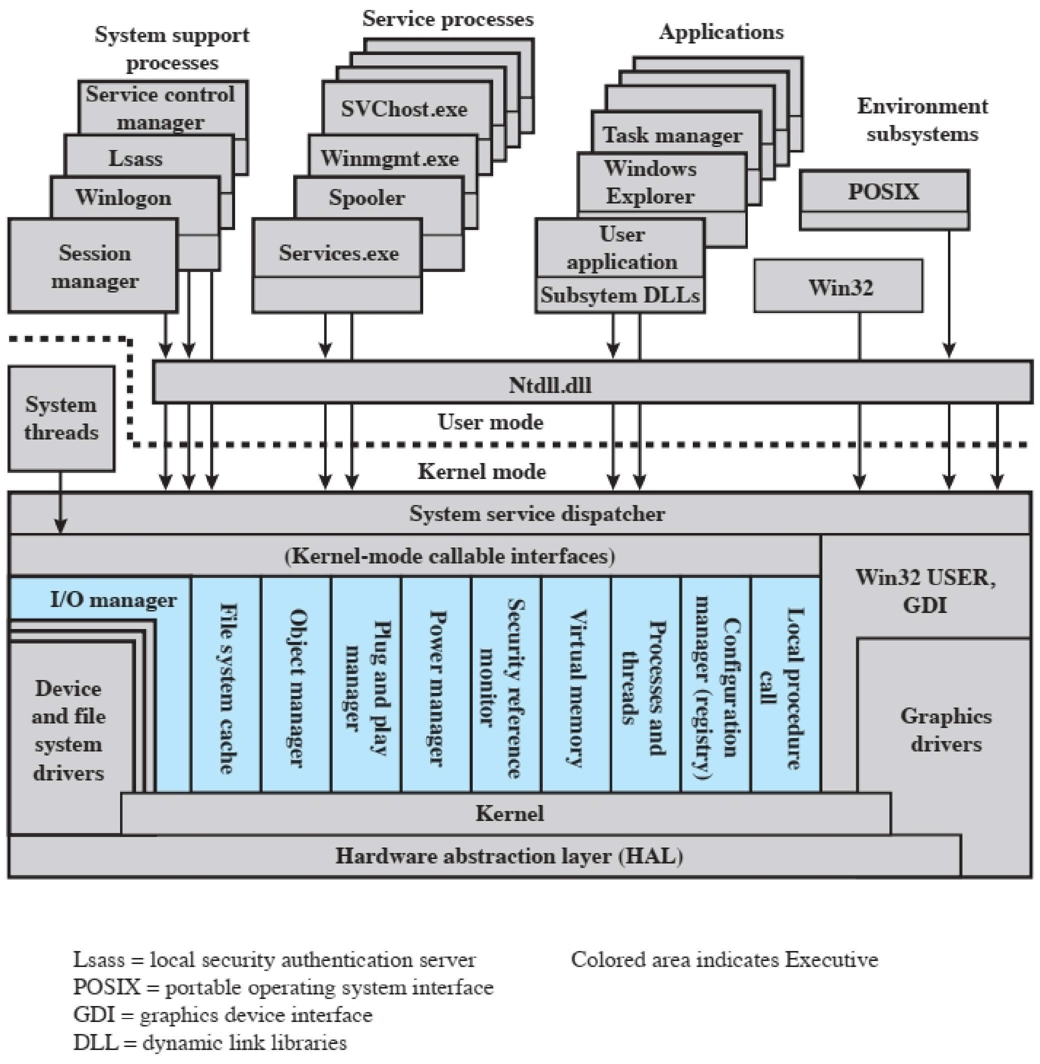

Figure 17: Windows Internals Architecture

Note that in Windows, we differentiate between the kernel (the portion that is always resident) and the operating system.

# 11 UNIX

## 11.1 Traditional UNIX

UNIX is an old operating system – development began in 1969[39]. The C programming language was developed by Kernighan and Ritchie specifically to develop UNIX in a high level, portable, programming language. At the time, operating systems were generally written in assembly because of time-critical tasks and performance. The principle reasons for writing in assembly were

- Memory (both RAM and secondary storage (think "disk") was small and expensive. Often, specialized techniques such as overlays has to be used, which could result in decreased performance.
- Compilers were considered, by most of the industry, as not producing efficient code. With small resource capacity, efficient code, in terms of both time and space, was essential.
- Processor and clock speeds were slow, so saving clock cycles, via efficient, hand-tuned assembly code was important for performance, especially for frequently executed code, such as the scheduler or virtual memory translation.

This approach demonstrated to the industry the advantages of using a high-level language to most, if not all, operating system code. Today, virtually all UNIX implementations are written in C, with small amounts of assembly for boot code, which must run before the compiled C code can be loaded and executed.

There have been many UNIX variants over the years, with UNIX System V (AT&T) prevailing at this time. Many of the features relied on today, however, came from the BSD distribution.

Traditional UNIX is organized as a unified kernel with very little modularization, which resulted in an unwieldy approach to adding new features.

[39] See this paper under "References" on the class web page.

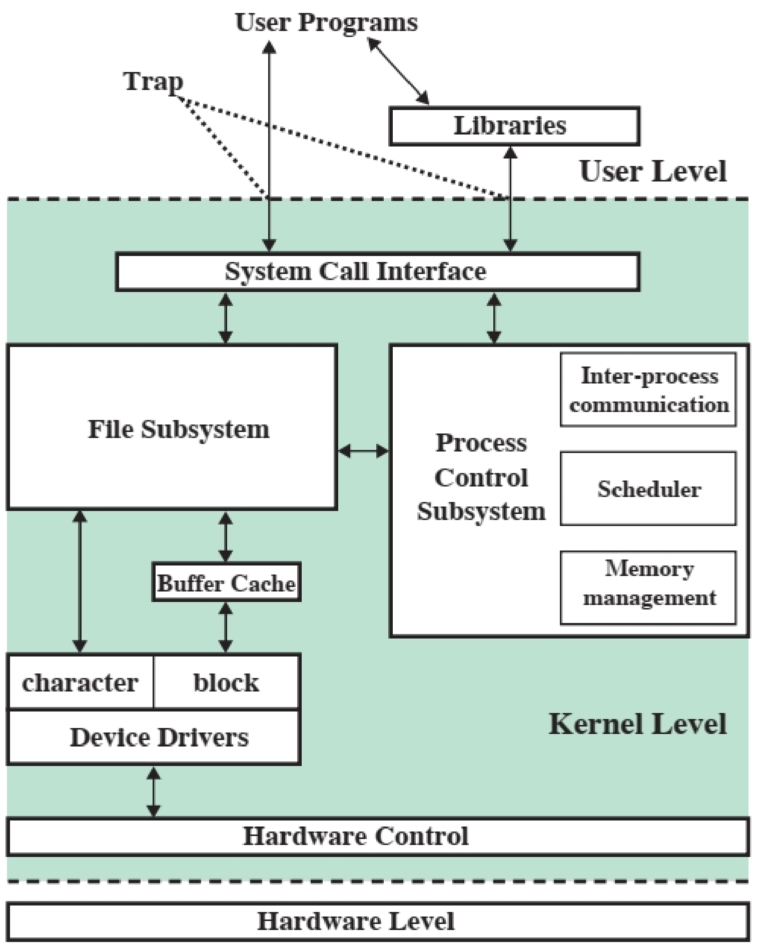

Figure 18: Traditional UNIX Architecture

## 11.2 Modern UNIX

Modern UNIX kernels are much more modular in design. Many useful features from a variety of transitional approaches were unified in this architecture, of which there are three main exemplars.

- System V Release 4 (SVR4)
- BSD 4.x
- Solaris 11

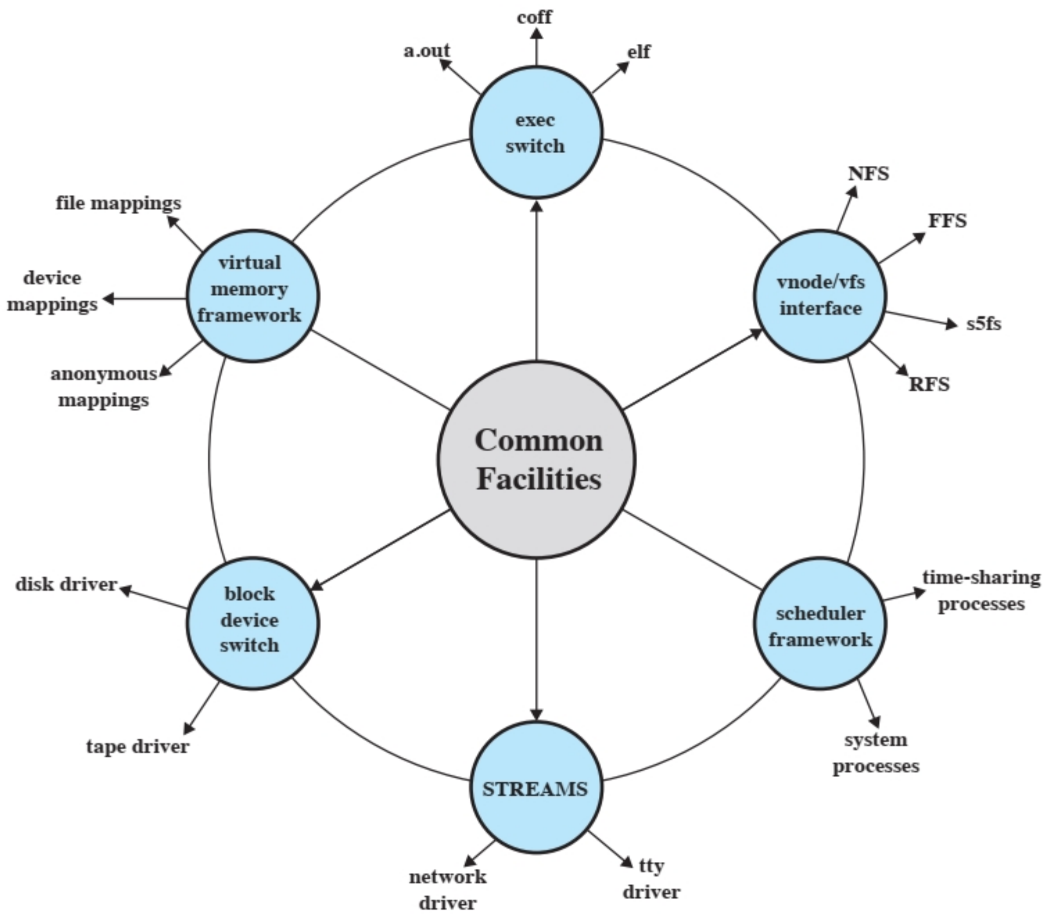

Figure 19: Modern UNIX Kernel Architecture

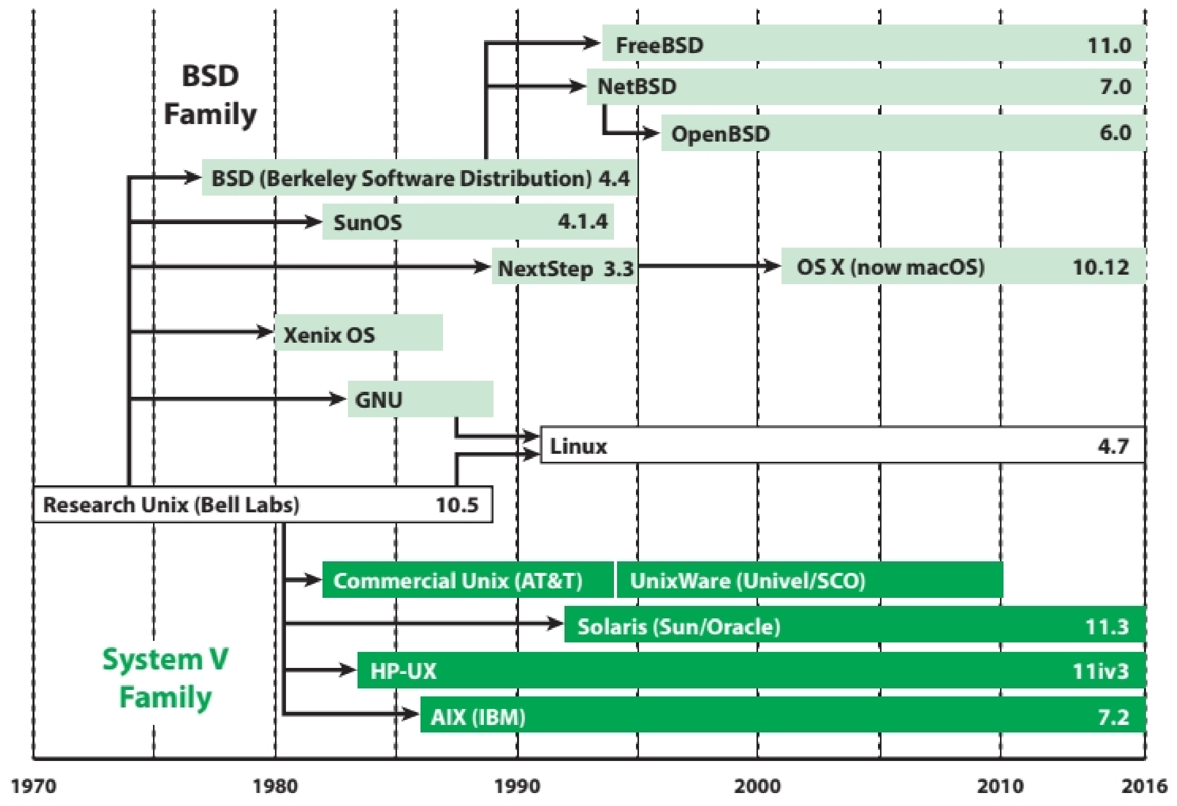

Figure 20: UNIX Family Tree

## 12 Linux

Linux was developed by Linus Torvalds who had the foresight to use the UNIX programming interface for his new, innovative operating system. There is a dispute regarding the name Linux.

> The GNU/Linux naming controversy is a dispute between members of the free software community and the open-source software community over whether to refer to computer operating systems using a combination of GNU software and the Linux kernel as “GNU/Linux” or “Linux”[40].
>
> Proponents of the term Linux argue that it is far more commonly used by the public and media and that it serves as a generic term for systems that combine that kernel with software from multiple other sources, while proponents of the term GNU/Linux note that GNU alone would be just as good a name for GNU variants which combine the GNU operating system software with software from other sources.
>
> GNU/Linux is a term promoted by the Free Software Foundation (FSF) and its founder Richard Stallman. Their reasoning is that the GNU project was the main contributor for not only many of the operating system components used in the subsequent development of modern “Linux” systems, but also the associated free software philosophy.

[40]See What’s in a Name? by Richard Stallman.

> Several distributions of operating systems containing the Linux kernel use the name that the FSF prefers, such as Debian, Trisquel and Parabola GNU/Linux-libre.

It is important to note that while Linux implements the Single UNIX Specification, it is not, internally, UNIX. This confuses a great many module developers. It is safe to say that UNIX user-mode programmers can easily reuse C code using the UNIX programming interface with Linux, but internally, Linux is not UNIX. Linux has many system calls that can be used to replace certain UNIX system calls to provide enhanced functionality.

A key feature of Linux is its use of *loadable modules* which are relatively independent blocks of functionality, most of which can be dynamically loaded (and unloaded) at runtime. A module does not run in its own process or thread; rather, a module is executed within the kernel on behalf of the invoking process.

While still a monolithic kernel, the modular structure overcomes some of the limitation of UNIX in developing and evolving the kernel.

Linux loadable modules have two important characteristics

1. Dynamic linking. A kernel module can be loaded and linked into the kernel at runtime while the kernel is already in memory and executing.
2. Stackable modules. Modules are arranged in a hierarchy, similar to modern SVR4. Lower-level modules can be used as libraries for modules higher in the hierarchy. That is, the higher level module can be seen as a *client* of the lower-level one. Modules can also be used as clients for modules further down the hierarchy. With stackable modules, dependencies between modules can be specified. This has two important benefits
   (a) Common code can be abstracted to a single module, eliminating duplication.
   (b) The operating system can ensure that these dependencies are satisfied before a specific module is loaded and executed. This contributes to reliability.

Figure 21 shows the main components of the Linux kernel as of 2016[41].

[41] If anyone has a link to a current figure, I would appreciate a pointer.

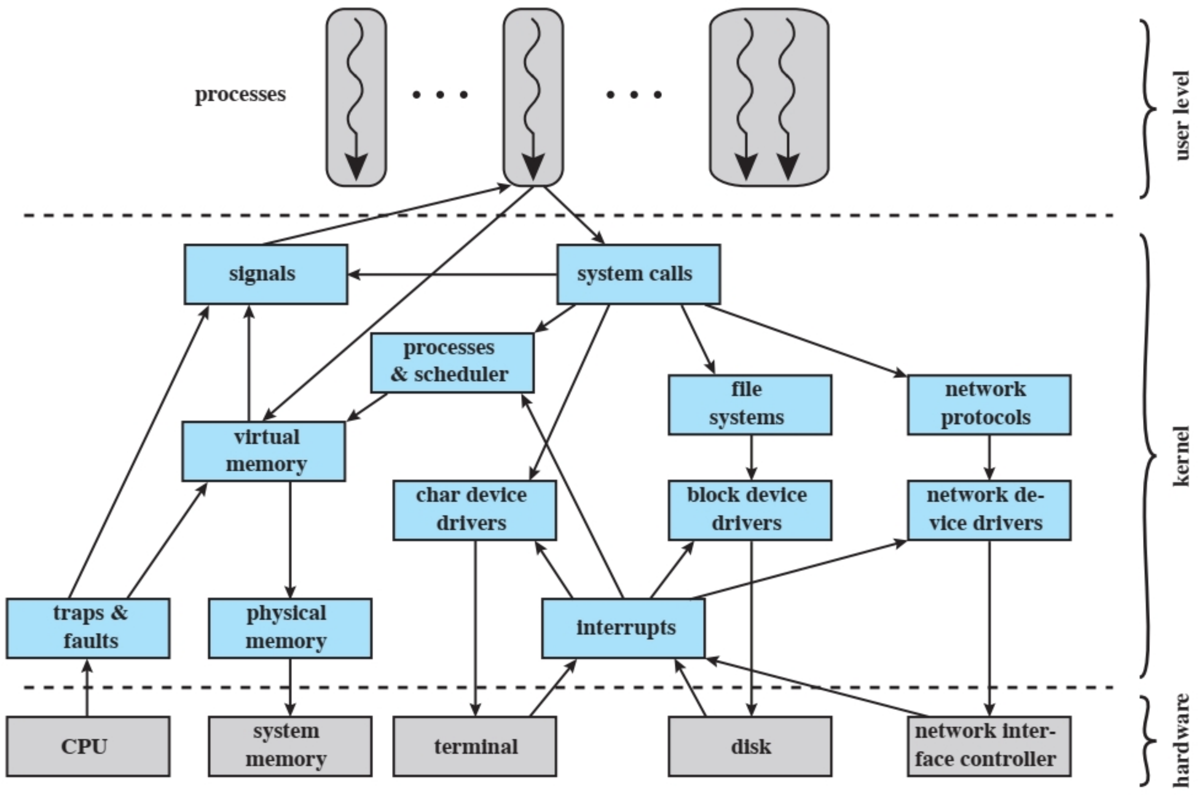

Figure 21: Linux Kernel Layout

The principle components are

- Signals. The kernel uses signals to call into a process. Signals are used to notify (signal) user processes of events such as divide by zero or even the termination of a child process.
- System calls. System calls, see Table 8 for some examples, are the primary mechanism by which user processes request privileged services from the operating system. System calls are organized into roughly six categories

  1. File systems
  2. Processes
  3. Scheduling
  4. Interprocess communications (IPC)
  5. networking (sockets, streams)
  6. Miscellaneous

- Processes and scheduler. Creates, manages and schedules processes.
- Virtual memory. Allocates and manages virtual memory on behalf of processes and the kernel.

| **System Call** | **Description** |
|---|---|
| **File-system related** | |
| open | Open and possibly create a file or device. |
| close | Close a file descriptor. |
| read | Read from file descriptor. |
| write | Write to file descriptor |
| link | Make a new name for a file. |
| Process related | |
| fork | create a new process by cloning the process invoking the call. |
| execve | Execute program. |
| _exit | Terminate the calling process. |
| getpid | Get process identification. |
| setuid | Set user identity of the current process. |
| ptrace | Provides a means by which a parent process my observe and control the execution of another process, and examine and change its core image and registers. |

Table 8: Some Linux System Calls

## 13 Study Questions

1. Is attendance required for this class?
2. What should you do if you cannot attend one or more classes?
3. (T/F) The survival guide is required reading.
4. Where can you find written information on how to submit your homework?
5. What is the process for challenging the score for a class assignment?
6. A project report that is not submitted in PDF format will receive what score?
7. What are the ramifications of academic dishonesty for this class?
8. Where can you find information on coding standards for this class? What is its URL?
9. What is the URL of the class slack site?
10. List and explain the steps for creating an archive file of your source code for submission and grading. Be sure to account for any new files that you may have created.
11. Explain why a consistent and understandable coding style is important.
12. List and explain the primary objectives for an operating system.
13. What are the four main parts of a computing system?
14. Define *latency*.
15. Explain why it is useful for modern processors / cores to have more than one mode.
16. What is the principle difference between *parallel* and *concurrent* execution?
17. List and explain the parts of the "instruction execution cycle".
18. List the steps of the instruction execution cycle as explained in class.
19. Define the term "trap".
20. Define the term "exception".
21. Define the term "interrupt".
22. What does is mean for an interrupt to be lost?
23. Why is *reentrancy* a problem with concurrent programming when it can be mostly ignored in non-concurrent programming?
24. Why is that C library routines must be reentrant, even on a system that does not support threads?
25. List and explain two approaches to handling multiple interrupts.
26. Explain why traps are synchronous to the executing thread of control.

27. Define “privileged instruction”.
28. How is protection provided by a processor?
29. (T/F) A process can tell that it has been preempted.
30. On an interrupt, why is a special stack frame created?
31. Caches work as long as our process obeys what principle?
32. Define “working set”.
33. What is a principle benefit for using *virtual memory*?
34. Explain the term *false sharing*.
35. List and define the two types of locality.
36. Define *cache coherence*.
37. What is meant by the term *prefetch*?
38. Define *serializability*.
39. Define *linearlizability*.
40. What is meant by the term *snapshot*?
41. In Figure 11, why is the instruction set architecture placed where it is?
42. What are the principle differences between a *program* and a *process*?
43. (T/F) A process contains all the data/information in a program file.
44. Why must an operating system be a hardware resource manager?
45. What is a *virtual machine*?
46. What do you think is a potential downside to *nested virtualization*? (this question asks for an opinion)
47. What is *containerization*?
48. Explain how Figure 14 and Figure 15 illustrates multiprogramming.
49. Multiprogramming helps improve utilization of what operating system resource?
50. What is meant by the term *schedule compression*?
51. What invention enabled time-sharing?
52. What is the difference between a C library call and a system call?
53. In Figure 15, why are libraries/utilities shown as above the operating system?
54. How does a user program gain access to protected resources? Why?

55. Why are processes typically prevented from direct communications?
56. Why might we allow two processes to directly communicate. That is, to exchange information without involving the operating system?
57. (T/F) Time-sharing is a critical component to multiprogramming?
58. What is the purpose of the *programmable interrupt controller* (PIC)? (you may need to do some research for this one)
59. What is the purpose of system calls?
60. From the perspective of the programmer, what is the principle benefit of *virtual memory*?
61. Which provides better portability, (1) the single UNIX specification; or (2) the standard C library?
62. How is *preemption* of a user process typically initiated?
63. (T/F) Multi-threading is a technique that is beneficial to programs that have a high degree of parallelism.
64. Why is "multicore" likely to dominate the processor market?
65. Highly efficient compilers enabled more and more operating systems code to be written in a high-level programming language. Why?
66. The C programming language invented in order to write the UNIX operating system. Why?
67. Until UNIX, why were most operating systems written in assembly language?
68. Summarize the issue that some have with how Linux is named.
69. What are the two important characteristics of Linux loadable modules?
70. These will require that you use the man pages on a Linux system and the C Library Reference manual.
    - Why is it acceptable for a `read()` system call to return fewer bytes that were requested.
    - How would you ensure that the number of bytes read equaled the number of bytes requested?
    - Chapter 22 of the C library reference manual is all about process limits. Why does the operating system impose these limits?
    - Without process limits, what type of attack could be launched against the users of the system?
    - Section 14.2.4 of the C library reference manual contains source code for listing the contents of a directory. How would you modify this program to not print *dot files*?
    - What is the purpose of the `getpid()` system call?
    - What is a UNIX pipe?
    - The `time()` system call returns what information?
    - The `random_r()` C library call addresses the reentrancy issue of the `random()` C library call how? Why does this work?