Notes for the Operating Systems course (CS347)

Purushottam Kulkarni
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

puru@cse.iitb.ac.in

April 15, 2021

Chapter 1

Ring for Operating Systems[1]

The world of *Systems* and more specifically *Operating Systems* is the subject of this course. As part the first chapter, we will setup context for this sub-area in Computer Science and Engineering, understand its need, requirements and services, discuss a few key building blocks in the design of a modern operating system and end with a set of examples of the above.

1.1 What is an operating system?

Systems in general and more specifically, operating systems are the engine of a machinery that enables interesting and new software products to be developed. An apt analogy for a student, practitioner, researcher in the area of Systems is that of a mechanic, one who knows the inside-out details of a vehicle. She can fix a puncture, tune an engine, take the engine apart and reassemble, change the carburetor etc. and use all this information to reason about performance and utility of the vehicle and also to design better engines and vehicles. The Operating System is the (software) engine of the computer engineering and applications world and a systems practitioner/researcher it's (software) mechanic. The key to building fast, fancy, interesting, new, efficient applications is a reliable and efficient engine—*the fate of the free world rests on the operating systems.* Okay! analogies apart, what is an operating system?

1.1.1 Abstractions

What is a machine without an operating system or any software? — cold stone, a paper weight. The hardware system—microprocessors, memory, devices and all the paraphernalia, requires to be told what to do. The interface to program all hardware is via the instruction set architecture (ISA), an interface specified via a set of instructions and a system model (consisting of registers and memory). e.g., the CISC and RISC ISAs, the x86 ISA.

While this is the basis, we seem to be happily coding away to glory and developing programs, applications, spam and what-not like a current run-rate of 20 runs per over[2] . . . what is the catch? Is programming hardware with low-level instruction interfaces programmed in to our DNA? Well! obviously no. The catch is the design and engineering practice of *abstractions*, in fact a layer of abstractions. The idea of an abstraction is to hide details of implementation and to expose an interface for specific functionality. In fact, the ISA itself in an abstraction. The system software stack consists of several abstractions, and on top of which is the user and her applications. Some common examples of abstractions,

[1]The title is a rip off of (inspired by) the book, *Ring for Jeeves*, by P.G. Wodehouse. A highly recommended read.
[2]Cricket is boring when this happens.

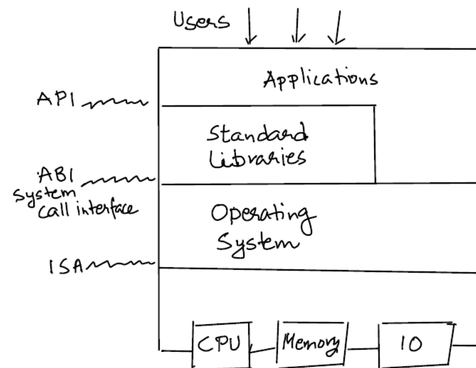

Figure 1.1: A typical software stack with its interfaces.

1. **ISA** The Instruction Set Architecture.
   The instruction set itself provides a set of functionalities and an interface to use them. Users do not need to manually manipulate bits for encoding operations on logic gates to get the hardware moving. Users issue instructions (`mov, jmp, cmp`) etc. and the implementation associated with the instruction does what is required.

2. **File** The ubiquitous file on a disk. A file abstraction provides functionalities to create, open, read, write, append files, without the user of a file knowing how the file data is being manipulated on the disk—which sector on disk to read from, how to get to the sector, how to identify a file and its contents on disk etc. The file abstraction provides a layer which hides the details of managing the specifics of file related meta-data (information about files themselves—permissions, size, location etc.), the content data, and their access. Once the abstraction and its implementation is in place, we can assume files and their operations as a given, and move onwards to world peace!

3. **Network interface** homework.

4. **Process** homework.

5. **Virtual memory** homework.

Fine, but where does the operating system figure in all this?

1.1.2 The stack of abstractions

As discussed earlier, the basic hardware interface, the ISA, is not suitable for general purpose programming. The **operating system** is a layer that executes on top of bare hardware and hosts several common functionalities related to resources of the hardware by providing different types of abstractions and their implementations. A few definitions of an operating system,

- Software to *manage* computer hardware and help users build programs
- Software to provide an *environment* for execution of programs
- Software to *enable* an usable computing system

The most popular software stack for general purpose computing and for production software is as shown in Figure 1.1. The figure also shows the interfaces exposed by each layer to access functionality of abstractions provided by that layer. The operating system interfaces with the hardware using the

ISA interface (the instruction set, registers, memory addressing etc.) provided by the microprocessor and associated devices.

A modern operating system provides a sizeable set of services and functionality to the higher layers of the software stack. The primary interface to access the operating system functionalities is through an *application binary interface* (ABI). The ABI provides a mechanism for programs to communicate and invoke functionality provided by the operating sustem. The ABI interface of UNIX-like operating systems is called the *system call interface*. A system call invokes a pre-defined functionality in the operating system and requires a strict syntax and procedure for invocation. For example, typically operating systems provide system calls for file management—open, close, etc., which are different from the 'C' Standard library calls. Since functionality of the operating system is already compiled in to a binary, the interface is called the application binary interface.

Referring to Figure 1.1, typically programs need not invoke system calls (and hence OS services) directly, they are routed through standard libraries. For example, *libC*, is the standard C library that translates calls from user programs and invokes appropriate system calls. The standard libraries decouple the program from the underlying OS interface, as long as the program to library interface is standardized, the underlying OS and its interface can change. Standard libraries expose an *application programming interface* for programs to integrate and use with their source code. For example, a typical binary of a C program, will be genrated through a series of steps, one of which involves *linking* the application source code libraries that implement OS and hardware-specific implementation of the API. As long as the API is standardized and has a standard library implementation, programs are portable across OS and hardware instances. For example, the same *helloworld.c* C program runs correctly on Linux and Windows.
Hold on, but this requires the programs to be _______ ? _______ . Now you know why.

1.2 Operating system services

1. Program execution
   This is the basic service for any user of an operating system—to load and start execution of programs, with facilities for graceful or abnormal exit. Additionally, every program in execution (a process) is associated with an *isolated* environment, one process unintentionally or intentionally cannot access and use resources of other processes.

2. IO operations
   A program cannot do many (any?) useful things if it cannot interact with the external world. The interactions of a program are consuming inputs from and sending outputs to different types of input-output devices—storage, network, keyboard etc. Each of these devices have their own standardized interface and hardware implementation. The IO operations of programs are coordinated by the operating system.

3. File management
   The file abstraction is the most ubiquitous abstraction associated with persistent storage. User (via programs) create, store, delete, append data on to files and expect them to be around for later use. The file abstraction supported by operating systems enables this view and shields programmers from getting their hands dirty with details of how to index files, how to get to data that belongs to files, how to manage this meta-data, how to interface with different types of disks etc.

4. Communication
   Similar to input-output, another vital requirement for programs is to communicate with each other, e.g., here is the result you asked for, or here is the web page you asked for. Programs may want to communicate with each other on the same machine or on different machines

across the network. An operating system enables all such modes of communication via the shared-memory, network stack, remote calls and other such services.

5. Resource management
   This is the bread-and-butter service of an operating system. A system has resources and several programs want/compete for resources. Resource demands vary over time and in quantity. The juggler (actually the scheduler, multiplexer, de-multiplexer, decision maker) who manages these resources is the operating system. Example management decisions include, which process to execute next?, has this process run long enough? how do I order these requests to read data of the disk? which process is binging on memory? etc.

6. Error detection
   Packets that you receive may have errors, the disk may have failed—bad sectors, memory locations may be misbehaving—what you write is NOT what you get, a device may malfunction, disks can crash etc.
   "Blistering barnacles (ten thousand or more) who worries,detects and fixes, all this."
   "Ahem! that you would be me, the Operating System."

7. Protection and security
   The execution environment and related state (persistent storage) needs a strong protection mechanism to avoid undesired accesses and usage from internal and external programs. For example, only certain types of users have access to view, modify or execute certain programs, per-user privileges on different directories, blocking or enabling network ports, isolated execution environments etc. The operating system provides several features from authentication, encryption/decryption, isolated resource handling, controlled access to IO devices etc. to build a protection and security framework to safeguard the system.

8. Accounting
   Managing a system and its components is not easy and one of the basic requirements is to know things about the system. This is the goal of the accounting service—to provide usage and record keeping services. Examples include tracking and counting various resources used/consumed by different entities, logging access and error situations etc.

9. User interface
   Last but not the least, operating systems live to serve and they need a channel, a mechanism to offer their services to humanity. All operating systems provide an interface to use their services, from hard-coding user code, to command-line shell interpreters to GUI based interfaces.

1.3 A computer architecture interlude

The hardware of a machine is the real performer in the computing world, the rest is all window dressing (a.k.a software). Be as that may, we all know where the computing world would have been without the abstractions enabled via the software entities, the formidable amongst them, at least for this course, being the operating system. Let's call it a truce! An operating system is tightly coupled with the hardware that it is designed for and needs to understand the features and functionalities provided by the hardware architecture. The following is a less than brief summary of architecture of a computer, its components and interconnections.

A generic model representing the architecture of computers is as shown in Figure 1.2. The three main components are, the CPU (the central processing unit), memory and controllers for peripheral devices. Each of these components are inter-connected via a shared system bus. The CPU and the

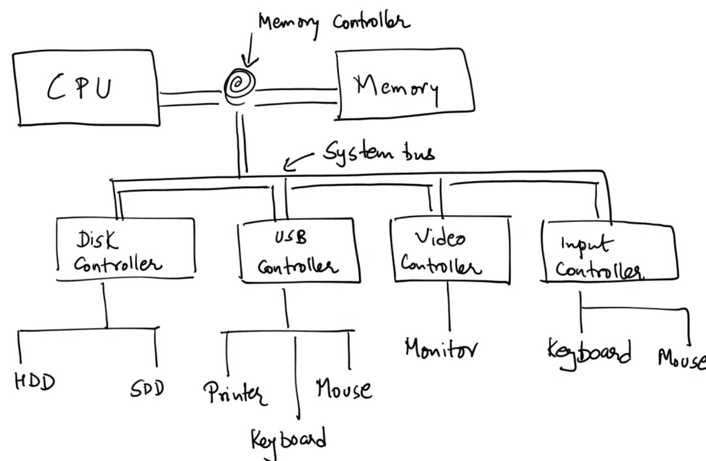

Figure 1.2: A simple conceptual architecture model.

controllers[3] have compute elements and can execute in parallel, but contend on access and use of the system bus. Next, each of these components are described briefly.

1.3.1 Processors

The CPU is characterized by the ISA that it is architected for, it executes instructions of the ISA and implements features specified by the ISA. The von Neumann architecture (which is adhered widely by the general purpose CPUs) specifies a computation model, where the processing unit is separated from the program and associated data. A general purpose compute mechanism interacts with the memory of a system to fetch and execute instructions and access data in memory and on external devices. The compute unit itself has no *embedded* logic if its own. All *logic* and state of programs is loaded in memory, and is to be fetched and executed by the processor.

Instruction execution

The basic execution cycle of the CPU consists of the following sequence of operations—fetch, decode and execute. The CPU consists of a special register called the *program counter* (`ip` or the `eip` of the x86 architecture) that stores the memory address of the next instruction to execute. Once an instruction is fetched, the program counter is incremented to point to the next address, and the loop continues. Advancements in CPU design have led to parallelizing the fetch, decode and execute components via the *pipelining* based multi-stage execution technique. The pipelining idea being, each instruction stage has a different pipeline making progress in parallel— while one instruction is executed, another is decoded and yet another fetched from memory, all in parallel. The pipelining benefits are affected by *branch* instructions, (conditional) instructions that change the address of the instruction pointer to something other than the next sequential address. Branch instructions force flusing of the instruction pipelines and loose benefits of parallel execution. An implication of pipelining on OS design is of code optimization— how to generate machine code to exploit pipelining?

[3]Typically, controllers are examples of specialized hardware running custom software to communicate with and operate peripheral devices.

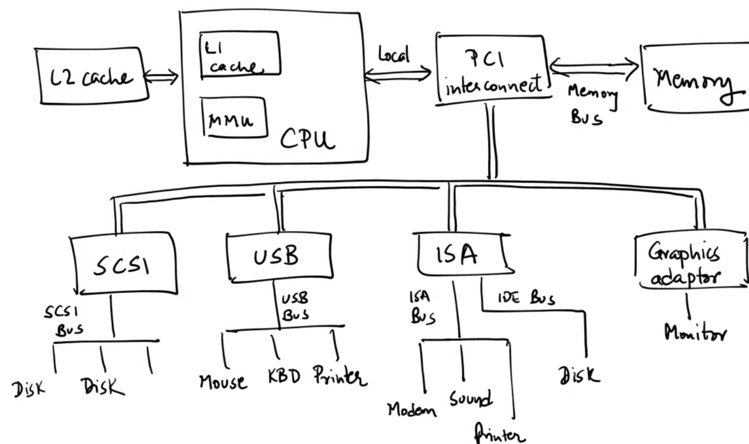

Figure 1.3: A modern computer architecture.

Registers

Other than the program counter (register), the CPU also contains a set of registers to aid in quick access and temporary storage of data, for memory management, to control behavior of the CPU, for indexing , for debuging etc. Details of a few are as follows,

- The general purpose registers are primarily used to store intermediate results, temporary variables and to exploit the fact that access to registers is orders of magnitude faster than access to memory. Usually, a CPU will have instructions that copy data across registers, and from register to a memory location and vice-versa.
- Current state of a CPU is stored in a register called the *program status word* (PSW), also the `eflags` register of the x86 architecture. This register is usually a register of bit-values, each of which reflects a condition or configuration of the CPU. For example, the zero flag bit is set to one if an arithmetic operation by the CPU results in a zero. Similarly, the carry flag, the sign flag, the overflow flag etc. are set for correspoding side-effects of operations. The interrupt flag indicates whether the CPU is currently accepting interrupts etc. To update bits of the status word register privileged access is required. Not all bits can be read or written from user space, while some can be read but not written.
- Another commonly required register of the CPU is the *stack pointer*, which points to the top of the stack in current execution context. The stack pointer is used to store frames of functions—return addresses, local variables, input arguments etc. during a nested function calling sequence.
- Control registers that control behavior of the CPU. For example, bit 0 in the `CR0` register enables or disables protected mode of execution on x86 hardware[4]

On an x86 CPU the following types of CPU registers exist: general purpose registers, segment registers, index registers, pointer registers, flags register, control registers, debug registers, test registers, descriptor table registers, performance monitoring registers etc.

[4]Protected mode of execution is not the same as user-mode vs. kernel-mode execution. We park this point till we reach the memory management discussion.

Multiprocessors

Another angle to push to limits of CPU capabilities was the design of multiprocessors and hyper-threading (multi-threading) systems. With hyper threading, a single CPU supports multiple *hyper-threads*, process equivalents but which can be switched in the order of nanoseconds. The hyper-threads appear the operating system as CPUs, a physical CPU with 2 cores with 2 hyper-threads each, appears are 4 CPUs to the operating system. With multiprocessor systems, the system has multiple CPU cores and processes can execute on any of them. As we will see later CPUs have local optimizations related to memory etc. and also need to communicate with each other to share events. An operating system needs to be aware of the impacts of its process scheduling mechanism to exploit the benefits of parallel compute facility and simultaneously avoid *blind spots* in a multi-processing situation. For example, co-scheduling two processes on two cores which communicate with each other is far better than scheduling them one after the other.

1.3.2 mmm ... Memory

The next important hardware building block of a computer is the memory. Memory itself is available with different characteristics—volatile vs. non-volatile, read-only vs. read+write, different bandwidths (speed of access) etc. In general, access to memory for fetching instructions or transferring data can be orders of magnitude slower than execution instructions. As a result, one would ideally want memory that is characterized by high-bandwidth, large capacity and inexpensively priced. It is seldom the case that this is possible.

To overcome the constraints of not being able to get an ideal single memory setup, the hardware and software architecture of a computing system relies on a hierarchy of memory types. The spectrum of memory utilized on a machine varies from very fast and small capacity to large capacity with relative slower transfer and access rates.

The registers of a CPU are memory that is *closest* to the CPU. Registers are part of the CPU and access to them is as fast as execution of instructions, but these are very few in number (compared to data needs of a program). A standard mechanism used by a system is to use a hierarchy of memory, with different access latencies and capacities, and operate a caching framework for data storage and accesses. The general rule being, if data or instructions are not found at one level (in the hierarchy), move to the next. Each layer being relatively slower as compared to the previous.

Once past the very limited-capacity very-fast registers, the CPU maintains a series of instruction and data caches, the *L1 cache* and the *L2 cache* (Level 1 and Level 2). The L1 cache is usually on the same die as the CPU and does not need to cross a bus for accesses. The L2 cache usually is connected via a special bus, which the CPU has to access the cache, adding an extra delay factor. Size of L1 caches is in the order of tens of Kilobytes (64 KB per core on Intel i7 processors) and hundreds of Kilobytes for L2 caches (25 KB per core on Inter i7 processors) and tens of Megabytes for L3 caches. Different hardware architectures organize L2/L3 caches in different configurations, either as shared caches across all cores, or local to each core with strong consistency maintenance mechanisms.

Next in line the *random access memory*, the cheapest form of memory, very large in capacity and the one with highest access latency, compared to previous L1/L2/L3 caches. The basic CPU architecture assumes the main memory (RAM) to be present where all programs are loaded and data is read from and written to. The CPU has an address bus, which 32-bits or 64-bits wide, that is used to address and access memory. The width of the address space specifies the maximum addressable memory by the CPU. Main memory capacity is in the orders of tens to hundreds of Gigabytes.

All the forms of memory discussed are *volatile* in nature; on power-off contents of memory no longer exist. A new technology trend is maturing that persists data and can be assessed via the address bus, called NVRAM, non-volatile RAM. A similar property exists with flash devices, which appear as block devices, but are faster than conventional hard-disk drives. A nagging question for an operating system designer is to come up with a policy that utilizes the memory the best—no use

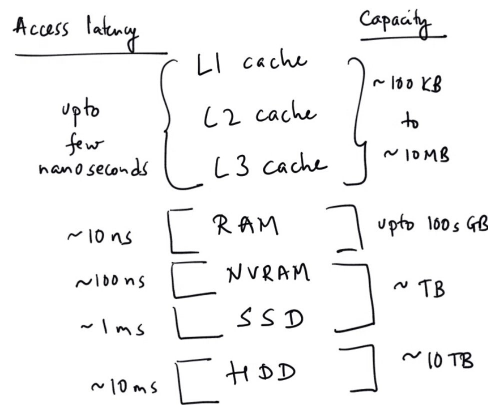

Figure 1.4: The memory hierarchy with access latency and capacity details.

keeping state in memory that is never going to be used! With NVRAMs, an additional dimension of what data should be persisted adds further complexity.

Other types memory on a system are *ROM* (read-only memory) and *EEPROM* (electrically erasable programmable ROM). EEPROMS are typically used to store startup code of machine, which initializes the machine, probes and resets devices and looks for the boot-up procedure.

Figure 1.4 shows a list of different types of memory along with their typical access latencies and capacities.

1.3.3 IO and IO controllers

The last part an architectures jigsaw is the work of IO devices. The architectural details of devices and their low-level interfaces are scary! Take a disk for example, it has spinning plates, several heads, tracks, sectors and a set of low level commands to control their movements. Working with such a fine-level of interface to meet performance requirements of which sector to read-of next is a huge task—moving the head takes time and which sequence of sectors to read and give corresponding commands is the heavy-lifting work.

To simplify matters, as you may have guessed is a layer of abstraction, provided and implemented by device controllers. A disk controller for example exports a view which contains a linearly ordered set of sectors and blocks. An operating system component, called the *device driver* interacts with the controller based on this view and lets the controller handle the actual commands to move the hardware for the intended task. Each device requires a device driver, that understands how to interface with the device. Drivers are device specific and without a driver, interacting with a device is not scalable. Hence, device drivers form an intrinsic component of the IO layer of the operating system.

Devices themselves have several methods of communication, programmed IO, memory mapped IO, direct-memory access (DMA), kernel/operating-system by-pass etc. The operating system (the driver) has to adhere to these interfaces and provide support, so that an OS can setup appropriate configurations for their use. For example, with memory mapped IO, addresses in memory act as device registers. Accessing memory locations implicitly translate to reading and writing to device registers. An operating system needs to know how much memory is required, which memory to map, maintain information about mapping of memory with which devices etc. Similarly, for a DMA setup, the OS has to instruct the CPU to setup a DMA transfer to a memory area that it has setup for use by the corresponding devices driver.

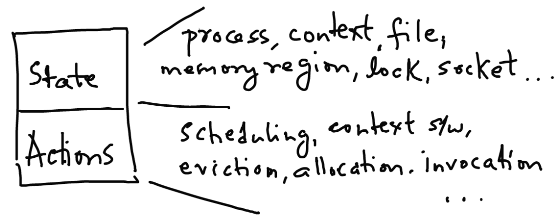

Figure 1.5: The two state OS representation.

1.4 How does the OS do what it does?

This section will reveal the secret formula of every OS, without actually revealing much.

1.4.1 The two boxes explanation

Figure 1.5 shows a conceptual block diagram of an operating system (again, similar to any useful software). All operating systems store *state* and perform *actions*. For the engine of the software world, the state and related actions are centric to the services to be provided and the interfacing required with the underlying hardware. A few examples,

1. The operating system needs a representation to identify processes and store information about them (*state*) of the execution environment maintained by the OS). The OS uses this state to point to processes and take *actions* on intended processes. e.g., context switching, delivery of packets etc. Examples of per-process state are its execution state, the list of open files and offset at which file operations to be performed, the memory region of each process etc.
2. The file system is part of the OS which is responsible for enabling the file abstraction. Files are referred by name, exist on a set of fragmented blocks on disk, exist on different types of disks, have permissions etc. The file system meta-data maintained by an OS forms the crucial state required to carry out the necessary actions for correct implementation of the abstraction.
3. State and action related to IO devices.
4. State and action related to events.
5. State and action related to locks.

1.4.2 Two pillars

An operating system depends on two very critical aspects of operation, (i) privileged mode (dual-mode) of execution, and (ii) support for interrupt-driven execution. We discuss these next.

The kernel and user modes of execution

The operating system trusts no one but itself to correctly offer the services that it promises to offer. Imagine the chaos that will ensue in this world (if already not enough), if each process when scheduled to execute is also given the responsibility of descheduling itself to make way for others, or for programs to adhere to an honor code of not peaking in their neighbors memory. Well! that is not happening in today's world, some day when we figure out a way to leave as a community, may be such a honor-code model could be revisited.

For now, the OS trusts no one and requires absolute *control* of all resources at all times to correctly multiplex/divide them across different execution entities. This control is established via two modes of execution—the *user mode* and the *kernel/privileged* mode. The game plan is as follows,

- All actions to manipulate and in some cases access operating system state and hardware resources are privileged and hence can be performed only in the privileged kernel mode. The kernel mode of execution is the unfettered mode of execution, with access to all resources and state. The operating system obviously sets itself up to execute in the kernel mode and uses its ownership on all its state and resources for control.
- User-level programs execute in a less-privileged mode, the user-mode of execution. In this mode the programs are autonomous and execute without interference or arbitration by the operating system. The caveat being that execution in user mode continues till a privileged action is required, in which case execution switch to the kernel mode and the operating system takes over. The OS then does what needs to be done and reverts back to the less privileged user mode to continue progress of the user program.
- A few examples of the dual-mode of operation,
  - A hardware register, the CR3 register, on x86 systems, is a critical element to support the virtual memory service of the operating system. Each process has a unique entry that is written to CR3 when a process is scheduled for execution. The OS cannot risk a direct access to CR3 from user-space, it takes control, executing in the kernel mode writes the correct value to this register before user-mode process execution begins.
  - An user program needs memory to store state during execution. While operating in the user mode, a program makes a call to request for memory. Every program cannot start marking memory regions to use on their own (world chaos will only increase! verify this at your own risk.). The operating system, the boss, takes care of the situation by looking up its internal *state* and servicing the request. The user-mode request for memory, results in a switch to the kernel mode and back.
- A key enabler to this dual mode of execution is the process of switching between the two modes of execution, especially on-demand from the user-level program. The switch from user-mode to kernel mode can happen due to exceptions—divide-by-zero, illegal memory access, page fault, access to instructions without enough privileges etc. or through an explicit requirement (memory allocation request mentioned earlier). The explicit switch is initiated via the operating system system call interface (Section 1.1.2) to OS services. A system call executes in the kernel mode on behalf of the program (the process context) and executes the desired OS service (as specified via the system call invocation).

  Note that the system call invocation needs a standardized interface and a hardware-assisted process for the switching between modes and invoking the correct system call binary code. More on this to follow in later classes or lookup relevant question in the exercises list.
- Another obvious requirement for the dual mode of execution is that the hardware should support the different modes of execution. Almost all modern general purpose compute hardware already support different privileges modes of execution and an associated minimum privilege level for correct execution for each of its instructions. The operating system uses this functionality to setup the different modes of execution. For example, x86 traditionally supports four levels (rings) of privileges, with each instruction requiring a minimum privilege level for correct execution. In fact, the x86 hardware stores the current level of execution and checks it during instruction execution. Instructions without enough privileges may raise an exception or maybe ignored.

Interrupt-driven execution
The second key of the operating system game plan is interrupt-driven execution. As mentioned earlier (in Section 1.2), interesting stuff happens when programs interact with each other and with the external world. This is accomplished via the hardware-assisted mechanism of *interrupts*, which literally means current execution is interrupted to inform that an event awaits service.

The interrupt service is a tight coupling between the hardware and the operating system. A hardware interrupt is raised by physically toggling the voltage level on pin connected to the CPU. Next, the CPU abandons all work (assuming interrupts are enabled) and switches to servicing the interrupt. The hardware supports the execution pause and switch to interrupt processing, the interrupt handler (also know as the interrupt service routine, ISR) itself is all operating system. The OS determines nature of the interrupt and invokes the appropriate handler for further processing. Typical hardware interrupts are keyboard, network packet arrival, disk block read completion, timer interrupt, etc. All hardware interrupts are non-deterministic (except maybe the timer interrupt), i.e., they can occur at any time.

Software interrupts or exceptions on the other hand are deterministic, they occur due explicit invocation or due to undesired effects during execution. A system call invocation depends on a software interrupt mechanism.

All modern day hardware and operating systems rely on interrupt-driven execution as a basis for computation. Also to add, as devices get faster (100 Gbps network cards) the gap between processing and IO capacity is changing and on some subsystems (like the network) *polling* based IO handling is fast gaining traction. The idea is not to receive and process an interrupt on each packet arrival, but to poll for packets/events and consume several in one go. This mode can be treated as a scope building block for modern day efficient operating systems.

1.5 Operating system types

The above listed services and more are realized via different design and several types of implementation optimizations. The main categorization, similar to any large software development effort, is that of a tightly couple, all functionality rolled in one design or a loosely coupled, communication based modular design—a *monolithic* operating system and *micro-kernel* operating system, respectively. Each of these, has it pros and cons, in terms of maintenance, ease of upgrade, independent trajectories of development for OS subsystems, reliability, ease of use etc. The micro-kernel approach is to build a "thin" kernel that supports the most basic operating system services (memory isolation, process execution etc.) and rely on rest of the services as independent subcomponents of the operating system. As and when functionality of these components is required, the micro-kernel provides and coordinates message passing between components that need to communicate. A monolithic kernel on the other hand, has all services rolled up in a single compiled operating system binary and all state of the operating system is common and shared. Some operating systems also provide programmable extensibility to the kernel, for example loadable kernel module supported by Linux. (for now, more on this via self-study).

Figure 1.6 shows a listing of operating systems of different types—monolithic, micro-kernel, real-time, targeted for embedded systems etc.

1.6 System call implementation details

This will be covered in a later lecture.

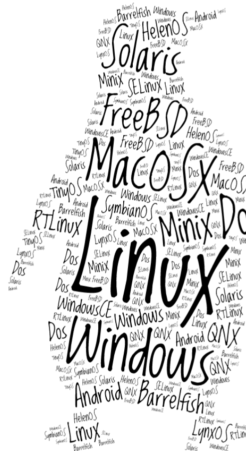

Figure 1.6: Names of different operating systems.

1.7 Food for thought

1. What is the difference between a kernel and an operating system?
2. Linux, Ubuntu, Redhat, Debain ... all these the same?
3. List operating systems based on different types of requirements that are target, e.g., job scheduling, interactive users, real-time systems etc. Describe their target operating environments.
4. On a Linux system, you can know the list of all system calls supported by the kernel. Dig it up.
5. Loadable kernel module in Linux. What are they?
6. Find out characteristics of the example operating systems shown in Figure 1.6.
7. System calls need a standard method for invocation and hardware support. How does Linux handle this on x86 hardware?
8. The timer interrupt is special for an operating system. Why?
9. How do programs all accessing a lot of memory, cumulatively more than the physical memory, make progress?
10. What happens if physically memory is setup for a system which is much smaller than the addressable range?

Chapter 2

A matter of processes

We begin our discussion of the operating systems underbelly with the most tangible entity of the software world—a process. Machines exist to serve, by executing stuff (or is it all the Matrix[1]?). The execution entity, *process*, is also referred to as a *thread*[2], a job, a task, to name a few. Here, we concern ourselves with how an operating system provides and manages the process abstracttion.

2.1 What is a process?

A process is a program in execution/action that consumes the CPU and gets work done. A program or a binary is dead weight, it exists but does not do anything, except warm the disk blocks that it uses. To do anything, the program has to be instantiated, given a life form and setup to run (execute).

An instance of a process is inseparably bound with a program, more precisely with the binary code (sequence of hardware instructions) of the program. Execution of this code on a CPU defines the program and hence the process. The organization of most hardware systems is such that the CPU fetches instructions from memory for execution. As a result a basic requirement to breath life in to a passive program is to *load* the program binary code in memory. Once instructions are in memory, the CPU can access instructions of the process and execute them.

Once the program is available in memory, is process instantiation complete? Nope.
A modern day OS support execution of multi-processes simultaneously, further what about program related data and the support required for programming abstractions like functions. The concept of a process is not complete without including these. Conceptually, a process is made of the following,

- Program code
  The binary program that encodes the work that has to be performed on the CPU. A process is the active entity of the passive program code. Also known as the *text section*, this corresponds to the region of memory that holds the program code.

- Program counter
  Loading a program in memory for execution is not enough, the CPU needs to know which instruction to execute next. This state variable maintains a pointer to the next instruction to be fetched for execution. The counter is also required during process/context switch. When a process is switched out for another process to use the CPU, process *context* has to be saved and restored. An important context of a process is the program counter, to enable resumption of execution from the point where it was paused.

[1] https://en.wikipedia.org/wiki/The_Matrix
[2] Thread is an overloaded term and also refers to a lightweight process.

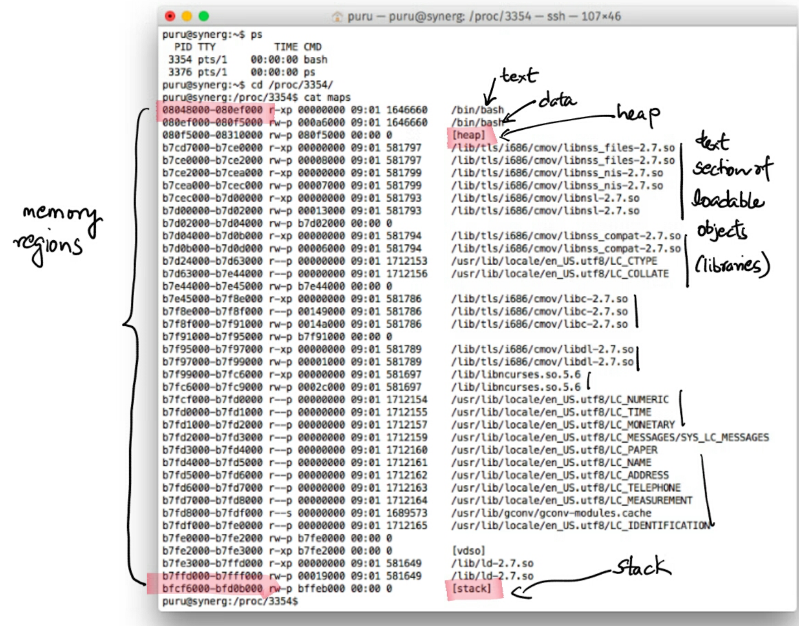

Figure 2.1: Memory regions and components of a process as shown on a Linux system.

- Processor registers
  A process during execution uses the hardware registers available as part of the ISA. Values stored in these registers as part of the process.
  Note that most hardware provides a special instruction pointer register which is saved and restored during context switch. The CPU relies on value on this register for instruction fetch-decode-execute.

- Stack
  All programs rely on some form of function-call abstraction. An abstraction that exposes a functionality along with an interface to specify input arguments and expect return values. A function calling sequence requires the return address and values of the local variables to be stored, before execution is transferred to a function being invoked. The *stack* data structure is an appropriate fit to store function call state and roll it back in an last-in-first-out manner. Each process has an associated stack to utilize for this purpose.

- Data section
  Typically programs are not CPU-centric code, they work with data stored in variables. The data section of a process is the memory region that stores static variables—declared at compile time and pre-allocated memory area for the same. These include, global and local variables.

- Heap
  Similar to static allocation of memory for variables, processes also depend on dynamic allocation—a standard design practice to use memory only when needed. All dynamic allocations and related memory requirements are part of the *heap* area associated with a process.

The *context* of a process is its execution state, which is captured by a pointer to the next instruction for execution, the values of the CPU registers during execution, and memory regions that form the different components of the process (stack, heap, data, text etc.).

Figure 2.1 shows a memory map of a process on Linux system. The following form the figure are to be noted, the memory region for each component of the process, the different components

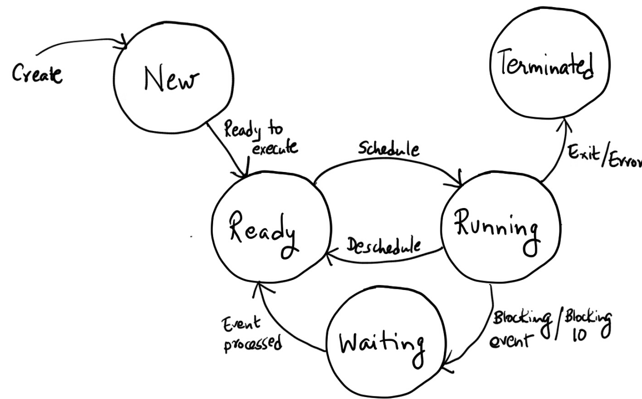

Figure 2.2: The different states of execution a process can be in its lifetime.

themselves—the text section, the data section, the heap, the stack and the text section mapped from several shared libraries. The format of each entry is as follows:
`address range, permissions, offset, device, inode, pathname`

2.2 Process states

Once a process starts execution on a CPU, does it occupy the CPU till it finishes? Nope. There are several other "things" going on in the system, that almost always do not let this happen.

1. There are several process competing for the CPU and the OS needs to keep all them happy. The OS *schedules* processes on the CPU, an action that implies removing a process from the CPU and assigning the CPU to a new process. The *state* of a process executing on the CPU and processes waiting for the CPU are different. Basically, a process once created, is either executing on the CPU or is not assigned a CPU.

2. Another event which defines a different state of process is related to invoking blocking functions. A synchronous or a blocking function is one which blocks the progress of a process till a certain task is done. For example, a process issues a disk read, the process blocks on the read call and waits till the a read is issues and data is read off a storage device. The latency for the read to complete is orders of magnitude slower than the rate at which the CPU can execute instructions. As a result, the OS switches out a process *waiting* for IO to complete and schedules another processes which is *ready* and waiting for the CPU. Once the disk read (IO event) completes, the blocked process can execute again, but not so soon, a process already is occupying the CPU.

Figure 2.2 shows the different states in which a process is during its lifetime. When a new process is created, which implies creation of new operating system state to identify and store context of the process—this state is marked *new* in Figure 2.2. Once the OS state is created the process is ready to execute and is in the *ready* state. Only processes in the ready state can be scheduled for execution on a CPU. Once scheduled, status of a process changes from the *ready* state to the *running* state. On an operating that blocks, like read from disk, state of a process changes to *waiting*, which gets updated to *ready* on completion of the IO request. A CPU may determine to give other processes a chance to use the CPU and deschedule an executing process. In such a scenario, state of the executing process changes from *running* to *ready* and the process is descheduled from the CPU.

Once a processes finishes its task and exits or hits an error condition, the state of the process is set to *terminated*, in which clean up and book-keeping tasks are completed, before the process vanishes in to thin air.

2.3 Process representation

As mentioned in Section 1.4, an operating systems maintains *state* so that it can perform actions to fulfill the services that it promises to deliver. Supporting all aspects of process management—creation, isolated execution environments, memory areas, scheduling etc. require the operating system to maintain state/information about processes. For example, index/tag a process by a name or a number or a combination, so that it can do the equivalent of "Hey Rastapopoulus, guess what? break over, get on the CPU and churn out some work and Mr. Haddock, why don't you take break and do not worry I will keep the engine running for you". The operating needs a mechanism to represent every process and its context for managing processes.

A *process control block*(PCB) is the operating system's representation (state) of a process[3]. The PCB stores all the information relevant for each process:

- Identifier
  A number and name of the process. This information is often used by user-level tools to communicate with processes. e.g., `kill -9 <pid>`.

- State
  The state of execution of a process—ready, running, waiting, terminated/halted etc. For example, a process which has been halted would not be selected by the OS to be scheduled on the CPU.

- Execution context
  During execution, a process occupies the CPU and uses its registers which defines the execution context of the process. The PCB needs to store this context—registers, program counter, stack pointer etc., which need to preserved for correct pause-and-resume of processes in a multi-processing setup.

- Memory layout information
  As discussed in Section 2.1, a process consumes memory for different purposes—to load the executable in memory, to allocated memory for its variables, the stack etc. The OS needs to maintain state to know the memory regions allocated to each process. This information is related to the memory-related information that the OS stores on a per process basis in the PCB. Not only does the OS use this information to identify memory regions, store and use information regarding attributes of the region—read only, read/write, execute etc. The story of memory is much more interesting, *virtual memory is coming!*

- IO-related information
  The primary IO related information stored as part of the PCB is related to files—the list of files opened by the process, offset in each file at which the next file operation will execute, a pointer to the cache where file content is cached etc. Linux also maintains a pointer to a list of functions that can override the default file functions on a per process basis, e.g., system programmer, with appropriate rights, can override the functions associated with file operations. `read` can be replaced with `myread`.

- Scheduling and accounting information
  This category includes information related to scheduling of processes, the time spent so far

[3] When you mean an OS maintains state, you mean store stuff in memory right? Yes.

on the CPU, priority of the process, time spend waiting, number of time switched out and scheduled etc. Further, information related to time spend in user-mode and in kernel-mode for this process is stored, size of active memory, size of virtual memory and several such parameters that characterize execution of the process.

- Event information
  Events can be queued up for processes when a process is not executing, e.g., signals in UNIX-like operating systems, these are events that user-level processes send to each other. Information about these events needs to be stored per process, so that the process can address them as soon as it is scheduled.

2.4 Context switching

A process context switch is a basic building block of an operating system that supports multi-processing. A context switch, changes the *context* of execution across different entities, two different processes or from a process to the kernel. In each of these cases the operating system has to perform a save-and-restore operation—save state of the outgoing process and restore state of the incoming process. The exact nature of the state depends on the entities involved in the switch. For example, with Linux, on a system call, execution switches from the user mode to the kernel mode, where all registers of the user process are maintained and no new state is restored. *Each process has the kernel in it!*[4] A save-and-restore operation across processes involves accessing the process-control-block of each process (to read and to write).

Note that the context switch operation by itself is pure overhead, it does not contribute to the progress of a process. The switch is an OS operation required for process management. A vital issue is the extent of context switching overhead which directly impacts the performance of applications. Most modern systems require to support fairly high context switch rates to support multi-processing models. On x86 machines, a context switch requires access to memory twice (once to write to the PCB and once to read PCB of the incoming process). Switching overhead (or inefficiency) depends on the switching rate and time required to complete each switch. Most modern hardware that uses PCB for save-and-restore, operates in the range of sub-milliseconds. Some architectures support complex instructions that can copy multiple registers in one go, while others may have a context register that points to valid context of each process and a context switch involves updating a single register.

2.5 Process creation

An operating system exists to serve! and the service is requested by and provided for processes (user-level processes). A basic functionality then is to enable creation of work, i.e., instantiation and termination of new processes/applications.

A general strategy for this as follows, the operating system *boots-up*, after loading itself in memory (after intializing devices, doing configuration and verification tasks) and then waits, and waits and waits some more, and will keep waiting till the user (or a user-level program) requests the operating system for services. This is the operating loop for general purpose machines and their corresponding operating systems. Specialized hardware, like embedded systems etc., usually have the operating system and the application rolled-up all in one and once the machine starts, so does the application. Back to general purpose operating systems, and they wait and wait after boot-up to be issued work. To beat this agony, the following approach is used—create a handcrafted first user-level process and

[4] The keen reader will notice that this is a loaded statement, shrouded in some kind of mystery, all to be revealed in good time.

once the boot process is complete, the operating system switches[5] to execution of the first user-level process. Once in user-land, *Voila!* everyone is back in business.

Operating systems typically provide an interface (a system call) that allows a (user) process to create other user processes. Conceptually, the requirement is that of creating new processes and configuring each process to execute the desired programs/applications. Each of these processes will individually depend on OS services and the purpose of operating systems to serve is served! The process that creates a new process is called the *parent* process and the new process is a *child* process of the parent. When a new process is created it can execute in one of the possibilities,

1. The parent and the child process execute the same program and both execute concurrently. For example, a web-server process may periodically create new child process to serve incoming web requests, while simultaneously serving requests itself.
2. The parent creates a child process, and the child process loads a new program. This is the modus operandi of operating systems—setup a special first user-level process, which in turn creates new (child) processes, each of which starts other user-level programs (the login screen, the graphical user interface, initialization programs, etc.).

Unix-like operating systems provide two system calls (and their associated variants) for this purpose—`fork` and `exec`.

fork
The `fork` system call is responsible for the following tasks,

1. *Duplicate* a process by using the process state of the parent/calling process. Creates a new process-control-block and in effect creates a new

exec

1. a

2.6 System calls revisited

Figure 2.3 shows a typical invocation path of a system call. A user-level program intends to perform an action that requires operation system functionality, e.g., read from disk. Assuming a 'C' program, the `fread` API function of the standard C library is inovked, which in turns uses the system call interface to invoke the operating system functional via the system call. The system call corresponding to `fread` is the `read` call, which the C library invokes. On Linux `x86` machines, a system call can be issued by invoking an explicit software interrupt using the instruction `int 0x80`. The instruction results in two things—(i) stops execution of the user-level program, saves it execution context on the stack and switches the mode of execution from the user mode to kernel mode, and (ii) transfers control the generic system call handler.

[5]This first switch from the OS to a process is subtle, most switching to user-mode happens when a user-level program has previously executing and is paused for a switch to kernel-mode. This first switch is unsolicited!

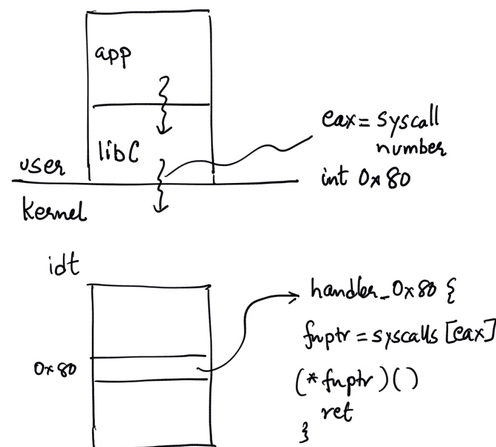

Figure 2.3: System call invocation and operating system actions.

```
// the exit system call
mov eax, 1
mov ebx, 0
int 0x80

// the read system call
mov eax, 3
mov ebx, 1
mov ecx, buf
mov edx, 10
int 0x80
```

Further, the parameters required for system calls are passed using the general purpose registers— `eax, ebx ...`. The two examples show invocation of the `exit` and `read` system calls. Value in the `eax` register is the system call number. This number is used by the generic system call handler of the operating system to determine which system call is being requested. As the Figure 2.3 shows, the system call number can be used as index in a list of function pointers to system calls. The appropriate function is invoked and kernel then returns to user-mode, with the return value stored in the `eax` register.

2.7 Process scheduling

This will discussed as a separate chapter.

2.8 Food for thought

1. Browseable and searchable Linux source can be found here: `http://elixir.free-electrons.com/linux/latest/source`. Dive in.

2. What is the Linux (struct) variable that stores process state/information? Pick your favorite five member variables of the structure.

3. Referring to Figure 2.1, what does each field/column mean?

4. If you look up the memory map of different processes (similar to the example in Figure 2.1) you will notice that the memory regions for different processes overlap or are the same. What is going on? What happened to the whole isolation story? and How are things even working correctly? ... are you loosing sleep yet☺, you should!

5. Execute the following command on a Linux machine: `ps -lax`.
   Which process has pid 2? Is this process different from other processes?

6. Search for the files `syscall_32.tbl` and/or `syscall_64.tbl` in the Linux kernel source to see a list of system calls, their numbers/index and entry points (functions names in the kernel source).

Chapter 3

mmm . . . memory