OS Concepts 1.1: What Operating Systems Do

- **What is an OS?:** OS's vary widely in design and in function, but basically an OS is the software that sits between application programs and computer hardware. It provides an environment in which application programs are executed, by allocating physical resources like CPU, memory, and I/O devices.

OS Concepts 1.2: Computer-System Organization

1.2.1: Interrupts

- **Device Controller:** The I/O managing processor within a device. Basically the hardware, that the device driver talks to, which controls the I/O device.
- **Device Driver:** A software component in the OS that understands how to communicate with its respective device controller and manages I/O to those devices.
- **Interrupts:** Interrupts are used in OS's to handle asynchronous events originating from outside the processor (interrupts originating from within the processor are called exceptions). Device controllers and hardware faults raise interrupts. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is necessary for good system performance.
- **Interrupt Vector:** A table of pointers stored in low memory that holds the addresses of the interrupt service routines.
- **Basic Interrupt Implementation:** The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction. When the CPU detects a device controller has asserted a signal on the wire, it reads the interrupt number and jumps to the respective interrupt-handler routine by using the interrupt number as an index into the interrupt vector. It then saves the current state of whatever was interrupted, and starts execution of the interrupt-handler routine. Once the handler is finished executing, it performs a state restore and returns the CPU to the execution state prior to the interrupt.
- **Interrupt Terminology:** We say that the device controller **raises** an interrupt by asserting a signal on the interrupt request line, the CPU **catches** the interrupt and **dispatches** it to the interrupt handler, and the handler **clears** the interrupt by servicing the device.
- **More Sophisticated Interrupt Implementation:** We need the ability for the following:
  - Defer interrupt handling during critical processing
  - Efficiently dispatch to the correct interrupt-handler without having to first poll all devices to see which one raised the interrupt

  - Multilevel interrupts, so that the OS can distinguish between high and low priority interrupts and respond with the appropriate level of urgency
  - A way for an instruction to get the OS's attention directly (separately from I/O requests), for activities such as page faults and errors such as division by zero. This task is accomplished by "traps".

  To do this, most CPU's have two interrupt request lines: one is the nonmaskable interrupt, which is used for events such as unrecoverable memory errors, and the second is the maskable interrupt, which the CPU can turn off before the execution of critical instruction sequences that must not be interrupted. Device controllers use the maskable interrupt to request service.

1.2.2: Storage Structure

- **Firmware:** Software stored in ROM or EEPROM for booting the system and managing low level hardware.

1.2.3: I/O Structure

- **Direct Memory Access (DMA):** Interrupt-driven I/O as described in section 1.2.1 is fine for moving small amounts of data but can produce high overhead when used for bulk data movement, like when moving data to and from nonvolatile memory. DMA is used to avoid this overhead. The device controller sets up buffers, pointers, and counters for its I/O device, and transfers entire blocks of data to or from the device and main memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. The CPU is able to perform other work while the device controller is performing these operations.

OS Concepts 1.3: Computer-System Architecture

1.3.1: Single-Processor Systems

- **CPU:** The hardware that executes instructions.
- **Processor:** A physical chip that contains one or more CPU's.
- **CPU Core:** The core is the component of the CPU that executes instructions and contains registers for storing data locally.
- **Single-Processor System:** A computer system with a single processor containing one CPU with a single processing core. These systems often also have other special-purpose processors as well, such as disk, keyboard, and graphics controllers. These special-purpose processors run a limited instruction set and do not run processes;

their use is incredibly common and does not turn a single-processor system into a multiprocessor system.

1.3.2: Multiprocessor Systems

- **Multiprocessor Systems:** A computer system containing multiple processors (figure 1). Traditionally contains two or more processors, each with a single-core CPU.

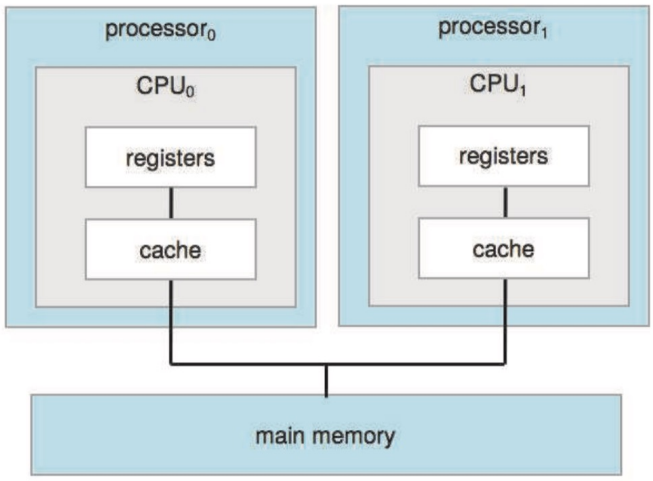

Figure 1: Symmetric *multiprocessing* architecture

- **Multiprocessor Advantages (Increased Throughput):** Primary advantages of multiprocessor systems is increased throughput. The speed-up ratio with N processors is not N, however; it is less than N because there is overhead incurred and contention for shared resources when dealing with multiple processors.

- **Multicore Systems:** A computer system containing multiple cores on the same processor chip (figure 2). Such systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. Additionally, one chip with multiple cores uses significantly less power than multiple single-core chips, an issue especially important for mobile devices.

- **Multiprocessor Bottleneck:** Adding additional CPU's to a multiprocessor system increases computing power, but does not scale very well. Once too many CPU's are added, contention for the system bus becomes a bottleneck and performance begins to degrade.

- **Non-uniform Memory Access (NUMA):** To avoid bottleneck performance degradation arising from system bus contention, we can provide each CPU with its own local memory that is accessed via a small and fast local bus (figure 3). The CPU's are connected by a shared system interconnect, so that all CPU's share one

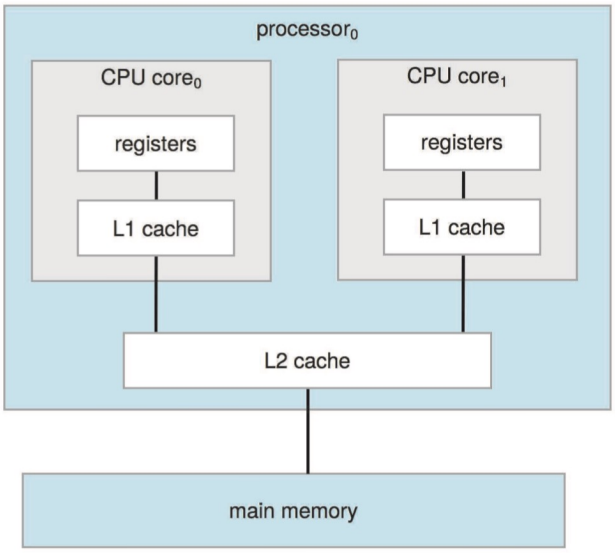

Figure 2: *Multicore* architecture

physical address space. The advantage is that when a CPU accesses its local memory, not only is it fast, but there is also no contention over the system interconnect. Thus, NUMA systems can scale more effectively as more processors are added.

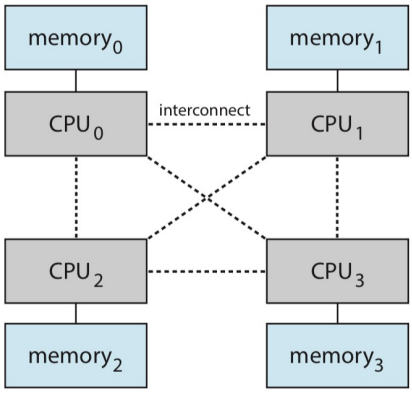

Figure 3: NUMA architecture

- **NUMA Drawbacks (Increased Latency):** A potential drawback is increased latency when a CPU must access remote memory across the system interconnect (accessing the local memory of another CPU). OS's can minimize this NUMA penalty through careful CPU scheduling and memory management.

OS Concepts 1.4: Operating-System Operations

- **Bootstrap Program:** The initial program that is run when a computer starts running for the first time. Typically a very simple program and stored in firmware. The program must know how to load the OS kernel into memory and start executing it.
- **System Daemons:** Services provided outside of the kernel that are loaded into memory at boot time, which run the entire time the kernel is running.

1.4.1: Multiprogramming and Multitasking

- **Multiprogramming:** Increase CPU utilization by organizing programs so that the CPU always has one to execute. Execute a process until it needs to wait on some task like I/O, then switch to another process. Keep switching between processes such that the CPU is never idle.
- **Process:** In a multiprogrammed system, a program in execution is termed a process.
- **Multitasking:** The logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them incredibly frequently. This is how interactive I/O like keyboard input works: rather than letting the CPU sit idle in the time between the keystrokes of the user, the OS will rapidly switch to another process in the meantime.

1.4.2: Dual-Mode and Multimode Operation

- **Modes of Execution:** A properly designed OS must ensure that an incorrect (or malicious) program cannot cause other programs - or the OS itself - to execute incorrectly. To do this, we need to distinguish between the execution of OS code and user-defined code. Most computer systems provide hardware support that allows differentiation among various modes of execution.
- **User Mode and Kernel Mode:** At the very least, we need two separate modes of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). Whenever the OS gains control of the computer, it is in kernel mode. The system always switches back to user mode before passing control to a user program. The concept of modes can be extended beyond two modes (e.g. the four protection rings in Intel processors).
- **Privileged Instructions:** Some machine instructions that may cause can only be executed in kernel mode. The instruction to switch to kernel mode is an example of a privileged instruction.

1.4.3: Timer

- **Timer:** We must ensure that the OS maintains control over the CPU: we allow user programs to execute, but they must eventually relinquish control to the OS (avoid situations like user program infinite loops or not calling system services and thus not returning control to the OS). To accomplish this, we can use a timer that is set to raise an interrupt after a specified amount of time. If the timer interrupts, control transfers automatically to the OS, which may treat the interrupt as a fatal error or may give the program more time. Instructions that modify the timer are clearly privileged.

OS Concepts 1.5: Resource Management

The OS can be seen as a resource manager. The following are things that the OS must carefully manage in a computer system.

1.5.1: Process Management

- **Program vs Process:** A program by itself is not a process. A program is a *passive* entity, whereas a process is an *active* entity. Remember that a process is just a program in execution, thus there can be multiple processes associated with the same program (and each is considered a separate execution sequence).

1.5.2: Memory Management

- **Memory Management:** The OS is responsible for keeping tack of which parts of memory are currently being used and which process is using them, allocating and deallocating memory, and deciding which processes (or parts of processes) and data to move into and out of memory.

1.5.3: File-System Management

- **File System:** The OS abstracts the physical properties of its storage devices to define a logical storage unit, the **file**. In other words, the OS implements the abstract concept of a file by managing mass storage media and the devices that control them.

1.5.4: Mass-Storage Management

- **Secondary Storage Management:** The proper management of secondary storage is critical to a computer system. The OS must take care of things such as mounting and unmounting, free-space management, storage allocation, disk scheduling, partitioning, and protection.

1.5.5: Cache Management

- **OS and Memory Hierarchy:** The OS is responsible for moving data between the different levels of the memory hierarchy that it has access to. The OS can only manipulate software-controlled caches, for instance transfer of data from disk to memory, while data transfer from CPU cache to registers is a hardware function.
- **Caches and Multitasking:** In a computing environment where only one process executes at a time, having the same data appear in multiple levels of the memory hierarchy is not an issue, since access to desired memory always will be to the copy at the highest level of the hierarchy. In a multitasking environment, however, extreme care must be taken to ensure that if several processes wish to access the same data, each of these processes obtains the most recently updated value of the data.
- **Caches and Multiprocessor Systems:** In a multiprocessing environment, not only do we need to make sure that processes access the most recenly updated value of the desired data (multitaking), but we now have CPUs that contain local caches in which data may esist simultaneously in several of these. We need to make sure that an update to the value of a given piece of data in one cache is immediately reflected in all other caches where this data resides. This issue if called *cache conherency*, and is usually handled in hardware (below the OS level).

1.5.6: I/O System Management

- **I/O Subsystem:** One of the purposes of an OS is to hide the peculiarities of specific hardware devices from the user. Often, these peculiarities are hidden from most of the OS itself by the I/O subsystem. Device drivers for specific hardware devices for instance are included in the I/O subsystem.

OS Concepts 1.7: Virtualization

- **Virtualization:** Virtualization allows us to abstract the hardware of a single computer (the CPU, memory, disk drives, etc.) into several different execution environments, creating the illusion that each separate environment is running on its own private computer. An OS that is natively compiled for a particular CPU architecture runs within another OS also native to that CPU.
- **Emulation:** Simulates computer hardware in software, typically used when the source CPU type is different from the target CPU type (e.g. Apple Rosetta when moving from PowerPC to x86). Usually much slower than native code.

OS Concepts 1.10: Computing Environments

1.10.4: Peer-to-Peer Computing

- **Peer-to-Peer (P2P) Computing:** In this distributed computing model, clients and servers are not distinguished from each other. Each node in the system may act as either a client or a server, depending on whether it is requesting or providing a service. In traditional client-server systems, the server is a bottleneck; but in a P2P system, services can be provided by several nodes distributed throughout the network.
- **Centralized P2P:** When a node first joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- **Decentralized P2P:** A decentralized system uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network.

OS Concepts 2.6: Why Applications are OS Specific

- **Why Applications are OS Specific:** Each OS exposes different functionalities (system calls), so applications cannot expect to be able to use the same functions across varying OS's. Even if system calls were somehow uniform, other barriers would still pose a challenge: binary formats, varying CPU ISA's, system call discrepencies (specific operands and operand ordering, how to invoke syscalls, syscall result meanings, etc.).
- **How to Make Applications Cross Compatible Across OS's:**
  - Write the application in an *interpreted language* (e.g. Python or Ruby); performance typically suffers. and interpreter usually only offers a subset of the OS's features.
  - Write the application in a language that includes a *virtual machine* (e.g. Java). The JVM has been ported to many OS's, and in theory any Java app can run within the JVM wherever it's available. Usually have similar disadvantages as with interpreted languages.
  - Use a language that compiles *machine and OS specific binaries* (e.g. C++ or Rust), and simply port to each OS on which it will run. Standard API's like POSIX can make this process easier.

OS Concepts 2.7: OS Design and Implementation

2.7.1: Design Goals

- **OS Goals and Specifications:** The first problem in designing a system is to define goals and specifications. These requirements can, however, be divided into two basic groups: user goals and system goals.

2.7.2: Mechanisms and Policies

- **Mechanisms and Policies:** An important principle is the separation of policy from mechanism. Mechanisms determine *how* to do something; policies determine *what* will be done. For instance, the standard Linux kernel has a specific CPU scheduling algorithm, which is a mechanism that supports a certain policy. However, anyone is free to modify or replace the scheduler to support a different policy.

OS Concepts 2.8: OS Structure

2.8.1: Monolithic Structure

- **Monolithic Structure:** A monolithic kernel places all of the functionality of the kernel into a single, static binary file that runs in a single address space. Everything below the system-call interface and above the physical hardware is the kernel, as seen in figure 4. Typically difficult to implement and extend, but have good performance due to very little overhead in the syscall interface, and communication within the kernel is fast.

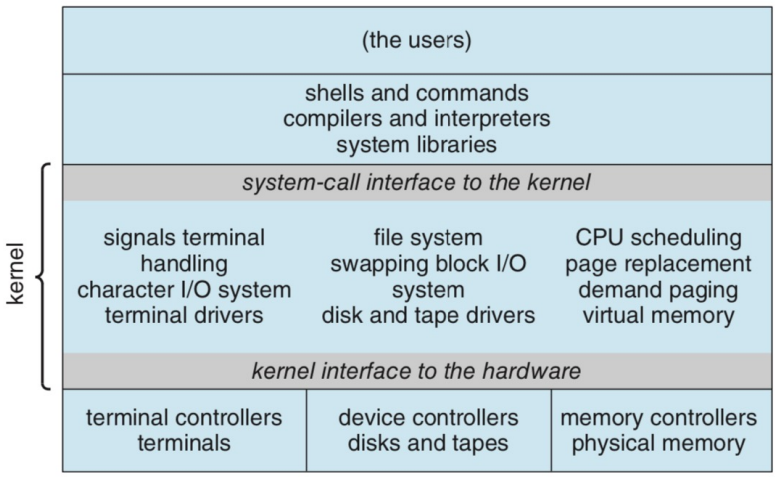

Figure 4: Traditional UNIX system structure (monolithic architecture)

2.8.3: Microkernels

- **What is a Microkernel:** The microkernel approach structures the OS by removing all nonessential components from the kernel and implementing them as user-level programs that reside in separate address spaces, resulting in a smaller kernel, seen in figure 5. There is little consensus on what services remain in the kernel, however, typically minimal process and memory management and a communication facility are provided. The main function of the microkernel is to provide communication between the client program and the various services also running in user space; communication is provided though message passing.

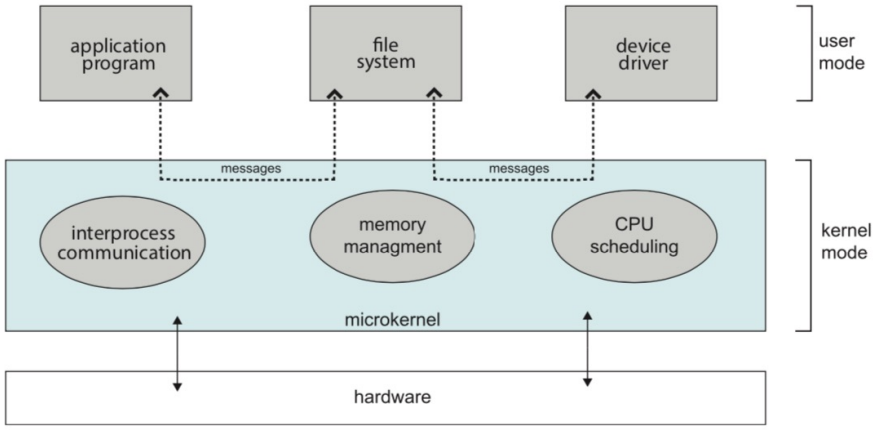

Figure 5: Architecture of a typical microkernel

- **Microkernel Benefits:** The microkernel approach makes extending the OS easier, as all new services are added to user space and do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer as the kernel is smaller. It also makes porting the OS to different hardware easier and provides more security and reliability, as most services are running as user -rather than kernel- processes.

- **Microkernel Drawbacks:** Performance of microkernels can suffer due to increased system-function overhead. The overhead involved in copying messages and switching between processes has been the larged impediment to the groth of microkernel-based OS's.

2.8.4: Modules

- **Loadable Kernel Modules (LKMs):** Using LKMs, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. The key idea is for the kernel to provide core services, while other services are implemented dynamically, as the kernel is running.

OS Concepts 2.9: Building and Booting an OS

2.9.2: System Boot

- **Booting OS:** When starting a computer system, how does the hardware know where the kernel is or how to load that kernel? This process of loading the kernel is known as booting the system. The boot process typically roughly follows as so:

  1. A small piece of code known as the bootstrap program or boot loader locates the kernel
  2. The kernel is loaded into memory and started
  3. The kernel initializes the hardware
  4. The root file system is mounted

- **BIOS:** Some (typically older) computers use a multistage boot process: when the computer first powered on, a small boot loader located in nonvolatile firmware known as BIOS is run. This initial bootloader usually does nothing more than load a second boot lader, which is located at a fixed disk location called the boot block, which is then responsible for loading the OS into memory and begin execution.

- **UEFI:** More recent computers have replaced the BIOS-based boot process with UEFI (Unified Extensible Firmware Interface). The biggest difference is that UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.

OS Concepts 3.1: Process Concept

OS Concepts 3.1.1: The Process

- **What is a process?** A process is a program in execution. A program becomes a process when an executable file is loaded into memory. Although two processes may be associated with the same program, they are nevertheless considered two separate esecution sequences.

- **How is the status of a running process represented?** The status of the current activity of a process is represented by the value of the program counter and the contents of the processor's registers (the preservation of a processes memory address space depends on the memory management method of the OS).

- **What does the memory layout of a process look like?** The memory layout of a process is typically divided into multiple sections (figure 6), the most important being:

  – **Text** - the executable code

  – **Data section** - global variables

- **Heap section** - memory that is dynamically allocated during program runtime
- **Stack section** - temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

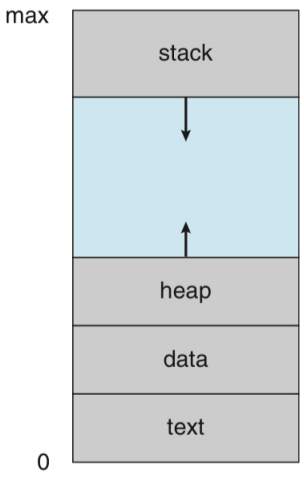

Figure 6: Memory layout of a process

OS Concepts 3.1.2: Process State

- **What are the different states a process can be in?** In general, a process may be in one of the following states (diagram in figure 7):
  - **New:** The process is being created.
  - **Running:** Instructions are being executed.
  - **Waiting:** The process is waiting for some event to occur (such as an I/O completion).
  - **Ready:** The process is waiting to be assigned to a processor.
  - **Terminated:** The process has finished execution.

- **How many processes can be running on a processor core at any given time?** Only one process can be actively running (in the running state) on any processor core at any given time. Many processes may be in the ready or waiting state, however.

OS Concepts 3.1.3: Process Control Block

- **How is information about a process tracked by the OS?** The OS maintains information about each process in a data structure called the process control block (PCB), shown in figure 8. Basically it contains all the necessary information required to start, or restart, a process, along with bookkeeping data. On systems

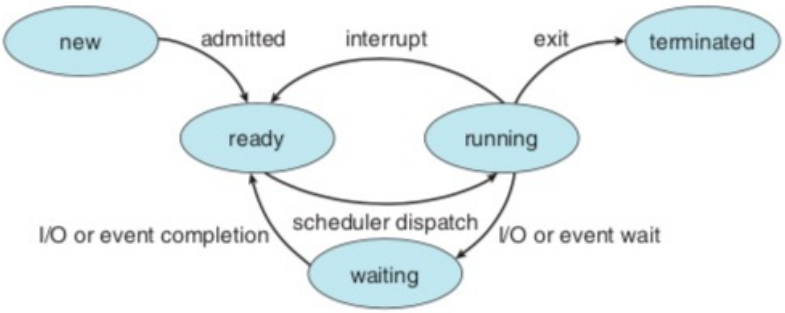

Figure 7: Diagram of process states

that support threads, the PCB is expanded to include information for each thread.

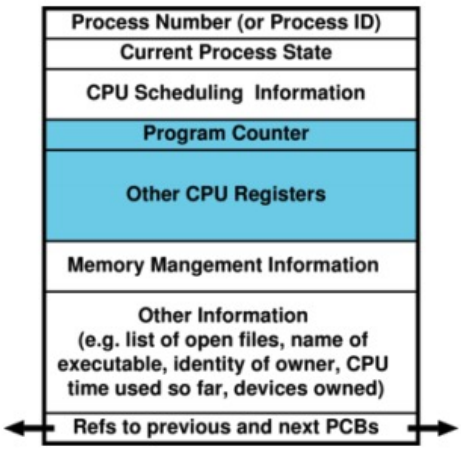

Figure 8: The process control block. The highlighted blue regions is the machine environment during the time the process is actively in control of the CPU.

OS Concepts 3.2: Process Scheduling

- **How are processes selected to be run?** To maximize CPU utilization by having some process running at all times and to support efficient multitasking by switching among processes frequently enough, the **process scheduler** is responsible for selecting an available process for program execution on a CPU core (balancing multiprogramming and time sharing). For a single core system there will never be more than one process running at a time, whereas a multicore system can run multiple processes at a time.
- **Degree of multiprogramming:** The number of processes currently in memory.

- **I/O Bound:** An I/O bound process is one whose speed is bound by the I/O (i.e. it doesn't matter if the CPU is blazing fast, the program speed will still be limited by how fast I/O completes).
- **CPU Bound:** A CPU bound process is one whose speed is limited by the speed of the CPU (i.e. it would go faster if the CPU could go faster).

OS Concepts 3.2.1: Scheduling Queues

- **What happens to a process when it first enters the system?** As processes enter the system, they are put into a **ready queue**, where they have the "ready" state and waiting to be executed (dispatched) on a CPU's core. This queue is typically implemented as a linked list, containg pointers to PCB structures. See figure 9.
- **What happesn to a process when it is waiting?** A process in a waiting state (waiting for I/O, time slice expired, etc.) is put into the **wait queue**. See figure 9.

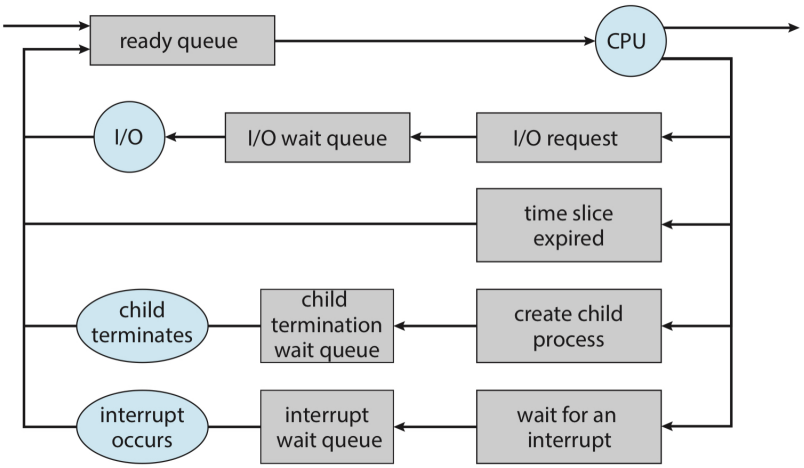

Figure 9: Representation of process scheduling

OS Concepts 3.2.3: Context Switch

- **What happens when the CPU needs to switch to a different process?** When switching to another process, whether that be because of an interrupt, time slice expired, or any another reason, the current context of the running process on the CPU core needs to be saved (state save). The context is represented in the PCB of the context. The CPU then needs to restore the context of the other process it wishes to switch to (state restore). This process is known as a **context switch**.
- **What are the performance implications of a context switch?** A context switch is pure overhead. The system does no useful work while switching processes.

A typical context switch takes several microseconds, but is highly dependent on the hardware the OS is running on.

OS Concepts 3.3: Operations on Processes

OS Concepts 3.3.1: Process Creation

- **How are processes created?** During the course of execution, a process may create several new processes. The creating process is called the **parent** process, and the new processes it has created are aclled the **children** of that process. Each of these new processes may in turn create other processes, forming a hierarchical tree.
- **How are processes identified?** Most OS's use unique process identifiers (PID, typically an integer) to identify processes. The PID provides a unique value for each process in the system.
- **What is the ultimate parent process (chicken and egg problem)?** In a Linux OS, the `systemd` process (which always has a PID of 1) serves as the root process for all user processes. It is the first user process created when the system boots. Once the system has booted, the `systemd` process creates processes that provide additional services, like an `ssh` server or a login server.
- **How are resources and initialization of processes handled?** In general, child processes require resources (CPU time, memory resources, I/O devices, files) to accomplish its task. There are several options how this is handled:
  - Parent and child share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- **How are parent and child processes executed?** The parent process may continue to execute **concurrently** with its children, or it may **wait** until some or all of its children have terminated.
- **What are the address-space possibilities for the new child process?** The child process is a **duplicate** of the parent process (it has the same program and data as input), or the child process has a program **loaded** into it.
- **What does process creation on a UNIX system typically look like?** Diagram in figure 10. The parent process calls `fork()`, at which both the parent and child process continue execution (of the same program). After a `fork()` call, one of the processes typicall uses the `exec()` system call to replace the process's memory space with a new program. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a `wait()` system call to move itself off the ready queue until the termination of the child.

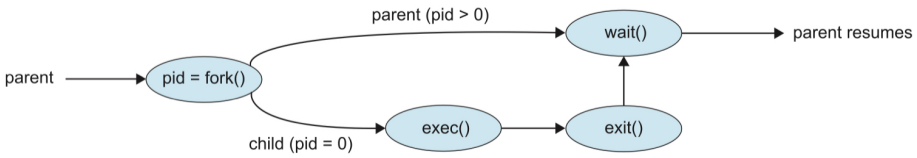

Figure 10: Representation of typical process creation on a UNIX system

OS Concepts 3.3.2: Process Termination

- **What happens when a process is terminated?** When a process is terminated, all the resources of the process (including virtual and physical memory, open files, and I/O buffers) are deallocated and reclaimed by the OS.
- **Under what situations can a process be terminated?** A process can be terminated under three circumstances:
  - Process executes last statement and asks the OS to delete it (`exit()` syscall, either called explicitly or implicitly by C run-time library)
  - Process performs an illegal operation (e.g. access memory it is not authorized for or execute a privileged instruction)
  - Parent may terminate child process (child process exceeding resource usage, task assigned to child no longer required, or parent is exiting and OS uses cascading termination - if a process terminates so must all its children)

OS Concepts 3.4: Interprocess Communication

- **How do processes communicate with each other?** Using interprocess communication (IPC): implemented using either shared memory or message passing.

OS Concepts 3.5: IPC in Shared-Memory Systems

- **How does shared memory IPC work?** Normally, the OS tries to prevent processes from accessing memory of other processes. Shared memory IPC requires that two or more processes agree to remove this restriction, so that they can then exchange information by reading and writing data in the shared memory areas.
- **How is shared memory access coordinated among processes?** Processes must themselves ensure that they are not writing to the same location simultaneously.

OS Concepts 3.6: IPC in Message-Passing Systems

- **What is message passing IPC?** Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing

the same address space (particularly useful in distributed environments). Message passing may be either **blocking** or **nonblocking** (also known as synchronous or asynchronous).

OS Concepts 3.7: Examples of IPC Systems

OS Concepts 3.7.4: Pipes

OS Concepts 3.7.4.1: Ordinary Pipes

- **How do ordinary pipes work for IPC?** Ordinary (in contrast to named) pipes allow two processes to coomunicate in standard producer-consumer fashion: the producer writes to the write end of the pipe, and the consumer reads from the read end of the pipe. As a result, ordinary pipes are unidirectional (one-way communication), and bidirectional communication requires two pipes.
- **What processes can access an ordinary pipe?** On UNIX systems, ordinary pipes cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process it created using `fork()`. In UNIX systems a child process inherits open files from its parent, hence why it is allowed to access the pipe. This also implies that ordinary pipes can only be used for communication between processes on the same machine.

OS Concepts 3.7.4.2: Named Pipes

- **What is difference between ordinary and named pipes?** Named pipes provide bidirectional communication, and do not require a parent-child process relationship. Once a named pipe is established, sefveral processes can use it for communication. Named pipes also continue to exist after communicating processes have finished, and must explicitly be deleted.

OS Concepts 4.1: Threads Overview

- **What are threads?** A thread is a basic unit of CPU utilization. Processes can have multiple threads of control to perform more than one task at a time (threads exist as subsets of a process).
- **What is a thread comprised of?** A thread is comprised of a thread ID, a program counter (PC), a register set. and a stack.
- **What parts of a process do threads share?** Threads technically share everything belonging to a process, but each thread has it's own independent stack (and of course corresponding registers and a program counter, and possibly thread local storage if supported). Basically each thread contains its own state. Figure 11 shows a rough visualization of how threads operate in the context of processes. Figure 12 shows how multiple threads are laid out in the memory of a process.

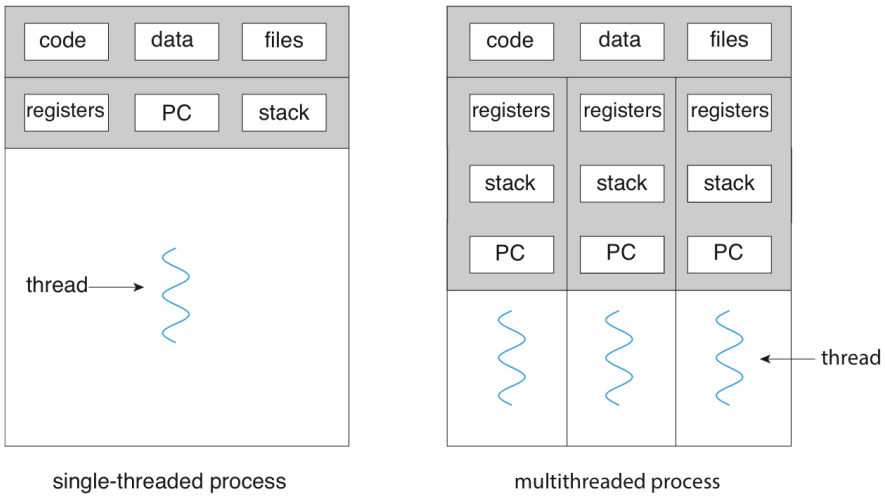

Figure 11: Rough visualization of single vs multithreaded processes

- **What is the high level difference between threads and processes?** Processes help organize two independent concepts in computing: resource grouping and execution. Resources (program text, data, open files, child processes, etc.) are grouped into a single process so that they can be managed easily. The other concept of a process is a thread of execution. Threads have program counters keeping track of which instructions to execute next, registers to hold its current working variables, and its own stack. Threads only operate in the context of processes. Processes are used to group resources together, while threads are entities scheduled for execution on the CPU.
- **Why might one want to use threads instead of processes?** Process creation is time consuming and resource intensive: it is generally more efficient to use one process that contains multiple threads. It makes particular sense to use threads when multiple tasks need to be performed at the same time using the same resources.

OS Concepts 4.2: Multicore Programming

- **What is multicore programming?** With the advent of multicore computer systems, multithreading provides a mechansim for more efficient use of thse multiple computing cores and improved concurrency.
- **What is the difference between concurrency and parallelism?** Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (figure 13). On a system with multiple cores, however, concurrency means that some threads can run in parallel, because the system can assign a separate thread to each core (figure 14). Thus, it is possible to have concurrency without parallelism.

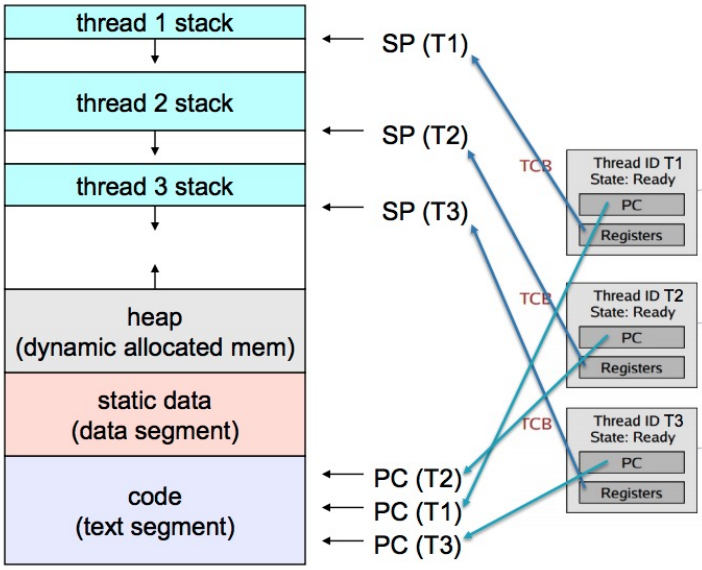

Figure 12: Memory layout of a process with multiple threads. Stack pointers and program counters are tracked in the respective thread control blocks (TCB)

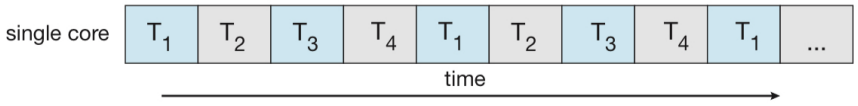

Figure 13: Concurrent execution on a single core system

- **Concurrency:** Multiple threads making progress.
- **Parallelism** Multiple threads making progress *simultaneously*.
- **What are the two types of parallelism?**
  - **Data Parallelism:** Data parallelism focuses on distributing *subsets* of the *same data* across multiple computing cores and *performing the same operation* on each core.
  - **Task Parallelism:** Distributing not data but *tasks* (threads) across multiple computing cores. Each thread is performing a *unique operation*.
- **Amdahl's Law:** The main idea behind Amdahl's law is that when we speed up one part of a system, the effect on the overall system performance depends on both what proportion of the overall system this part was and how much it sped up. We can interpret this in the context of multicore programming if we consider parallel components of a program being sped by up the number of processor cores N. The key takeaway is that to significantly speed up an entire system, we must improve the speed of a very large fraction of the overall system. Derivation taken from CS:APP p22.

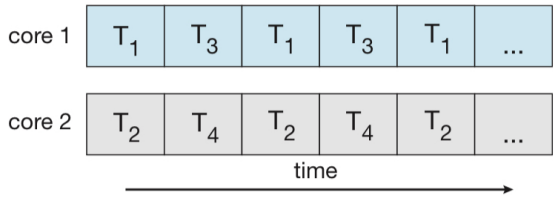

Figure 14: Parallel execution on a multicore system

- **Derivation:** Consider a system in which executing some application requires time T_{old}. Suppose some part of the system requires a fraction α of this time, and that we improve its performance by a factor of k. That is, the component originally required time αT_{old}, and now it requires time $(\alpha T_{\text{old}})/k$. The overall execution time would thus be

$$\begin{aligned} T_{\text{new}} &= (1-\alpha)T_{\text{old}} + (\alpha T_{\text{old}})/k \\ &= T_{\text{old}}[(1-\alpha) + \alpha/k] \end{aligned}$$

  From this, we can compute the speedup $S = T_{\text{old}}/T_{\text{new}}$ as

$$S = \frac{1}{(1-\alpha) + \alpha/k}$$

- **When N approaches infinity:** If the speedup $k = N$ (number of processor cores in multicore context) approaches infinity, the speedup converges to $\frac{1}{(1-\alpha)}$. If two thirds of a program must be performed serially (parallel portion $\alpha = 1/3$), then the maximum speedup is 1.5 times, regardless of the number or processing cores N we add. It pays to be aware of exactly how much a program can really be sped up using parallelism (not much point if much of it needs to be done serially).

OS Concepts 4.3: Multithreading Models

- **What is the difference between user and kernel threads?** User threads are implemented in userspace and managed without knowledge of the kernel (and neither does the kernel know about the user threads). Kernel threads are managed directly by the OS. User threads need to eventually map down somehow to kernel threads (important to take advantage of multicore systems, as user threads alone do not inherently support parallelism), different such multithreading models. Typically slightly more overhead when switching between kernel threads (user theads do not need to dip down to the kernel at all).

- **Many to One Model:** The many to one model maps many user level threads to one kernel thread (figure 15). Essentially all the kernel sees is a single process, the

threads are managed by a user-space thread library. Since this thread management is done by the thread library in user-space, it is very efficient (does not need to dip down into kernel, switching between kernel theads invoke expensive context switch costs). However, the entire process will block if a thread makes a blocking system call (again, because all the kernel sees is a single process). Also, since only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

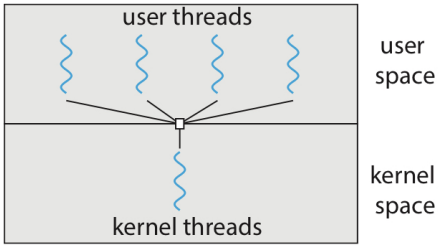

Figure 15: Many to one threading model

- **One to One Model:** The one to one model maps each user thread to a kernel thread (figure 16). It provides more concurrency than the many to one model by allowing another thread to run when a thread makes a blocking system call, as the kernel is responsible for managing these threads at the end of the day. Also allows multiple threads to run in **parallel** on multiprocessors. The only drawback to this model is that there is slightly more overhead involved with creating a kernel thread than just a pure user thread.

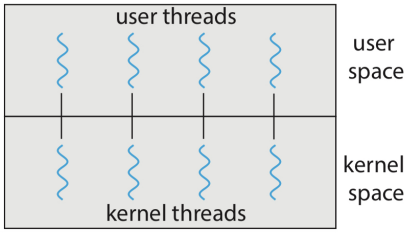

Figure 16: One to one threading model

- **Many to Many (M:N) Model:** The M:N model *multiplexes* M user level threads to N kernel threads, where $M \geq N$ (figure 17). In this model, the developer may create as many user theads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can simply schedule another thread for execution. Complex to implement, efficient performance requires extensive coordination between the userspace scheduler and the kernel scheduler.

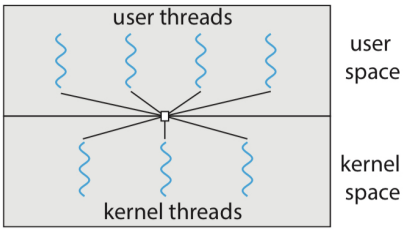

Figure 17: Many to many threading model

OS Concepts 4.4: Thread Libraries

- **What are the two general strategies for creating multiple threads:** Asynchronous threading and synchronous threading.
- **Asynchronous Threading:** With asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently of on another.
- **Synchronous Threading:** With synchronous threading, the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes. The threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed.

OS Concepts 4.5: Implicit Threading

- **What is implicit threading?** Implicit threading is basically transferring the responsibility of manually managing low level threads from developers to compilers and run-time libraries. Essentially, developers identify *tasks* that they wish to execute concurrently, and these compilers and libraries do the grunt work of managing threads.
- **What are the benefits of using thread pools?** Thead pools offer these benefits:
  - **Avoiding thread creation overhead:** Servicing a thread with an existing thread is often faster than waiting to create a thread.
  - **Enforcing an upper bound to the number of threads:** A thread pool limits the number of threads that exist at any one point, important on systems that cannot support a large number of concurrent threads.
  - **Moving from thread based to task based concurrency approach:** By using a thread pool, we can separate the idea of executing a task concurrently from the mechanics of creating such a task (threading). Basically lean on compilers and libraries to do threading grunt work (e.g. rayon and TBB).
- **How do thread pools work?** The general idea behind a thread pool is to create a number of threads at start-up and place them into a pool, where they sit and wait

for work. Users can submit task requests, and if there is an available thread in the pool, it is awakened, and the request serviced immediately. If the pool contains no available thread, the task is queued until one becomes free. Once a thread completes its service, it returns to the pool and awaits more work.

- **How do we determine the number of threads in a thread pool?** The number of threads in a pool can be heuristically set based on factors such as the number of CPUs, the amount of physical memory, and the expected number of concurrent client requests. More sophisticated thread pools can dynamically adjust the number of threads according to usage patterns (e.g. Grand Central Dispatch: makes pool smaller when system load is low, thereby consuming less memory).

OS Concepts 4.6: Threading Issues

OS Concepts 4.6.1: The `fork()` and `exec()` System Calls

- **How do the semantics of `fork()` change in a multithreaded program?** If one thread calls `fork()`, does the new process duplicatre all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of `fork()`, one that duplicates all threads and another that duplicates only the thread that invoked the `fork()` system call.

- **How do the semantics of `exec()` change in a multithreaded program?** If a thread invokes `exec()`, the program specified in the parameter to `exec()` will replace the entire process - including all threads.

- **Which version of `fork()` to call in a multithreaded program?** If `exec()` is called immediately after forking, then duplicating all threads is unnecessary, as the new program specifiedto `exec()` will replace the whole process anyways.

OS Concepts 4.6.2: Signal Handling

- **What is a signal (in UNIX systems)?** A signal is used in UNIX systems to notify a process that a particular event has occurred, and may be received either synchronously or asynchronously. Synchronous signals are delivered to the same process that performed the operation that caused the signal (examples: illegal memory access and division by 0). When a signal is generated by an external event to a running process, that process receives the signal asynchronously (examples: terminated process with CTRL-C and having a timer expire).

- **How are signals handled?** Every signal has a default signal handler that the kernel runs, but can be overridden by user-defined signal handlers.

- **How are signals delivered to processes?** By using the UNIX function `kill()`.

- **How are signals delivered to multithreaded programs?** Synchronous signals are sent to the thread that caused the signal, and not to other threads in the process.

Asynchronous signals are typically delivered only to the first thread found that is not blocking it (theads can choose which signals to accept and which to block).

OS Concepts 4.6.3: Thread Cancellation

- **What is thread cancellation?** Terminating a thread before it has completed.
- **What are the two types of thread cancellation?**
  - **Asynchronous Cancellation:** One thread immediately terminates the target thread to be cancelled. This is troublesome when threads are cancelled in situations where resources have been allocated to these threads, or when a thread is cancelled in the midst of updating data shared with other threads.
  - **Deferred Cancellation:** The target thread periodically checks whether it should terminate, allowing it to terminate itself in an orderly fashion (clean up and release resources).

OS Concepts 4.6.5: Scheduler Activations

- **How are M:N theading models typically implemented (high level)?** Such systems usually place an intermediate data structure between the user and the kernel theads, typically known as a **lightweight process** (LWP). To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and the kernel threads are scheduled by the OS to run on physical processors. If a kernel thread blocks (such as when waiting for I/O to complete), the LWP and the corresponding user-level threads attached to the LWP block as well. This is an issue when many concurrent blocking system calls are performed: if five I/O requests are made but only four LWPs are available, the firth request must wait for one of the four LWPs to return from the kernel before executing.

OS Concepts 4.7: OS Thread Examples

- **How are threads and processes created on Linux?** Linux does not distinguish between processes and threads. Linux instead uses the term **task** - rather than process or thread when referring to a flow of control within a program. The Linux `clone()` system call provides the ability to create such tasks. The arguments passed to `clone()` specify the amount of data shared between parent and child tasks, thus making tasks behave more like processes or threads. The `fork()` system call on Linux is typically implemented using `clone()`.

OS Concepts 5.1: CPU Scheduling Basic Concepts

- **What is the motivation for CPU schduling?** In a system with a single CPU core, only one process can run at a time. This means if this single process is in

control of the CPU and is waiting for some I/O request, the CPU is basically sitting doing nothing (which is a waste). With multiprogramming, several processes are kept in memory at a time, and we can swap a new process to do useful work while our original process is waiting for its I/O request to complete. This process of swapping out the tasks the CPU is currently executing is the realm of CPU scheduling.

- **What is the CPU-IO Burst Cycle?** Process execution consist of a cycle of CPU execution and then waiting for I/O (CPU burst and then I/O burst). This cycle is called the CPU-IO burst cycle; processes can be in one of the two states. Processes typically have a large amount of short CPU bursts and a small number of long CPU bursts. This frequency curve is shown in figure 18.

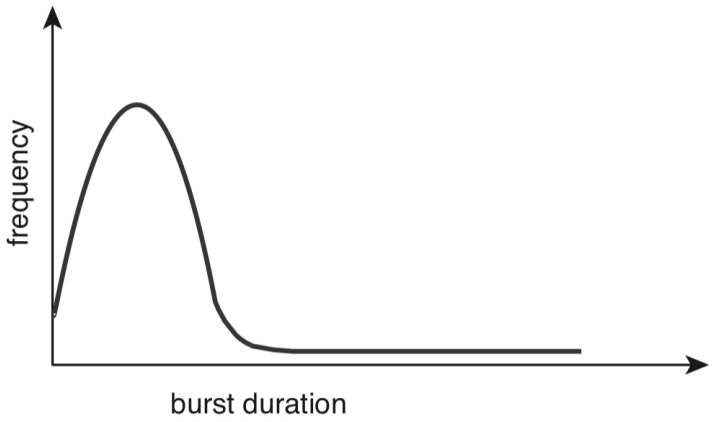

Figure 18: Histogram of CPU burst durations

- **CPU Bound:** CPU bound processes are when the rate of progress for the process is limited by the actual speed of the CPU. The process would go faster with a faster CPU. The opposite of being I/O bound.
- **I/O Bound:** I/O bound processes are when the rate of progress for the process is limited by the speed of the I/O it performs. A faster CPU would not help, only faster I/O completion times would cause the process to proceed faster. The process sits around doing nothing and waiting for I/O to complete (memory wall big issue, disparity between CPU and memory speeds).
- **What has the responsibility of choosing which process to execute on the CPU?** Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed (not this is not necessarily a FIFO queue, various scheduling algorithms have different implementations). This selection process is carried out by the **CPU scheduler**, which selectrs a process from the processes in memory that are ready to execute (in the ready queue) and allocates the CPU to that process. The items in the queues are generally process control blocks (PCBs) of the processes (or TCBs for threads? IDK not sure).

- **Under what circumstances can CPU scheduling decisions take place?** CPU scheduling decisions may take place under the following four circumstances:

  1. When a proecss switches from the running state to the waiting state (e.g. when requesting I/O or willingly switching to the waiting state like calling `wait()` for the termination of a child process)
  2. When a process switches from the running state to the ready state (e.g. when an interrupt occurs)
  3. When a process switches from the waiting state to the ready state (e.g. at completion of I/O)
  4. When a process terminates

- **Premptive vs Cooperative Scheduling:** In situations 1 and 4 above, there is no choice in terms of scheduling: a new process *must* be selected for execution. There is a choice, however, for situations 2 and 3. When scheduling takes place only under situations 1 and 4, scheduling is considered to be cooperative (process willingly gives up control): the process keeps control of the CPU until it releases it either by switching to the waiting state (situation 1) or by terminating (situation 4). When scheduling also occurs in situations 2 and 3, scheduling is considered to be preemptive: the process could keep on running and keep control of the CPU, but instead is kicked off the CPU by the OS. Virtually all modern OS's use preemptive scheduling. Premptive scheduling however can result in race conditions when data is shared among several processes.

- **Why and when does the kernel disable interrupts?** Because interrupts can occur at any time, and because they cannot always be ignored by the kernel, the sections of code affected by interrupts (e.g. atomic operations) must somehow be protected against suddenly losing control of the CPU when an interrupt comes in. This is achieved by simply disabling interrupts at entry and then reenabling interrupts at exit. It is important that these sections of code that disable interrupts do not occur very often and typically contain few instructions. The OS needs to accept interrupts at almost all times, so disabling interrupts should be kept to a minimum.

- **What is responsible for actually giving control of the CPU to processes?** The **dispatcher** is the OS module that gives control of the CPU's core to the process selected by the CPU scheduler. This involves:

  – Performing a context switch from one process to another
  – Switching to user mode
  – Jumping to the proper location in the user program to resume that program

  The dispatcher should be as fast as possible, since it is invoked during every context switch. The time it takes for the dispatcher to stop one process and start another is known as the **dispatch latency** and is illustrated in figure 19.

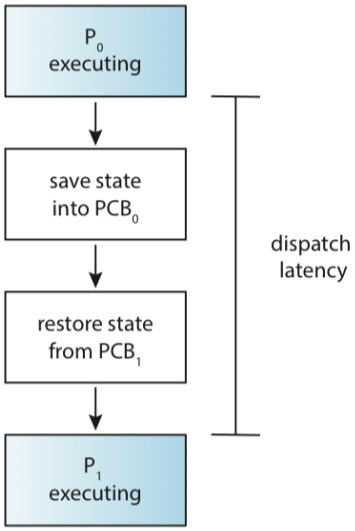

Figure 19: Dispatch latency: the time it takes for the dispatcher to stop one process and perform a context switch to another

OS Concepts 12.1: I/O Overview

- **Accomodating Varying I/O Devices using Device Drivers:** Hardware I/O devices are constantly evolving, and so are methods required to communicate with them. To accomodate this ever increasing variety of I/O devices, the kernel of an OS is structured to use device-driver modules. The device drivers present a uniform device-access interface to the I/O subsystem, much as syscalls provide a standard interface between the application and the OS.

OS Concepts 12.2: I/O Hardware

- **Port:** A device communicates with a computer system via a port.
- **Bus:** If devices share a common set of wires, the connection is called a bus. Uses a rigidly defined protocol that specifies a set of messages that can be sent on the wires. Historically called a data highway. Widely used in computer architecture and vary in signaling methods, speed, throughput, and connection methods.
- **Controller:** A controller is a collection of electronics that can operate a port, a bus, or a device.

12.2.1: Memory-Mapped I/O

- **Processor and Device Communication:** A controller has one or more registers for data and control signals. The processor communicates with the controller by

reading and writing bit patterns in these registers. I/O device control typically consists of four registers: the status, control, data-in, and data-out registers. Data registers are typically 1 to 4 bytes in size, meaning the processor needs to react quickly otherwise new data may overflow the registers and overwrite the old data. Also means I/O is basically done byte by byte.

- **Memory-Mapped I/O:** With memory-mapped I/O, the controller registers are mapped into the address space of the processor. The CPU executes I/O instructions using standard memory data transfer instructions to read and write the controller registers at their mapped locations in physical memory.
- **Memory-Mapped I/O Example:** Memory-mapped I/O for graphics. A thread sends output to the screen by writing data into the memory-mapped region. The controller then generates the screen image based on the contents of this memory. Writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions to read and write byte by byte.

12.2.2: Polling

- **Polling:** Polled mode I/O between a host and a controller is basically where the host has to repeatedly loop to check if the device is ready to read or write by continually checking the controller status register. Polling the controller naturally requires CPU cycles (read a device register, logical-and to extract a status bit, and branch depending on status). Polling becomes inefficient when it is attempted repeatedly yet rarely finds a device ready for service, while other useful CPU processing remains undone.

12.2.3: Interrupts

- **Purpose of Interrupts:** Interrupts are used throughout modern OS's to handle asynchronous events and to trap to supervisor-mode (kernel mode) routines.
- **Interrupt Handlers and OS Boot Time:** At boot time, the OS probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector.
- **Software Interrupts (Traps):** To get the attention of the OS, a special instruction called a trap can be executed. This instruction has an operand that identifies the desired kernel service. Library functions to issue syscalls typically implemented using traps.
- **Interrupt-driven I/O vs Polled I/O:** Interrupt-driven I/O is now much more common than polling, with polling being used for high-throughput I/O. Some device drivers use both: interrupts when the I/O rate is low, and switch to polling when the rate increases to the point where polling is faster and more efficient.

12.2.4: Direct Memory Access

- **Direct Memory Access (DMA):** Using the expensive general purpose CPU processor to watch status bits and to feed data into a controller register one byte at a time (programmed I/O, PIO) seems wasteful. We can avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a DMA controller. Essentially, DMA allows us to bypass the CPU and utilize direct memory to device I/O.
- **DMA High Level Implementation:** Basically, the CPU gives the DMA controller pointers to the source and destination locations and the number of bytes to be transferred. The DMA controller then proceeds to operate the memory bus directly, allowing us to perform I/O without the help of the CPU. When the entire transfer is finished, the DMA controller interrupts the CPU.
- **Cycle Stealing:** When the DMA controller seizes the memory bus, the CPU is momentarily prevented from accessing main memory, although it can still access data items in its caches. Although this cycle stealing can slow down CPU computation, offloading the data transfer work to a DMA controller generally improves the total system performance.

OS Concepts 12.3: Application I/O Interface

- **Accomodating Varying I/O Devices in an OS:** The wide variety of available devices poses a problem for OS implementors (each device has its own set of capabilities, control-bit definitions, and protocols for interacting with the host). How can the OS be designed so that new devices can be attached to the computer without rewriting the OS? And when devices vary so widely, how can the OS give a convenient, uniform I/O interface to applications? Answer: by abstracting I/O hardware with device drivers.
- **Device Drivers:** Device drivers are kernel modules that internally are custom-tailored to specific devices but that export one of the standard OS I/O interfaces. The purpose of the device driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel. Hardware manufacturers can design new devices to be compatible with existing host controller interfaces (such as SATA), or they can write device drivers for popular OS's. Each OS has its own standards for the device driver interface, so they must be ported for each OS.
- **Device Access Conventions:** I/O devices vary among many dimensions, such as synchronous vs asynchronous, sequential or random access, speed of operation, etc. But for the purpose of application access, many of these differences are hidden by the OS, and the devices are grouped into a few conventional types. The major access conventions include: block I/O, character-stream I/O, memory-mapped file access, and network sockets.

12.3.3: Clocks and Timers

- **Programmable Interval Timer:** Most computers have hardware clocks and timers that provide three basic functions: give the current time, give the elapsed time, and set a timer to trigger operation X at time T. This hardware is called a programmable interval timer. It can be set to wait a certain amount of time and then generate an interrupt, with the option of generating this interrupt periodically as well. The precision of triggers to generate interrupts is limited by the resolution of the timer, together with the overhead of maintaining virtual clocks.
- **Virtual Clocks:** Used to support more timer requests than the number of timer hardware channels. The kernel implements this simply by having a list of of interrupts scheduled both by the kernel and user requests, sorted in earliest-time-first order. It sets the timer for the earliest time, and when the timer interrupts, the kernel just signals the requester that the timer has gone off and reloads the timer with the next earliest interrupt time.

12.3.4: Nonblocking and Asynchronous I/O

- **Blocking I/O** A blocking call causes the execution of the calling thread to be suspended. Blocking application is easier to write than nonblocking application code.
- **Nonblocking I/O:** Nonblocking calls do not suspend the calling threads execution, like blocking calls do. Instead, it returns quickly, with a return value that indicates how many bytes were transferred. One way an application writer could implement this is with multithreading: some threads can perform blocking system calls, while others continue executing. Some OS's also provide nonblocking syscalls.
- **Asynchronous Calls:** An alternative to nonblocking calls. An asynchronous call returns immediately, without waiting for I/O or whatever computation to complete. The calling thread continues to execute its code. The completion of the task at some future time is then communicated to the thread (either setting some variable in the thread address space, or triggering a signal or software interrupt or a call-back routine that is executed outside the control flow of the thread).

12.3.5: Vectored I/O

- **Vectored I/O:** Allows one syscall to perform multiple I/O operations involving multiple locations. This allows multiple separate buffers to have their contents transferred via one syscall, avoiding context switching and syscall overhead. Some versions also provide atomicity.

OS Concepts 12.4: Kernel I/O Subsystem

12.4.1: I/O Scheduling

- **I/O Scheduling:** The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete. OS developers implement scheduling by maintaing a wait queue of requests for each device, rearranging the order of the queue according to a scheduling algorithm.

12.4.2: Buffering

- **Buffering:** Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. A second use is to provide adaptations for devices that have different data transfer sizes (especially common in computer networking). A third use is to support copy semantics for application I/O.
- **When Buffering Can't Help:** Buffering is useful for smoothing peaks and troughs of data rate, but it can't help if on average:
  - Process demand > data rate (process will spend time waiting)
  - Data rate > capability of the system (buffers will fill and data will spill)
  - Downside: can introduce jitter (deviation from true periodicity of a presumably periodic signal) which is bad for real-time or multimedia
- **Double Buffering:** Double buffering allows decoupling of the producer of the data from the consumer, thus relaxing timing requirements between them. This works by alternating between two buffers, where the producer is writing into one and the consumer is reading from the other, and making the switch when the producer finishes writing into its buffer.
- **Copy Semantics:** Suppose an application has a buffer of data that it wishes to write to disk by calling the `write()` syscall. If the application modifies the buffer while the I/O is being performed, is the original or the updated data now being written? Copy semantics ensure that the version of the data is the original, the version at the time of the application syscall. A simple way to implement copy semantics is for the OS to copy application data into a kernel buffer before returning control back to the application. Despite the overhead this copying introduces, copying data from application data space to kernel buffers is common because of the useful copy semantics. Clever use of virtual memory mapping and copy-on-write page protection can also be used to implement the same effect.

12.4.4: Spooling

- **Spooling:** Queue output for a device, such as a printer, that cannot accept interleaved data streams.