# Chapter 1 : Introduction

## 1.1 What is an Operating system

An operating system (OS) is the layer of software that manages a computer’s resources for its user and their applications. They are essential to more general-purpose systems such as smartphones, desktop computers, and servers. Increasingly, operating systems technologies developed for general-purpose commuting are migrating into the embedded sphere. For example, early mobile phones had simple operating systems to manage their hardware and to

run a handful of primitive applications. Today, smartphones  are capable of running independent third-party applications. Likewise, automobiles are increasingly software controlled, raising a host of operating system issues.

**Referee:** operating systems manage resources shared between different applications running on the same physical machine. An operating system is therefore a referee because it decides who gets to do what when.

**Illusionist:** Operating systems provide an abstraction of the physical hardware to simplify application design. An operating system is an Illusionist, because it simulates that application as a limitless computer power, and lets it use what it needs, and fixes all the hardware stuff for the application.

**Glue:** Operating systems provides a set of common services that facilitate sharing among applications. The operation system links different components and applications together in a computer.

### 1.1.1 Resource Sharing: Operating system as Referee

Sharing is central to most uses of computers. The operating system must somehow keep all of these activities separate, yet allow each the full capacity of the machine if the others are not running. Therefore an operating system needs to be able to do multiple things at once. On multiprocessors the computation inside a parallel application can be split into units that can be run independently for faster execution.

**Resource allocation:** The operating system must keep all simultaneous activities separate, allocating resources to each as appropriate.

**Isolation:** Error in one application should not disrupt other applications, or even the operating system itself. This is called fault isolation. Fault isolation requires restricting the behavior of applications to less than the full power of the underlying hardware. Without fault isolation provided by the operating system, any bugs in any program might irretrievably corrupt the disk.

**Communication:** The flip side of isolation is the need for communication between different applications and different users.

In the role as referee, an operating system is somewhat akin to that of a particularly patient kindergarten teacher. It balances needs, separates conflicts, and facilitates sharing.

### 1.1.2 Making Limitations: Operating System as Illusionist

A second important role of an operating system is to mask the restrictions inherent in computer hardware. Virtualization provides an application with the illusion of resources that are not physically present. Pushing this one step further, some operating systems virtualize the entire computer, running the operating system as an application on top of another operating system. This is called a virtual machine. The system thinks it is running on a real, physical machine, but it is an illusion presented by the true operating system running underneath.

### 1.1.3 Providing Common Services: Operating system as Glue

Operating systems play a third key role: providing a set of common, standard services to applications to simplify and standardize their design. An important reason for the operating system to provide common services, rather than letting each application provide its own, is to facilitate sharing among applications.Most operating systems also provide a standard way for applications to pass messages and to share memory. Another standard service in most modern operating systems is the graphical user interface library.

### 1.1.4 Operating system Design Patterns

The challenges that operating systems address are not unique; they apply to many different computer domains. Many complex software systems have multiple users, run programs written by third-party developers, and/or need to coordinate many simultaneous activities.

**Cloud computing:** is a model of computing where applications run on shared computing and storage infrastructure in large-scale data centers instead of on the user's own computers.

**Web browsers:** such as Chrome, Internet Explorer, Firefox, Safari play a similar role to an operating system. Browsers load and display web pages, but, and protect the user from bugs and dangers.

**Multi Players:** such as Flash and Silverlight, are often packaged as browser plug-ins, but they themselves provide an execution environment for scripting programs. Thus, these systems face many of the same issues as both browsers and operating systems.

**Multiplayer games** often have extensibility API to allow third party software vendors to extend the game in significant ways. Often these extensions are miniature games in their own right, yet game extensions must also be prevented from breaking the overall rules of the game.

**Multi- user database systems:** such as Oracle and MIcrosoft's SQL server allow large orgs to store, query, and update large data sets, such as detailed records of every purchase ever

made. Large scale dataset analysis greatly optimizes business operations, but, as a consequence, databases face many of the same challenges as an OS.

**Parallel applications:** are programs designed to take advantage of multiple processors on a single computer. Each application divides its work onto a fixed number of processors and must ensure that accesses to shared data structures are coordinated to preserve consistency.

## 1.2 Operating system Evaluation

### 1.2.1 Reliability and Availability

Perhaps the most important characteristics of an operating system is its reliability. Reliability means that a system does exactly what it is designed to do. As the lowest level of software running on the system, operation system errors can have devastating and hidden effects. If the operating system breaks, you may not be able to get work done, and in some cases, you may even lose previous work, e.g, if the failure corrupts files on disk. A related concept is availability, the percentage of time that the system is usable. A buggy operating system that crashes frequently losing the user's work, is both unreliable and unavailable.  Thus, both reliability and availability are desirable. Availability is affected by two factors: the frequency of failures, measured as the meantime to failure (MTTF), and the time it takes to restore a system to a working state after a failure, called the meantime to repair (MTTR) Availability can be improved by increasing the MTTF or reducing the MTTR.

### 1.2.2 Security

Two concepts closely related to reliability are security and privacy. Security means the computer's operation cannot be compromised by a malicious attacker. Privacy is an aspect of security: data stored on the computer is only accessible to authorized users. Even with strong fault isolation, a system can be insecure if its applications are not designed for security. Complicating matters is that the operating system must not only prevent unwanted access to shared data, it must also allow access in many cases. Thus, an operating system needs both an enforcement mechanism and a security policy. Enforcement is how the operating system ensures that only permitted actions are allowed. The security policy defines who is permitted who is allowed to abscess what data and who can perform which operations.

### 1.2.3 Portability

All operating systems provide applications with an abstraction of the underlying computer hardware; a portable abstraction is one that does not change as the hardware changes. Portability also applies to the operating system itself, as we have noted. Operating systems are among the most complex software systems ever invented, making it impractical to re-write them

from scratch every time new hardware is produced or a new application is developed. This means that operating systems must be designed to support applications that have not yet been written and to run on hardware that has not yet been developed. How might we design an OS to achieve portability? It helps to have a simple , standard way for applications to interact with the OS. The abstract VM of AVM. This is the interface provided by OS's to applications, including the applications programming interface, the list of function calls the operating system provided to an app, the memory access model and which instructions can be legally executed. This notion of portable hardware abstractions is so powerful that operating systems use the same idea internally: the operating system itself can largely be implemented independently of the hardware specifics.

### 1.2.4 Performance

Performance of an operating system is often immediately visible to its users. Although we often associate performance with each individual app the OS's design can greatly affect the apps perceived performance. The OS decides when an app can run how much memory it can use and whether its files are cached in memory or clustered efficiently on disk. Performance is not a single quantity. Rather it can be measured in several different ways. One performance metric is the overhead of the added resources cost of implementing an abstraction presented to applications. A related concept is efficiency, the lack of overhead is an abstraction. One way to measure overhead is the degree to which the abstraction impedes app performance. One issue is fairness between different users or apps running on the same machine. Two related concepts are response time and throughput. Response time, sometimes called delay is how long it takes for a single task to run from start to finish. Throughput is the rate at which the system completes tasks. Throughput is a measure of efficiency for a group of tasks rather than a single one. A related consideration is performance predictability whether the system's response time or other metric is consistent over time. Predictability can often be more important than average performance.

### 1.2.5 Adoption

In addition to reliability, portability and performance, the success of an OS depends on two factors outside its immediate control: the wide availability of the app ported to that OS, and the wide availability of hardware that the OS can support. The network effect occurs when the value of some tech depends not only on its intrinsic capabilities, but also on the number for other people who have adopted it. A more subtle issue is the choice of whether the OS programming interface, or OS source code itself is open or proprietary. A proprietary system is one under control of a simple company. An open system is one where the system's source code is public, giving anyone the ability to inspect and change the code.

### 1.2.6 Design tradeoffs

Most practical OS designs strike a balance between the goals of reliability, security, portability, performance, and adoption.

## 1.3 Operating systems: Past, Present ,and Future

### 1.3.1 Impact of Technology trends

The most striking aspect of the last fifty years in computing tech has been the cumulative effect of Moore’s law and the comparable advances in related techs, such as memory and disk storage, Moore’s law states that transistor density increases exponentially over time; similar exponential improvements have occurred in many other component techs.

### 1.3.2 Early Operating systems

The first OSs were runtime libraries intended to simplify the programming of early computer systems. Rather than the tiny inexpensive yet complex hardware and software of today, the fists computers often took up an entire floor of a warehouse and cost millions of dollars, and yet were capable of being used only by a single person at a time. Since computers were enormously expensive, reducing the likelihood of an error was paramount. The first os was made to do this.

### 1.3.3 Multi-User Operating Systems

The next step forward was sharing, introducing many of the advantages and challenges that we see in today's OSs. A batch OS works on a queue of tasks. It runs a simple loop. Load, run and unload each job inturn. While on job, the OS sets up the I/O devices to do background transfers for the next/prev job using a process called direct memory access. With DMA, the I/O device transfers its data directly into memory at locations specified by the OS. Batch OS were soon extended to run multiple applications at once, called multitasking or sometimes multiprogramming. Multiple programs are loaded into memory at the same time, each ready to use the processor if for any reason the previous task needed to pasie, forecample to read additional input or produce output. Debugging on these systems required a restart which wasn't cost efficient. VMs address this limitation. Instead of running a test OS directly on the hardware, VMs run and os as an app. The host os, exports an abstract virtual machine that is identical to the underlying hardware.

### 1.3.4 Time-Sharing Operating Systems

Eventually, the cumulative effect of Moore’s Law meant that the cost of computing dropped to where systems could be optimized for users rather than for efficient use of the processor. This is

where time-sharing OSs come in like MacOS, Windows and Linux. Where designed to support interactive use of the computer rather than the batch mode processing of earlier systems.

### 1.3.5 Modern Operating Systems

**Desktop, laptop, and netbook os:** These systems are single user, run many apps, and have various I/O devices. One might think that with only one user there would be no need to design the system to support sharing, and indeed the initial personal computer Os took this approach-

**Smartphone OS:** A smartphone is a cell phone with an embedded computer capable of running third party apps. While smartphones have only one user, they must support many apps. KEy design goals include responsiveness, support for a wide variety of apps, and efficient use of the battery. Another design goal is user privacy.

**Virtual machines:** A VM monitor is an OS that can run another OS as if it were an app. VM monitors face many of the same challenges as other OSs, with the added challenge posed by coordinating a set of coordinators. A commercially important use of VMs is to to allow a single server machine toi run a set of unbdepoent serviueces, Each VM can be configured as needed by that particular service.

**Embedded systems:** Over time, computers have become cheap enough to integrate into any number of consumer devices. Embedded devices typically run a customized OS bundled with the task.specific software that controls the device.

**Server clusters:** For fault tolerance, scalem and responsiveness, web sites are increasingly implemented on distributed clusters of computers housed in one or more geographically distributed data centers located close to users. If one computer fails due to hardware fault, software crash or power failure, another computer can take over its role.

### 1.3.6 Future Operating Systems

- Very large scale data centers
- Verty Large scale multicore systems
- Ubiquitous portable computing devices
- Very heterogeneous systems
- Very Large scale storage

Managing all this is the job of the operating system

# Chapter 2 The kernel Abstraction

A central role of the operating systems is protection. Protection is essential to achieving several of the operating systems goals.

**Reliability:** Protection prevents bugs in one program from causing crashes in other programs or in the operating system.

**Security:** Some users or applications on a system may be less than completely trustworthy; therefore, the operating system must limit the scope of what they can do. Without protection, a malicious user might surreptitiously change application files or even the operating system itself, leaving the user none the wiser.

**Privacy:** On a multi-user system, each user must be limited to only the data that she is permitted to access. Without protection provided by the operating system, any user or application running on a system could access anyone's data, without the knowledge or approval of the data's owner.

Implementing protection is the job of the operating systems kernel. The kernel, the lowest level of software running on the system, has full access to all of the machine hardware. A process is the execution of an application program with restricted rights; the process is the abstraction for protected execution provided by the operating system kernel.

## 2.1 The process Abstraction

What is a process? -A compiler converts code into a sequence of machine instructions and stores those instructions in a file, called the program's ex
ecutable image. The operating system sets aside a memory memory region, the execution stack, to hold the state of local variables during procedure calls. The operating system also sets aside a memory region, called the heap, for any dynamically allocated data structures the program might need.  A process is an instance of a program, in much the same way that an object is an instance of a class in object-oriented programming. Each program can have zero, one or more processes executing it.  For each instance of a program, there is a process with its own copy of the program in memory, The operating system keeps track of the various processes on the computer using a data structure called the process control block, or PCB. The PCB stores all the information the operating system needs about a particular process.

## 2.2 Dual-mode operation

How can we safely execute multiple processes at the “same time”. Any easy way to solve the problem if we aren’t thinking about performance. Let the kernel execute all code and then later check if the right permissions are present for a process to execute what it's trying to do and check if it's not, interfering with other processes and then let the application execute the processes. Can we modify the processor in some way to allow safe instructions to execute directly on the hardware? To accomplish this, we implement the same checks as in our hypothetical interpreter but with hardware rather than software. In user mode, the processor checks each instruction before executing it to verify that it is permitted to be performed by the process. In kernel mode, the OS executes with protection checks turned off.

### 2.2.1 Privileged instructions

Process isolation is possible only if there is a way to limit programs running in user mode from directly changing their privilege level. Processes can indirectly change their privilege level by executing a special instruction, called a system call, to transfer control into the kernel at a fixed location defined by the OS. Other than this an application can not change its privilege level. The application cannot be allowed to change the set of memory locations it can access, limiting an application to accessing only its own memory is essential to preventing it from either intentionally or accidentally corrupting the data or code from other applications or the operating system. Instructions available in kernel mode, but not in user mode, are called privileged instructions. The operating system kernel must be able to execute these instructions to do its work - it needs to change privilege levels, adjust memory access and disable and enable interrupts. What happens if an application attempts to access restricted memory or attempts to change its privilege level? Such actions are called processor exceptions.A processor exception causes the processor to transfer control to an exception handle in the Os kernel. Usually , the kernel simply halts the process after a privilege violation.

### 2.2.2 Memory protection

To run an application process, both the OS system and the application must be resident in memory at the same time. The application must be in memory in order to execute at the same time. The application must be in memory in order to execute while the OS must be there to start the program and to handle any interrupts, process exceptions or system calls that happen while the program runs. To make memory sharing safe, the OS must be able to configure the hardware so that each application process can read and write only its own memory, not the memory of the OS or any other app. How does the OS prevent a user program from accessing parts of physical memory. A processor has two extra registers, called base and bound. The Base specifies the start of the process’ memory region in physical memory, while bound gives

its endpoint. These registers can be changed online by privileged instructions, that is by the os executing kernel code,. User-level code cannot change their values. Every time the processor fetches an instruction, it checks the address of the program counter to see if it is between the base and the bound registers. If so, the instruction fetch is allowed to proceed otherwise, the hardware raises an exception, suspending the program and transferring control back to the OS kernel. Likewise for instructions that read or write data to memory, the processor checks each memory reference against the base and bound register, generating a processor exception if the boundaries are violated. Because applications touch only theory's own memory, the kernel must explicitly copy aunty input or output into or out of the applications memoieryt region.

**Expandable heap and stack** With a single pair of base and bound registers per process the amount of memory allocated to a program is fixed when the program starts. Although the os can change the bound, most programs have two memory regions that need to independently expand depending on program behavior.

**Memory sharing**: Bae and bound registers do not allow memory to be shared between different processes, as would be useful for sharing code between multiple processes running the same program or using the same library.

**Physical memory addresses:** When a program is compiled and linked, the addresses of its procedures and global variables are set relative to the beginning of the executable file, that is starting at zero. With the mechanism we have just described using base and bound registers, each program is loaded into physical memory at runtime and must use those physical memory addresses.

**Memory fragmentation:** Once a program starts, it is nearly impossible to relocate it. The program might store pointers in registers or on the execution stack and these printers need to be changed to move the program to a different region of physical memory. Ove time as apps start and finish at irregular times, memory will become increasingly fragmented., Potentially, memory fragmentation may reach a point where there is not enough contiguous space to start a new process despite sufficient free memory in aggregate.

For these reasons most modern processors introduce a level of indirection called virtual addresses with virtual addresses, every process's memory starts at the same place e.g. zero. Each process thinks that it has the entire machine to itself, although obviously that is not the case inreality. The hardware translates these virtual addresses to physical memory locations. The layer of indications provided by virtual addresses gives the operating system enormous flexibility to efficiently manage physical memory. Virtual addresses can also let the heap and stack start at separate ends of the virtual address space so they can grow according to program need if either the stack or heap grows beyond its initially allocated region so the os can move it to a different larger region in memory. With virtual addresses, if multiple copies of this program run simultaneously each copy of the program will print exactly the same result. This would be impossible if each copy were directly addressing physical memory locations. In other words

each instance of the program appears to run in its own complete copy of memory when it sets a value to a memory location it alone sees its changes to that location. Other processes change their own copies.

### 2.2.3 Timer interrupts

Process isolation also requires hardware to provide a way for the operating system kernel to periodically regain control of the processor when the OS starts a user level program, the processor is free to execute any user level instructions it chooses call any function in the process memory region load or store aunty value to its memory anso forth. If the app controls the process the os by def is not running on that processor the os also needs to regain control of the processor in normal operations . Almost all computer systems include a device called hardware timer, which can be set to interrupt the processor after a specified delay each timer interrupts only one processor so multiprocessors will usually have a separate timer for each CPU. When the timer interrupts occurs, the hardware transfers control from the user process to the kernel running in kernel mode.

## 2.3 Types of Mode Transfer

### 2.3.1 User to Kernel mode

**Interrupts:** An interrupt is an asynchronous signal to the processor that an external event has occurred that may require its attention. Ast eh processor executes instructions, it checks for whether an interrupt has arrived. If so, it complestres or staples any instructions that are in progress. Each different type of interrupt requires its own handler. For timer interrupts, the handler checks if the current process is being responsive to user input to detect if the process has gone into an infinite loop.  Interrupts are also used to inform the kernel of the completion of I/O requests. AN alternative to interrupts is polling the kernel loops checking each input/output device to see if an event has occurred that requires handling. Interprocessor interrupts are another source of interrupts. A processor can send an interrupt to another processor. The kernel uses these interrupts to coordinate actions across the multi processor.

**Processor exceptions:** A processor exception is a hardware event caused by user program behavior that causes a transfer of control to the kernel. As with an interrupt the hardware finishes all previous instructions, saves the current execution state and starts running at a specialist designated exception handler in the kernel.  On a multiprocessor the exception only stops execution on the processor triggering the exception the kernel then needs to send inter processor interrupts to stop execution of the parallel program on the other processors. Processor exceptions are also caused by more benign program events.

**System calls:** User processes can also transition into the Is kernel voluntarily to request that the kernel perform an operation on the user's behalf as system call is any procedure provided by the kernel that can be called from user level. Most processors implement system calls with a special trap or syscall instruction. However, special instruction not strictly required on some systems a process triggers as system call by executing an instruction with specific incalig opcode.

### 2.3.2 Kernel to User mode

Just as there are several different types of transition form user to kernel mode, there are several types of transitions from kernel to user mode.

**New process:** to start a new process, the kernel cities the program into memory, sets the program counter to the first instruction of the process, sets the stack pointer to the base of the user stack and switches to user mode.

**Resume after an interrupt, processor exception or system call:** when the kernel finishes handling the request it resumes execution of the interrupted process by restoring its program counter, restoring its registers and changing the mode back to user level.

**Switch to a different process:** In some cases, such as on a timer interrupt,the kernel switches to a different process than the one that had been running before the interrupt. Since the kernel will eventually resume the old process, the kernel news to save he proces state.

**User-level upcall:** Many operating systems provide user programs with the ability to receive asynchronous notification of events.

## 2.4 Implementing Safe Mode Transfer

The context switch code must be carefully crafted, and it relies on hardware support. To avoid confusion and reduce the possibility of error, most operating systems have a common sequence of instructions both for entering the kernel whether due to interrupts, processor exceptions or system calls and for returning to user level, again regardless of the cause.

At a minimum, this common sequence must provide:
**Limited entry into the kernel.** To transfer control to the operating system kernel, the hardware must ensure that the entry point into the kernel is one set up by the kernel. User programs cannot be allowed to jump to arbitrary locations in the kernel.

**Atomic changes to processor state.** In user mode, the program counter and stack point to memory locations in the user process memory protection prevents the user process from

accessing any memory outside of its region. In kernel mode, the program counter and stack point to memory locations in the kernel memory that protection is changed to allow the kernel to access both its own data and that of the user process.

**Transparent, restarable execution.** An event may interrupt a user level process at any point, between any instruction and the next one. The operating system must be able to restore the state of the user proga exactly as it was before the interrupt occurred. On an interrupt, the processor saves its current state to memory, temporarily defers further events, changes to kernel mode, and then jumps to the interrupt or exception handler. When the handler finishes, the steps are reversed the processor state is restored form its saved location, with the interrupted program none the wiser.

### 2.4.1 Interrupt Vector Table

When an interrupt, processor exception or system call trap occurs, the operating system must take different actions depending on whether the event is a divide by zero exception, a file read system call or a timer interrupt. The interrupt vector table is an array of pointers, with each entry pointing to the first instruction of a different handler procedure in the kernel. An interrupt handler is the term used for the procedure called by the kernel on an interrupt. The format of the interrupt vector table is processor specific.

### 2.4.2 Interrupt Stack

Where should the interrupted process's state be saved, and what stack should the kernel's code use? On most processors, a special, privileged hardware register points to a region of kernel memory called the interrupt stack. When an interrupt, processor exception, or system call trap causes a context switch into the kernel, the hardware changes the stack pointer to point to the base of the kernel's interrupt stack. When The kernel handler runs it pushes any remaining registers onto the stack before performing its work. When returning from the interrupt, processor execution or system call trap, the reverse occurs first the handler props the saved registers, and then the hardware restores the registers it saved returning to the point where the process was interrupted. You might think you could use the process user -level stack to store its state. However, a separate kernel-level interrupt stack is needed for two reasons.

**Reliability.** The process's user-level stack pointer might not be a valid memory address.

**Security:** On a multiprocessor, other threads running in the same process can modify user memory during the system call. If the kernel handler stores its local variables on the user-level stack, the user program might be able to modify the kernel's return address, potentially causing the kernel to jump to arbitrary code.

### 2.4.3 Two stacks per Process

Most operating systems kernels go one step farther and allocate a kernel interrupt stack for every user-level process. Allocating kernel stack per process makes it easier to switch to a new process inside an interrupt or system call handler.

States of the process's user and kernel stack:

If the process is running on the processor in user mode its kernel stack is empty, ready to be used for an interrupt, processor exception, or system call.

If the process is running on the processor in kernel mode due to an interrupt, processor exception or system call its kernel stack is in use containing the save registers form the suspended user-level computation as well as the current state of the kernel handler.

If the process is available to run but is waiting for its turn on the processor its kernel stack contains the registers and state to be restored when the process is resumed.

If the process is waiting for an I/O event to complete, its kernel stack contains the suspended coalition to be resumed when the I/O finishes.

### 2.4.4 Interrupt Masking

Interrupts arrive asynchronous the processor could be executing either user or kernel code when an interrupt arrives. In certain regions of the kernel such as inside interrupt handlers themselves, or inside the CPU scheduler taking an interrupt could cause confusion. To simplify the kernel design, the hardware provides a privileged instruction to temporarily defer delivery of an interrupt until it is safe to do so. This instruction is called disable interrupts. If multiple interrupts arrive while interrupts are disabled, the hardware delivers them in turn when interrupts are re-enabled. However, since the hardware has limited buffering for pending interrupts, some interrupts may be lost if interrupts are diable for too long a period of time.

### 2.4.5 Hardware Support for saving and Restoring Registers

An interrupted process's registers must be saved so that the process can be restarted exactly where it left off. Because the handler might change the values in those registers as it executes, the state must be saved before, the handler runs.

## 2.6 implementing Secure system calls

The operating system kernel constructs a restricted environment for process execution to limit the impact of erroneous and malicious programs on system reliability. Any time a process needs to perform an action outside of its protection domain to create a new process read from the keyboard or write a disk block it must ask the operating system to perform the action on its behalf via a system call. System calls provide the illusion that the operating system kernel is simply a set of library routines available to user programs. Inside the kernel, a procedure implements each system call. This procedure behaves exactly as if the call was made form within the kernel but with one notable difference the kernel must implement its system calls in a way that protects itself from all errors and attacks that might be launched by the misuse of the interface.  We bridge these two views: the user program calling the system call and the kernel implementing the system call with a paro of stubs, is a pair of procedures that mediate between two environments, in this case between the user program and the kernel. Stubs also mediate procedure calls between computers in a distributed system.

The kernel stub has four tasks:
**Locate system call arguments**: Unlike a regular kernel procedure, the arguments to a system call are stored in user memory, typically on the user stack. Of course, the user stack pointer may be corrupted! Even if its is valid it is a virtual not a physical address. If the system call has a pointer argument the stub must check the address to verify it is a legal address within the user domain. If so the stub converts it to a physical address so that the kernel can safely use it.

**Validate parameters:**  The kernel must also protect itself against malicious or accidental errors in the format or content of its arguments.

**Copy before check** In most cases the kernel copies system call parameters into kernel memory before performing the necessary checks.  This is because the kernel cant let the app change the parameters as the call is being executed.

**Copy any results:** For the user program to access the results of the system call the stub must copy the result form the kernel into user memory.

## 2.7 Starting a new Process

To start running at user level in the first place the Kernel must:

- Allocate and initialize the process control block.
- Allocate memory for the process
- Copy the program from disk into the newly allocated memory.
- Allocate a user level- stack for user-level execution.

- Allocate a kernel-level stack for handling system calls, interrupts and processor exceptions.

To start running the program the kernel must also:

**Copy arguments into user memory:** When starting a program, the user may give it arguments, much like calling a procedure.

**Transfer control to user mode:** When a new process starts, there is no saved state to restore. While it would be possible to write special code for this case, most operating systems re-use the same code to exit the kernel for starting a new process and for returning from a system call. When we create the new process ,we allocate a kernel stack to it and we reserve room at the bottom of the kernel stack for the initial values of its user-space registers, program, we can then switch to the new stack and jump to the end of the interrupt handler. When the handler executes popad and iret the processor etuns to the start of the user program.

## 2.8 Implementing Upcalls

There are several uses for immediate event delivery with upcalls:

**Preemptive user-level threads:** Just as the OS kernel runs multiple processes on a single processor, an app may run multiple tasks or threads in a process A user-level thread package can use a periodic timer upcall as a trigger to switch tasks to share the processor more evenly among user.level task or to stop a runaway task.

**Asynchronous I/O notification:** An asynchronous I/O a system call starts the request and returns immediately. Later, the application can poll the kernel for I/O completion, or a separate notification can be set via an upcall to the application when the I/O completes.

**Interprocess communication:** A kernel upcall is needed if a process generates an event that needs the instan attention of another process.

**User-level exception handling:** Many apps have their own exception handling for this; the OS needs to inform the app when it receives a processor exception so the app runs, rather than the kernel handling the event.

**User-level resource allocation:** For this the OS must inform the process when its allocation changes because some other process needs more or less memory

Similarities with hardware interrupts:

**Types of signals:** In place of hardware-defined interrupts and processor exceptions, the kernel defines a limited number of signal types that a process can receive.

**Handlers:** each process defines its own handlers for each signal type, much as the kernel defines its own interrupt vector table. If a process does not define a handler for a specific signal then the kernel called s default handler instead.

**Signal stack:** Applications have the option to run UNIX signal handlers on the process's normal execution stack or on a special signal stack allocated by the user process in user memory. Running signal handlers on the normal stack makes it more difficult for the signal handler to manipulate the stack if the runtime needs to raise a language level exception.

**Processor state:** The kernel copies onto the signal stack the saved state of the program counter stack pointer and general-purpose registers at the part when the program stopped. The signal handler can also modify the task saved state so that the kernel resumes a different user-level task when the handler returns.

# 3 The programming Interface

What function do we need an OS to provide applications

- **Process management**
- **Input/output**
- **Thread management**
- **Memory management**
- **File system and storage**
- **Networking and distributed systems**
- **Graphics and window management**
- **Authentications and security**

As long as the OS provides that interface, where each function is implemented is up to the OS based on a tradeoff between flexibility, reliability, performance and safety

**Flexibility:** the OS needs to support all the apps written for it so if an update is made the OS still needs to support its apps that aren't made for that update.

**Safety** However resource management and protection are the responsibility of the OS kernel, Protection checks cannot be implemented in a user-level library because application code can skip any checks made by the library.

**Reliability** Improved reliability is another reason to keep the OS kernel minimal. “What can be at user level, should be” An extreme version of approach is to isolate privileged, but less critical parts of the OS. This is called a microkernel design. IN a microkernel the kernel itself is kept small and instead most of the functionality of a traditional OS kernel is put into a set of user-.level processes or servers,accessed from user-level apps via interprocess communication.

**Finally** transferring control into the kernel is more expensive than a procedure call to the library and transferring control to a user-level file system server via the kernel is still even more costly.

# 3.1 Process management

Today programs that create and manage processes include windows manager, web servers web browsers shell command lines interpreters, source code control systems, database compilers, and document preparation systems. If creating a process is something a process can do, then anyone can build a new version of any of these applications, without then anyone can build a new version of any of these apps, without recompiling the kernel or forcing anyone else to use it.

## 3.1.1 Windows Process Management

What steps does createProcess take:

- Create and initialize the process control block in the kernel.
- Create and initialize a new address space.
- Load the program prog into the address space.
- Copy arguments argos into memory in the address space
- Initialize the hardware context to start execution at “start”
- Inform the scheduler that the new process is ready to run.

## 3.1.2 UNIX Process Management

UNIX splits CreateProcess in two steps called fork and exec. UNIX fork creates a complete copy of the parent process, with one key exception. The child process sets up privileges, priorities and I/I for the program that is about to start by closing some files, opening others, reducing its priority if it is to run in the background. Once the context is set, the child process calls UNIX exec. UNIX exec brings the new executable image into memory and starts it running. With this design, UNIX forks takes no arguments and returns an integer. UNIX exec takes two arguments: the name of the process and its arguments in the form of an array.
The steps for implementing UNIX fork in the kernel:

- Create and initialize the process control block in the kernel
- Create a new address space

- Initialize the address space with a copy of the entire contents of address space of the parents
- Inherit the execution context of theperent
- Inform the scheduler that the new process is ready to run

UNIX exec does the following steps:

- Load the program prog into current address space
- Copy arguments argos into memory in the address space.
- Initialize the hardware context to start execution at “start”

Note that exec does not create a new process
UNIX has a system call, naturally enough called wait, that pauses the parent until the child finishes, crashes or is terminated.

## 3.2 Input/Output

One of the primary innovations in UNIX was to regularize all device input and output behind a single common interface. The basic idea in UNIX I/O interface are:

- **Uniformity** All devices I/O file operations, and interprocess communication use the same set of system calls open close read and write.
- **Open before use** before an app does I/O, it must first call open on the device file or communication channel. This gives the operating system a chance to check access permissions and to set up any internal bookkeeping.
- **Byte-oriented** All devices even those that transfer fixed-size-blocks of data are accessed with byte arrays. Similarly file and communication channel access is in terms of bytes, even though we store data structures in files and send data structures across channels.
- **Kernel-buffered reads.** Stream data, such as from the network or key-board is stored in a kernel buffer and returned to the application on request.
- **Kernel-buffered writes.** Likewise outgoing data is stored in a kernel buffer for transmission when the device becomes available.
- **Explicit close.** When an app is done with the device or file, it calls close. This signals to the OS that it can decrement the reference-count on the device and garbage collect any unused kernel data structures.
- **Pipes** A UNIX pipe is a kernel buffer with two file descriptors one for wiring and one for reading. Data is read in exactly the same sequence it is written, but since the data buffered, the execution of the producer and consumer can be decoupled reducing waiting in the common case- The pipe terminates when either endpoint closes the pipe or exits.
- **Replace file descriptors.** By manipulating the file descriptors of the child process, the shell can cause the child to read its input from, or send its output to, a file or a pipe instead of from a keyboard or to the screen.

- **Wait for multiple reads.** For client-server computing, a server may have a pipe open to multiple client processes. Normally read will block if there is no data to be read, and it would be inefficient for the server to pool each pipe in turn to check if there is work for it to do. The UNIX system call select addresses this. Select allows the server to wait for input from any set of file descriptors, it returns the file descriptor that has data, but it doesn't read the data.

## 3.3 Case study: Implementing a Shell

UNIX programs do not need to be aware of where their input is coming from, or where their input is going. This is helpful in a number of ways:

- **A program can be a file of commands.** Programs are normally a set of machine instructions, but on UNIX a program can be a file containing a list of commands for a shell to interrupt.
- **A program can send its output to a file.** By changing the stdout file descriptor in the child, the shell can redirect the child's output to a file.
- **A program can read its input from a file.** Likewise, by using dup2 to change the stdin file descriptor the shell can cause the child to read its input from a file.
- **The output of one program can be the input of another program.** The shell can use a pipe to connect two åprograms together so that the output of one is the input of another. This is called a producer-consumer relationship.

## 3.4 Case Study: Interprocess communication

- **Producer-consumer.** In this model, programs are structures to accept as input output of either program. Communications is one way the producer only writes and the consumer only reads.
- **Client-server** An alternative model is to allow two-way communication between processes as in client server computing the server implements some specialized task, such as managing the printer queue or managing the display. Clients send requests to the server to do some task, and when the operation is complete, the server replies back to the client.
- **File system.** Another way programs can be connected together is through reading and writing files. A text editor can import an image created by a drawing program and the ditor can in turn write an HTML file that a web server can read to know how to display a web page.

## 3.5 Operating system Structure

There are many dependencies among the modules inside the OS and there is often quite frequent interaction between these modules:

- Many  parts of the OS depend on synchronuzation primitives for coordinating access to shared data structures with the kernel
- The virtual memory system depends on low level hardware support for address translation support that is specific to a particular processor architecture
- Both the file system and the virtual memory system share a common pool of blocks of physical memory. They also depend on the disk device driver.
- The file system can depend on the network protocol stack if the disk is physically located on a different machine.

This has led OS designers to wrestle with as fundamental tradeoff by centralizing functionality in the kernel, performance is improved and it makes it easier to arrange tight integration between kernel modules however the resulting system is less flexible , less easy to change and less adaptive to user or apps needs.

### 3.5.1 Monolithic kernel

With a monolithic kernel, most of the OS functionality runs inside the OS kernel. In truth the term is a bit of a misnomer, because even so.-Called monolithic systems, there are often large segments of what users consider IS that runs outside the kernel  either as utilities like the shell,, or in system libraries such as liberties to manage the User interface. Internal to a monolithic kernel, the OS designer is free to develop whatever interface between modules that make sense, and so there's quite a bit of variation from OS to IS in those  internal structures.

A key goal of OSs is to be portable across a wide variety of hardware platforms. To  accomplish this, especially within a monolithic system, requires careful design of the hardware abstraction layer. The hardware abstraction layer is a portable interface to machine configuration and processor specific operations within the kernel. OSs that are portable across processor families, say between an arm and an x86 or between 43-bit and a 64-bit x86, wil process specific code for process and thread context switches. The interrupt,m processor exception, and system call tap handling is also processor specific; all systems have those functions, but the specific implementation will vary.

# Chapter 4 Concurrency and Threads

# 4.1 Thread Use cases

## 4.1.1 Four reasons to use Threads

Program structure: Expressing logically concurrent tasks. Programs often interact with or simulate a real.-world applications that have concurrent activities. Threads let you express an application's natural concurrency by writing each concurrent task as a separate thread.

Responsiveness: shifting work to run in the background. To improve user responsiveness and performance a common design pattern is to create threads to perform work in the background, without the user waiting for the result. This way the user interface can remain responsive to further commands regardless of the complexity of the user request.

Performance: exploiting multiple processors. Programs can use threads on a multiprocessor to do work in parallel; they can do the same work in less time or more work in the same time.

Performance managing I/O devices. To do useful work, computers must interact with the outside world via I/O dives. By running tasks as separate threads when one task is waiting for I/O the processor can make progress on a different task.

# 4.2 Thread Abstraction

Single execution sequence Each thread executes a sequence of instructions assignments conditionals loops procedures and so on just as in the familiar sequential programming model.

Separately schedulable task. The OS can run, suspend or resume a thread at any time.

## 4.2.1 Running, suspending and resuming threads

Threads provide the illusion of an infinite number of processors. How does the OS implement this illusion? It must execute instructions from each thread so that each thread makes progress, but the underlying hardware has only a limited number of processors and perhaps only one! To map an arbitrary set of threads to a fixed set of processors, operation systems include a thread scheduler that can switch between threads that are running and those that are ready but not running.

4.3 Is an example of how to program threads

# 4.4 Thread Data structures and life cycle

## 4.4.1 Per-Thread State and Thread Control BLock (TCB)

The operating system needs a data structure to represent a thread's state. A thread is like any other object in this regard. This data structure is called the Thread control block (TBC). For every thread the OS creates it create one TCB

The thread control block holds two types of per-thread info:

1. The state of the computation being performed by the thread.
2. Metadata about the thread that is used to manage the thread.

Per-thread computation state. To create multiple threads and to be able to start and stop each thread as needed, the OS musta allocate space in the TCB for the current state of each thread's computation, a pointer to the threads stack and accept its processor registers.

Stack: A thread's stack is the same as the stack for a single threaded computation; it stores information needed bv nested procedures the thread is currently running. When a new thread is created the OS allocates a new stack and stores a pointer to that stack in the thread's TCB.

Copy of processor registers. A processor's registers include not only its general-purpose registers for storing intermediate values for ongoing computations but they also include special purpose registers such as the instruction pointer and stack pointers. In some systems the general purpose registers for a stopped thread are stored on the pop of the stack and the TCB contains only a pointer to the stack. In other systems the tcb contains space for a copy of all processor registers.

Per-thread Metadata The TCB also includes per-thread metadata information for managing the thread.

# 4.5 Thread life cycle

INIT Thread creation puts a thread into its INIT state and allocates and initializes power-thread data structures. ONce that is done thread creation code puts the thread into the READY state by adding the thread to the ready list.

READY A thread in the READY state is available to be run but is not currently running its TCB is on the ready list, and the values of its registers are stored in its TCB . At any time the scheduler can cause a thread to transition from READY to running by copying its register values from this TCB to a processes registers.

RUNNING a thread in the running state is running on a processor. At this time its register values are stored on the processor rather than in the TCB. A RUNNING thread can transition to READY in two ways.

The scheduler can preempt a running thread and move it to the READY state by saving the thread's registers to its TCB and switching the processor to run the net thread on the readylist.

A running thread can voluntarily relinquish the processor and go from RUNNING to READY by calling yield.

WAITING A thread in the waiting state is waiting for some event Whereas the scheduler can move a thread in the RAD state to RUNNING state a thread in the waiting state cannot run until some action by another thread moves it from WAITING to READY. RUNNING to WAITING until the child thread exits. While a thread waits for an event it cannot make progress therefore it is not useful to run uit. Rather than continuing to run the thread or storing the TCB on the scheduleræs ready list THe TCB is stored in the ating list of some synchronization variable associated with the event.

FINISHED A thread in the FINISHED state never runs again. The system can free some or all of its state for other uses, through it may keep some remnants of the thread in the FINISHED state for time by putting TCB on a finished list

## 4.8 Implementing Multi-threaded Processes

### 4.8.1 Implementing Multi_threaded processes using kernel threads

The simplest way to support multiple threads per process is to use the kernel thread implementation. When a kernel thread creates, deletes, suspends, or resumes a thread it can use a simple procedure call. When a user-level thread accesses the thread library to do the same things it uses a system call to ask the kerel to do the operation on its behalf.

To create a thread, the user library allocates a user-level stack for the thread and then does a system call into the kernel. The kernel alloicates a TCB and interrupt stack, and arranges the state of the thread to start execution on the user-level stack at the beginning of the requested procedure.

### 4.8.2 Implementing User-level Threads with Kernel Support

The thread library instanisates all of its data structures within the process: TCBs the ready list, the finished list and then the calls to the thread library are just procedure calls, akin to how the

same functions are implemented within a multi-threaded kernel. To the OS kernel, a multi-threaded application using green threads appears to be a normal, single threaded process. The Process as a whole can make system calls, be time.-sliced, etc.

To implement preemptive multithreading for some process P:

1. The user-level thread lib makes a system call to register a timer signal handler and signal stack with the kernel.
2. When a hardware timer interrupt occurs, the hardware saves P's register state and runs the kernelæs handler.
3. Instead of restoring P's register state and resiming P where it was interrupted the kernelæs handler copies Pæs saved registers onto P's signal stack
4. The kernel resumes execution in P at the registered signal handler on the signal stack.
5. The signal handler copies the processor state of the preemptive user-level thread from the signal stack to that thread's TCB
6. The signal handler chooses the next thread to run, re-enables the signal handler and restores the new thread's state form tits TCB into the processor.- execution with the state stored on the single stack.

# 5 Synchronizing Access to Shared Objects

Most multithreaded programs have both a pre-thread state and a thread's stack and registers and shared state e.g shared variables on the heap. Cooperation threads read and write sheared state. We can't think of a multithreaded program in the same way as a normal program. This is because of these three reasons:

1. Program execution depends on the possible interleavings of threads accessing the shared state.
2. Program execution can be nondeterministic
3. Compilers and processor hardware can reorder instructions

## 5.1 Challenges

### 5.1.1 Race Conditions

A race condition occurs when the behavior of a program depends on the interleaving of operations of different threads.

### 5.1.2 Atomic Operations

An atomic operation is an indivisible operation that cannot be interleaved with or split by other operations

### 5.1.3 Too much Milk

Too much milk is a multithreaded programming problem that revolves around a group of people who lives together, and finds out that they are out of milk one by one which leads to too much milk being bought because the threads don't stop each other from doing the same tasks. 5.1.4 and 5.1.5 came up with solutions for this problem, which would be to use something called a lock.

## 5.2 Structuring Shared Objects

Decades of work have developed a much simpler approach to writing multithreaded programs than using hust atomic loads and stores. This approach extends the modularity of object-oriented programming ti multi-threaded programs. Shared objects are objects that can be accessed safely by multiple threads. All shared state in a program including variables allocated on the heap and static, global variables should be encapsulated in one or more shared objects.

### 5.2.1 implementing Shared Objects

Shared objects are implemented in three layers.

- Shared objects layer as in standard object oriented programming, shared objects denie application-specific logic and hide internal implementation details.
- Synchronization variable layer. Rather than implementing shared objects directly with carefully interleaved atomic loads and stores, shared objects include synchronization variables as member variables. Synchronization variables, stored in memory just like any other object, can be included in any data structure. A synchronization variable is a data structure used for coordinating concurrent access to shared state. Both interface and the implementation of synchronization variables must be carefully designed In particular, we build shared objects using two types of synchronization  variables locks and conditions variables. Synchronization variables coordinate access to state variables, which are just the normal member variables of an object that you are familiar with.
- Atomic instruction layer, Although the layers above benefit from a simpler programming model. It is not turtles all the way down. Internally synchronization variables must manage the interleaving of different threads' actions.

## 5.3 Locks: Mutual Exclusion

A lock is a synchronization variable that provides mutual exclusion when one thread holds a lock no other thread can hold it. Mutual exclusion greatly simplifies reasoning about programs because a thread can perform an arbitrary set of operations while holding a lock and those operations appear to be atomic to other threads.

### 5.3.1 Locks: API and Properties

- A lock can be in one of two states BUSY or FREE
- A lock is initially in the FREE state.
- Lock::acquire waits until the lock is FREE and then atomically makes the lock BUSY.
- Checking the state to see if it is FREE and setting the state to BUSY are together an atomic operation. Even if multiple threads try to acquire the lock, at most one thread will succeed. One thread observes that the lock is Free and sets it to BUSy , the other threads just see that the lock is BUSY and wait.
- Lock::release makes the lock FREE. If there are pending acquire operations this state change causes one of them to proceed.

Formal properties A lock can be defined more precisely as follows. A thread holds a lock if it has returned form a lock's acquire method more often than it has returned form a lock's release method. A thread is attempting to acquire a lock if it has called but not yet returned form a call to acquire on the lock. A lock should ensure the following properties

1. Mutual exclusion at most one threads holds the lock
2. Progress If no thread holds the lock and any thread attempts to acquire the lock then eventually some thread succeeds in acquiring the lock
3. Bounded waiting If thread T attempts to acquire a lock, then there exists a bound on the number of times other threads can successfully acquire the lock before T does.

## 5.4 Condition Variables: Waiting for Change

### 5.4.1 Condition Variable Definition

A condition variable is a synchronization object that lets a thread efficiently wait for a change to a shared state that is protected by a lock. A condition variable has three methods:

- CV::wait(Lock *lock) makes the thread wait for the lock
- CV::signal() makes a thread ready to run
- CV::broadcast() marks all variables off a thread as ready to run

Bottom line: Given the range of possible implementations and the modularity benefits, wait must always be done form within a loop that tests the desired predicate.

## 5.5 Designing and Implementing Shared Objects

### 5.5.1 High Level Methodology

Most high level design challenges for a shared object's are the same as for class design in single threaded programming:

- Decompose the problem into objects
- For each object
  Define a clean interface

Identify the right internal state and invariants to support that interface
Implement methods with appropriate algorithms to manipulate that state.

Compared to how you implement a class in a single-threaded program, the new steps needed for the multi threaded case for shared objects are straight forward

1. Add a lock
2. Add code to acquire and release the lock
3. Identify and add condition variables
4. Add loops to wait using the condition variables
5. Add dinsal and broadcast calls

# Chapter 6 Multi-Object Synchronization

What happens as programs become more complex with multiple shared objects and multiple locks? To answer this, we need to reason about the interaction between shared objects. Several considerations arise in this context:

- Multiprocessor performance
- Correctness
- Deadlock

## 6.1 Multiprocessor Lock Performance

Even with sample request parallelism, however, performance can often be disappointing. Most often this can be due to three causes:

1. Locking
2. Communication of shared data
3. False sharing

Fortunately these effects can be reduced through careful design of shared objects. To implement caching of web pages, the server might have a shared data structure, such as a hash table on the search terms, to point to the cached page if it exists.

## 6.2 Lock Design Patterns

Design patterns:

- Fine-Grained Locking: Partition: an object's state into different subsets; most accesses are performed by threads running on the same processor.
- Per-Processor Data Structures: Partition an object's stature so that all or most accesses are performed by threads running on the same processor.
- Ownership Design Pattern: Remove a shared object from a shard container so that only a single thread can read or modify the object.

- Staged Architecture: Divide the system into multiple stages, so that only those threads within each stage access that stage's shared data.

## 6.3 Lock Contention

Two alternate implementations of the lock abstraction:

- MCS Locks: MCS is an implementation of a spinlock optimized for the case when there are a significant number of waiting threads.
- RCU Locks: RCU is an implementation of a reader/writer lock, optimized for the case when there are many more readers than writers. RCU reduces the overhead for readers at a cost of increased overhead for writers. Most importantly, RCU has somewhat different semantics than a normal reader/writer lock, placing a burden on the use of the lock to understand its danger.

## 6.4 Multi-Object Atomicity

Once a program has multiple shared objects, it becomes both necessary and challenging to reason about interactions across objects. Sometimes it is possible to address issues through careful class and interface design. This includes design of individual objects.

### 6.4.2 Acquire-All/Release-All

One approach called acquire-all/release-all is to acquire every lock that may be needed at any point while processing the entire set of operations in the request. Then, once the thread has all of the locks it might need, the thread can execute the request, and finally release the locks. Acquire-all/release-all allows significant concurrency. When individual requests touch non-overlapping subsets of state protected by different locks, they can proceed in parallel. A key property of this approach is serializability across requests; the results of any program execution is equivalent to an execution in which requests are processed one at a time in some sequential order. Serializability allows one to reason about multi-step tasks as if each task executed alone. One challenge to using this approach is knowing exactly what locks will be needed by a request before beginning to process it. A potential solution is to conservatively acquire more locks than needed.

### 6.4.3 Two-Phase Locking

Two phase locking refines the acquire-all/release-all pattern to address this concern. Instead of acquiring all locks, locks can be acquired as needed for each operation. However, locks are not released until all locks needed by the request have been acquired. Most implementations simply release all locks at the end of the request. Two-phase locking avoids needing to know what looks to grab.

# 6.5 Deadlock

A challenge to constructing multi threaded programs is the possibility of deadlock. A deadlock is a cycle of eating among a set of threads, where each thread waits for some other in the cycle to take some action. Deadlock can occur in many different situations. Deadlock can occur when one thread holds the lock on the first object, and another thread holds the lock on the second object if the first thread calls into the second object while still holding ointoits lock, it will need to wait for the second object's lock. If the other thread does the same thing in reverse neither will be able to make progress.

## 6.5.1 Deadlock vs Starvation

Deadlock and starvation are both liveness concerts. In starvation, a thread fails to make progress for an indefinite period of time. Deadlock is a form of starvation but with stronger condition: a group of threads forms a cycle where none of the threads make progress because each thread is waiting for some other thread in the cycle to take action. JUst because a system can suffer from deadlock or starvation does not mean that it always will. A system that is subject to starvation or deadlock may live in many or most runs and stare or deadlock only for particular loads.

## 6.5.2 Necessary Conditions for Deadlock

There are four necessary conditions for deadlock to occur. Knowing these conditions is useful for designing solutionis if you can prevent any one of these conditions, then you can eliminate the possibility of deadlock:

1. Bounded resources There are a finite number of threads that can simultaneously use a resource.
2. No preemption. Once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it.
3. Wait while holding. A thread holds one resource while waiting for another. This condition is sometimes called multiple independent requests because it occurs when a thread first acquires one resource and then tries to acquire another.
4. Circular waiting. There is a  set of waiting threads such that thread is waiting for a resource held by another.

The four conditions are necessary but not sufficient for deadlock. When there are multiple instances of a type of resource, there can be a cycle of waiting without deadlock because a thread not in the ccle may return resources that enable a waiting thread to proceed.

# Chapter 7 Uniprocessor Scheduling

## 7.1 Uniprocessor Scheduling

We begin with three simple policies first in first out, shortest job first and round robin. A workload is a set of tasks from some system to perform, along with when each task arrives and how long each task takes to complete. Given a workload ,a processor scheduler decides when each task is to be assigned the processor. A scheduler is work-conserving if it never leaves the processor idle if there is work to do. Obviously a trivially poor policy has the processor sit idle for long periods of time while there are tasks to do.

### 7.1.1 First-In-First-Out

Perhaps the simplest scheduling algo possible is first in first out. Do each task in the order in which it arrives is sometimes also called first come first served. FIFO minimizes overhead, switching between tasks only when each one completes. Because it minimizes overhead if we have a fixed number of tasks and those tasks only need the therpoocessor, FIFO will have the ebay throughput; it will complete the most tasks the most quickly.  Unfortunately, FIFO has a weakness. If a task with very little work happens to land in line behind a task that takes a very long time then the system will seem very inefficient.

### 7.1.2 Shortest job first

Suppose we could  know how much time each task needed at the processor. If we always schedule the task that has the least remaining work to do, that will minimize average response time. What counts as the shortest is the remaining time left on the task not the original length. It turns out that SJF is pessimal for variance in response time. By doing the shortest tasks as quickly as possible, SJF necessarily does longer tasks as slowly as possible. Worse, SJG can suffer from starvation and frequent contact switches when enough short tasks arrive, long tasks may never complete.

### 7.1.3 Round Robin

With round robin tasks take turns running on the processor for a limited period of time. The scheduler assigns the processor the first task in the ready list, setting a timer interrupt for some delay called the time quantum,. At the end of the quantum if the task has not completed the task is preempted and the processor is given the net trask in the ready list. With round robin there is no possibility that a task will starve it will eventually reach the form of the queue and get its time quantum. One way of viewing Round robin is a compromise between fIFO and SJF. At one extreme if the time quantum is infinite, Robin behaves as FIFO. At the other extreme, it was possible to switch between tasks with zero overhead so we could choose a time quantum of a

single instruction. With fine grained time slicing tasks would finish in the order of length as with SJF.

### 7.1.4 Max-Min Fairness

In many settings a fair allocation of resources is as important to the design of a scheduler as responsiveness and low overhead. On a multi user machine or on a server we do not want to allow a single user to be able to monopolize the resources of the machine, degrading service for other users. Round Robin among threads can lead to starvation if applications with only a single thread are competing with applications with hundreds of threads. While there are many possible definitions of fairness a particularly useful one is called max min fairness it iteratively maximizes the minimum allocation given to a particular process until all resources are assigned. If all processes are compute bound the behavior of max min is simple we maximize the minimum by giving each process exactly the same share of processor.

### 7.1.6 Summary

- FIFO is simple and minimizes overhead.
- If tasks are variable in size then fifo can have a very poor average response time.
- If tasks are equal in size fifof is optimal in terms of average response time
- Considering only the processor SJFis optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time
- If tasks are variable in size ROUND robin approximates SJF,
- If tasks are equal in size Round Robin will have a very poor average response time.
- Tasks that intermix processor and I/o beefit form SJF and can do poorly under Round robin.
- Max-min fairness can improve response time for I/o bound tasks.
- Round robin and max min fairness both avoid starvation
- By manipulating the assignment of tasks to priority queues na MFQ scheduler can achieve a balance between responsiveness low overhead and fairness

## 7.2 Multiprocessor scheduling

### 7.2.1 Scheduling Sequential application on multiprocessors.

A simple approach would be to use a centralized level feedback queue with a lock ensuring only one processor at a time is tadin or modifying the data structure.

# Chapter 8 Address Translation

What can an OS do with address translation

- Process isolation
- Interprocess communication
- Shared code segments
- Program initialization
- Efficient dynamic memory allocation
- Cache management
- Program debugging
- Efficient I/O
- Memory mapped files
- Virtual Memory
- Checkpointing and restarts
- Persistent data structures
- Process migration
- Information flow control
- Distributed shared memory

## 8.1 Address Translation Concept

The translator takes each instruction and data memory and converts it to a physical memory address that can be used to fetch or store instructions or data. The data itself whatever is stored in  memory is returned as is; it is not transformed in any way.. The translation is usually implemented in hardware and the OS kernel configures the hardware to accomplish its aims.

Here are some goals we might want out of a translation box:

- Memory protection
- Memory sharing
- Flexible memory placement
- Sparse addresses
- Runtime Look up efficiency
- Compact translation tables
- Portability

## 8.2 Towards Flexible Address Translation

The base register specifies the start of the process's region of physical memory; the bound register specifies the extent of that region. If the base register is added to every address generated by the program, then we no longer need a relocating loader: the virtual address of the program starts from 0 and goes to bound, and the physical address starts from base and goes to base+bound. Since physical memory can contain several processes the kernel resets the contents of the base and bounds registers on each process context switch to the appropriate values for that process. Base and bound translation support only coarse grained protection at the level of the entire process; it is not possible to prevent a program from overwriting its own code.

### 8.2.1 Segmented Memory

Many of the limitations of base and bound translation can be remedied with a small change instead of keeping only a single pair of base and bounds registers per process, the hardware can support an array of pairs of base and bounds registers, for each process. This is called segmentation. Each entry in the array controls a portion, or segment, of the virtual address space. TRhe physical memory for each segment is stored ontigouslyt, but different segments can be stored at different locations; the rest of the address is then treated as above added to the base and checked against the bound stored at that index. Program memory is no longer a single contiguous region, but instead it is a set of regions. Each different segment starts at a new segment boundary. Although simple to implement and manage, segmented memory is both remarkably powerful and widely used. With segments the OS can allow processes to share some regions of memory while keeping other regions protected. The processes can share the same code while working off different data, byt setting up a segment table to point to different regions of physical memory for the data segment. Likewise with a for example a graphics lib.

If it's not clear when the program starts how much memory it will use the heap could be anywhere for a few kilobytes to several gigabytes, depending on the program. The OS can address this using zero-on-reference. With zero-on-reference, the OS allocates a memory region for the heap, but only zeros the first few kilobytes, Ibnsted it sets the bound register in the segment talve to limit the rogra to just the zeroed part of memory. However we chose to place new segments as more memory becomes allocated, the OS may reach a point where there is enough free space for a new segment, but the free space is not contiguous. This is called external fragmentation. The OS is free to compact memory to make room without affecting applications, because virtual addresses are unchanged when we relocate a segment in physical memory.

## 8.2.2 Paged Memory

An alternative to segmented memory is paged memory. With paging, memory is allocated in fixed chunks called page frames. Address translation ios simulator not how it works with segmentations. Instead of a segmented table whose entries contains pointers to variable-sized segments, there is a page table for each process whose entries contain pointers to page frames. There is no need for a bound on the offset; the entire page in physical memory is allocated as a unit. What will seem odd, about paging is that while a program thinks its memory is linear. However the instructions located at the end of a page will be located in a completely different region of physical memory form the next instruction at the start of the next page.

Paging addresses the principal limitation of segmentation free-space allocation is very straightforward. The operation system can represent physical memory as a bitmap with each bit representing a physical page frame that is either free or in use. Finding a free frame is just a matter of finding an empty bit. Sharing memory between processes is also convenient. We need to set the page table entry for each process sharing a page to point to the same physical page frame. The data structure for this is called a core map; it records info about each physical page frame such as which page table entries point ti uit. Many of the optimizations we discussed under segmentation can also be done with paging. Page tables allow other features to be added. Once the first few pages are in memory, however, the Os can start execution of the program in user-mode, while the kernel continues to transfer the rest of the pgoramæs code in the background.

A data breakpoint is a request to stop the execution of a program when i references or modifies a particular memory location. It is helpful during debugging to know when a data structure has been changed particularly when tracking downpointer errors. A downside of paging is that while the management of physical memory becomes simpler the management of the virtual memory space becomes more challenging. The size of the page tables is proportional to the size of the virtual address space not to the size of physical memory. The more sparse the virtual address space, the more overhead is needed for the page table. We can reduce the space taken up by the page table by choosing a larger page frame. A larger page frame can waste apsce if a process does not use all of the moment inside the frame thi si called internal fragmentation.

## 8.2.3 Multi level Translation

Many systems use tree-based address translation, although the details varty from system to system, and the terminology can be a bit confusing. Almost all multi-level address translation systems use paging as the lowest level of the tree. The main difference between systems is in how they reach the page table at the leafs of the tree. There are several reasons for this:

- Efficient memory allocation
- Efficient disk transfers
- Efficient lookup

- Efficient reverse lookup
- Page-granularity protection and sharing

### Page segmentation

With page segmentation, memory is segmented, but instead of each segment entry pointing directly to a contiguous region of physical memory each segment table entry points to a page table, which in turn points to the memory baking that segment. The segment table entry bound describes the page table length that is the length of segment in pages.

### Multi-level paging

A nearly equivalent approach to page segmentation is to use multiple levels of page tables. Only the top-level page table must be filled in to lower level for the tree to be allocated only if those portions of the virtual emm are in use by a particular process.

### 8.2.4 Portability

This software memory management data structure mirror but are not indeidail to the hardware structures consent of three parts:

- List of memory objects
- Virtual to physical translation
- Physical to virtual translation

## 8.3 Towards Efficient Address translation

### 8.3.1 Translation Lookaside buffers

The hardware will first translate the program counter for the add instruction walking the multi level translation tale to find the physical memory where add instructions are stored. When the program counter is incremented the processor must walk the multiple levels again to find the physical memory where the mult instructions is stored.
A translation lookaside buffer is a small hardware tableecontinang the results of recent address translation each entry in the tlb maps a virtual page to a physical page. Instead of finding the relevant entry by a multi level lookup or by hashing the TLB hardware checks all the entries simultaneously against the virtual page. If there is a match, the processor uses that henry ro form the physical affres skipping the rest of the steps of address translation. This is called a TLB hit. S TLB miss occurs if none of the entries in the TLB match. THe translation is done normally. A TLB also requires an address comparator for each entry tochek in parallel if there is a match. To reduce this cost, some TLbs are set to be associative. Compared to fully associated TLBs, set associative ones need fewer competitors but they may have a higher  miss rate.
***cost(address translation ) Cost(TLB lookup ) + cost(fullttanslation)*(1-p(hit))***

### 8.3.2 Superpages

One way to improve the TLB hit rate is using a concept called superpages superpage is a set of contiguous pages in physical money that mapo a contiguous region of virtual mem, where the pages are aligned so that they share the same height order address.

### 8.3.3 TLB Consistency

Whenever we introduce a cache into a system, we need to consider how to ensure consistency of the cache with original data when the neries ar emdoifed A Tlb is no exception. For secure and correct program execution the OS must ensure that the each program sees its memory evel translation table and the portableOS layer is a potential corrections and security flaw There are three issues to consider:

- Process context switch
- Permission reduction
- TLB shootdown

### 8.3.4 Virtual addressed caches

The same consistency issues that apply to TLBs apply to virtually addressed caches

### 8.3.5 Physically Addressed Caches

Physical addresses caches serve dual purposes:

- Faster memory reference
- Faster TLB misses

## 8.4 Software Protection

Why do we need software protection:

- Simplify hardware
- Application-level protection
- Protection inside the kernel
- Portable security

### 8.4.1 Single Language Operating Systems

A very simple approach to software protections is to restrict all applications to be written in a single, carefully designed programming language. If the language and its environment permits only safe programs to be expressed, and the compiler and runtime system are trustworthy then no hardware protection is needed.  Unfortunately language-based software protection has some practical limitations, so that on modern systems. It is often used in tandem, rather than as a

replacement for hardware protection. USing an interpreted language seems like a safe option, but requires trust in both the interpreter and its runtime libs.

### 8.4.2 Language-Independent Software Fault isolation

A limitation of trusting a language and its interpreter or compiler to provide safety is that many programmers value the flexibility to choose their own programming language. Since it would be dumb for the OS to trust every compiler.

# Chapter 9 Caching and Virtual Memory

Regardless of the context, all caches face three design challenges:

- Locating the cached copy
- Replacement policy
- Coherence

## 9.1 Cache Concept

The simplest kind of cache is a memory cache. It stores pairs. When we need to read the value of a certain memory location, we first consult the cache, and it either replies with the value and otherwise it forwards the request onward. If the cache has the value, that is called a cache hit. If the cache does not, that is called a cache miss. The cost of a cache hit must be less than a miss, or we would just skip using the cache. Second, the likelihood of a cache hit must be high enough to make it worth the effort. One source of predictability is temporal locality programs tend to reference the same instructions and data that they had recently accessed. Another source of predictability is spatial locality. Programs tend to reference data near other data that has been recently referenced. A related design technique that also takes advantage of spatial locality is to prefetch data into the cache before it is needed.

$$\begin{aligned}\text{Latency (read request)} = {} & \text{Prob(cache hit)} * \text{Latency(cache hit)} \\ & + \text{Prob(cache miss)} * \text{Latency(cache miss)}\end{aligned}$$

In the background the system checks if the address is in the cache. If not, the rest of the cache block must be fetched from memory and then updated with the changed value. Finally if the cache is write-through, all updates are sent immediately onward to memory. If the cache is write-back, updates can be stored in the cache, and only sent to memory when the cache runs out of space and needs to evict a block to make room for a new memory block.

## 9.2 Memory Hierarchy

- First-level cache: close to the processor designed to keep the processor fed with instructions and data at the clock rate of the processor
- Second-level cache: Handles misses from the first-level
- Third-level cache: Handles misses from the second-level, and is normally shared across all of the on-chip processor cores.
- First and second level TLB
- Main memory(DRAM)
- Data center memory (DRAM)
- Local disk or non volatile memory
- Data center disk
- Remote data center disk

## 9.3 When Caches Work and When They Do Not

As the cache grows in size the hit rate grows in size.

### 9.3.1 Working Set Model

Regardless of the program, a sufficiently large cache will have a high cache hit rate. The miss rate will be zero once the data is loaded into the cache. At the other extreme , a sufficiently small cache will have a very low hit rate. The cache can have a phase change behavior meaning that different operations done can affect the hit rate of the cache.

### 9.3.2 Zipf Model

Web access pattern cause two challenges toa cache designer:

- New Data: New pages are being added to the web at a rapid rat, and page contents also change. Every time a user accesses a page the system needs to check whether the page has changed in the meantime.
- No working set: Although some web pages are much more popular than others, there is no small subset of web pages that, if cached, give you the bulk of the benefit. Unlike with a working set, even very small caches have some value. Conversely, increasing cache size yields diminishing return even very large caches tend to have only modest cache hit rates as there is an enormous group of pages that are visited from time to time.

Zipf distribution. Zipf developed the model to describe the frequency of individual words in a text, but it also applies in a number of other settings . Suppose we have a set of web pages, and we rank them in order of popularity. Then the frequency user visits a particular web page is inversely proportional to its rank. A characteristic of a Zipf curve is a heavy tail. Although a

significant number of references will be to most popular items, a substantial portion of references will be to less popular ones.

## 9.4 Memory cache Lookup

We can use a linked list, a multi level tree or a hash table. Operating systems use each of those techniques in different settings, depending on the size of the cache, its access pattern and how important it is to have a very rapid lookup. FOr hardware caches the design choices are more limited. Three common mechanism for cachelook up are:

- Fully associative: array storages
- Direct mapped: Hash table storage
- Set associative: Map/Dictionary storage

## 9.5 Replacement Policies

Policies vary depending on the setting hardware cache use a different replacement policy than operating system does in managing main memory as a cache for disk.

### 9.5.1 Random

Although it may seem arbitrary, a practical replacement policy is to choose a random block to replace. Particularly for a first-level hardware cache, the system may not have time to make more complex decisions, and the cost of making the wrong choice can be small if the item is in the next level cache.

### 9.5.2 First-In-First-Out (FIFO)

A less arbitrary policy is to evict the cache block or page that has been in memory the longest, that is FIFO. Particularly for using memory as a cache for disk, this can seem fair; each programæs pages spend a roughly equal amount of time in memory before being evicted. Unfortunately, FIFO can be the worst possible replacement policy for workloads that happen quite often in practice.  ON a repeated scan through memory, FIFO does exactly the wrong thing; it always evicts the block or page that will be needed next.

### 9.5.3 Optimal Cache Replacement (MIN)

The optimal policy called MIN, is to replace whichever block is used farthest in the future. Equivalently, the worst possible strategy is to replace the block that is used soonest. As with Shortest Job First, MIN requires knowledge of the future, and so we can not implement it directly. Rather, we can use it as a goal we want to come up with mechanisms which are effective at predicting which block will be used in the near future, so that we can keep those in the cache.

### 9.5.4 Least Recently Used (LRU)

One way to predict the future is to look at the past. If programs exhibit temporal locality,  the location they reference in  the future are likely to be the same as the ones they have referenced in the recent past. A replacement policy that captures this effect is to evict the block that has not been used for the longest period of time or the least recently used block. In software, LRU is simple to implement on every cache hut, you move the block to the form of the list, and on a cache miss, you evict the block at the end of the list. In hardware, keeping a linked list of cached blocks is too complex to implement at high speed. Instead we need to approximate LRUm and we will discuss exactly how in a bit.

### 9.5.5 Least Frequently Used (LFU)

LFU fiscards the block that has been used least often; it therefore keeps popular pages, even when less popular pages have been touched more recently. LRU and LFU both attempt to predict the future behavior, and they have complementary strengths. Many systems meld the two approaches to gain the benefits of each. LRU is better at keeping the current working set in memory once the working set is taken care of, however, LRUI will yield diminishing returns. INsted, LFU may be better at predicting what files or memory blocks will be needed in the more distant future after the next working phase change.

### 9.5.6 Belady's Anomaly

In some cases adding space to a cache can actually hurt its hit rate. This is called Belady’s anomaly, after the person that discovered it. FIFO suffers from this.

## 9.6 Case Study : Memory-Mapped Files

### 9.6.1 Advantages

- Transparency
- :Zero copy I/O
- Pipelining
- Interprocess communication
- Large files

### 9.6.2 Implementation

When a process touches an invalied mapped address, a sequence of event soccurs:

- TLB miss
- Page table exception
- Convert virtual address to file offset
- DIsk block read

- Disk interrupt
- PAge table update
- Resume process
- TLB miss
- Page table fetch
- Select a page to evict
- Find Page table entries that point to evicted page
- Set each page table entry to invalid
- Copy back any changes to the evicted page

### 9.6.3 Approximating LRU

A further challenge to implementing demand paged memory-mapped files is that the hardware does not keep track of which pages are recently or least frequently used. Doing so would require the hardware to keep a linked list of every page in memory, adn to modify that list on every load and store instructions. Most processor architectures keep a use bit in each page table entry. The OS clears the use bit when the page table entry is initialized; the bit is set in the hardware whenever the page table entry is brought into the TLB. As with the dirty bit, a physical page is sussed if any of the page table entries have their use bit set. The OSD can leverage the use bit in various ways, but a commonly used approach is the clock algorithm. Periodically, th eOS scans through the core map of physical memory pages. For each page frame, it records the value of the use bit in the page table entries that point to that frame, and then clears their use bits. Because the TLB can have a cache dcopt of the translation, the OS also does a shootdown for any page table entry where the use bit is cleared. The policy for what to do with the usage information is up to the OS kernel.  Common policy is called nor recently used, or k'th chance. If th eOS needs to evict a page, the kernel picks one that has not been used for ath elas k sweeps of the clock algo. The algo prationsoins pages based on hwo recently they have been used.

## 9.7 Case study Virtual Memory

The advantages of virtual memory is flexibility. The system can continue to function even though the user has started more processes that can fit in main memory at the same time. The OS simply makes room for the new processes by paging memory of idle applications to disk.

### 9.7.1 Self-Pagin

One consideration is that the behavior of one process can significantly hurt the performance of other programs running at the same time. “Pig " It allocates an array in virtual memory equal in size to physical memory, it then uses multiple threads to cycle through memory, causing each page to be brought in while the either pages remain very recently used. A normal program sharing memory with the pig will eventually be frozen out of money and stop making progress.

When the pig touches a new page it triggers a page fault but all its pages are recently used because of the background thread. Meanwhile the normal program will have recently touched many of its pages but there will be some that are recently used. The clock algo will choose those for replacement. A widely adopted solution is self-paging With self--påaging wach process or user is assigned its fair share of page frames, using the max-min scheduling algo. If all of the active processes can fit in memory at the same time, the system does not need aage. As the system starts to page, it evicts the page form whichever process has the most allocated to it. This pig would only be able to allocate its fair share of page frames.

### 9.7.2 Swapping

If the working sets of the applications easily fit in memory, then as page faults occur, the clock algo will find lightly used pages that is those outside of the working set of any process to evict to make room for new pages As we keep adding active process however their working sets matt no longer fit even if each process is given their fair share of memory. In this case the performance of the system will degrade dramatically. Evicting an entire process from memory is called swapping. When there is too much paging activity, the OS can prevent a catastrophic degradation in performance by moving all of the page frames of a particular process to disk, preventing it from running at all. Although this may seem terribly unfair, the alternative is that every process, not just the swapped process, wil return much more slowly. By distributing the swapped process's pages to other ricesses, we can reduce the number of page faults, allowing the system to recover.