What is an OS?

• Interface between application programs and hardware
• Ultimate control program
  – Exploits hardware resources of one or more processors to provide a set of services to users
  – Coordinates the use of hardware among various application programs for different users
  – Manages secondary memory and I/O devices on behalf of its users
• Two different views of an OS
  1. Extended machine view
     – Virtual machine that is easier to understand and program
     – Tool to make programmer’s job easy
  2. Resource manager view
     – Tool to facilitate efficient operation of computer system
     – Provides services to users; processor, memory, I/O, system bus
     – Must be fair; not partial to any process, specially for process in the same class
     – Must discriminate between different class of jobs with different service requirements
     – Do the above efficiently
       * Within the constraints of fairness and efficiency, an OS should attempt to maximize throughput, minimize response time, and accommodate as many users as possible

Basic elements

• A computer has a number of modules, with possibly more than one instance of each
• Processor/CPU
  – Controls the operations of computer and performs data processing functions
  – Exchanges data with memory using memory address register and memory buffer register
  – Exchanges data with I/O devices using I/O address register and I/O buffer register
• Main memory/Primary memory
  – Stores data and programs
  – Typically volatile
  – Abstracted as a set of locations, defined by sequentially numbered addresses
  – Each location contains a bit pattern to be interpreted as either an instruction or data
• I/O modules/architecture
  – Move data between a computer and external environment
  – Communicate with a variety of devices including secondary memory (disks), communications equipment, and terminals
  – Data flows between CPUs, RAM and I/O devices over buses
  – System bus
    * Provides for communications among processors, main memory, and I/O modules
    * Connects most of the internal hardware devices
    * PCI bus, ISA bus, EISA bus, SCSI, USB
Introduction

* A typical computer may have several buses of different types, linked by bridges
  - Frontside Bus
    * Connects the CPUs to the RAM controller
  - Backside bus
    * Connects the CPUs directly to the external hardware cache
  - System bus linked to frontside bus by the host bridge

Evolution of the microprocessor

- Most of the current microprocessors move bits around in subnanoseconds timeframes, making them extremely fast in computation
- Almost all the current microprocessors contain multiple CPUs (cores) making them behave as multiprocessors
- Multiple levels of large memory cache on the chip
- Multiple logical processors sharing the execution units of each core, leading to hyperthreading
- Numerical computations on GPUs – GPGPU using SIMD
  - Modern CPUs have the capability to operate on arrays of data by integrating vector units into processor architecture
- DSPs to deal with streaming audio and video signals
- System on a Chip (SoC), for handheld devices

Instruction execution

- Simplest form is a two-step process (instruction cycle) in a loop
  1. Processor fetches instruction from memory (fetch stage)
     - Address of next instruction to be executed is contained in program counter (PC) register
  2. Processor executes the fetched instruction (execution stage)
- Execution halts only if the processor is turned off, some sort of unrecoverable error occurs, or an explicit instruction to halt the processor is encountered
- Fetched instruction is loaded into instruction register (IR)
- Processor interprets the instruction and takes the specified action, broadly classified into four categories
  1. Processor-memory – Data transfer between processor and memory
  2. Processor-I/O – Data transfer between processor and I/O module or device (peripheral device)
  3. Data processing – Arithmetic or logic operation on data
  4. Control – Alter the sequence of instructions
- An instruction could also be a combination of the above actions

Interrupts

- Mechanism to interrupt the normal execution of sequence of instructions by a CPU; an event that alters the sequence of instructions executed by CPU
Events correspond to electrical signals generated by hardware circuits inside/outside the CPU

- Help to improve CPU utilization
- Consider a CPU working at 3GHz, or $3 \times 10^9$ instructions per second
- A typical hard disk drive will have a rotational speed of 7200 RPM, giving half track rotation time of 4ms
- In 4ms, our CPU would have executed 12 million instructions

- Different classes of interrupts

  **Program**  Based on instruction execution, such as divide by zero
  **Timer**  Generated by a clock in the CPU to allow a function execution at a specified time or after an interval
  **I/O**  Generated by I/O controller, to indicate the status of an I/O request
  **Hardware failure**  Power failure or memory parity error

- Synchronous interrupts/Exceptions
  - Produced by the CPU control unit
  - Control unit issues them only after terminating the execution of an instruction

- Asynchronous interrupts
  - Generated by other hardware devices at arbitrary times with respect to CPU clock signals

- Most of the tasks performed by a kernel are in response to a process request by
  - Dealing with a special file
  - Sending an ioctl
  - Issuing a system call

- Another important task for the kernel is to talk to the hardware connected to the machine

- Two types of interactions between CPU and the rest of computer’s hardware
  1. CPU giving orders to the hardware
  2. Hardware may need to tell something to the CPU – interrupt

- Interrupts are harder to implement because it has to be dealt with when convenient to hardware, not the CPU
  - Hardware devices typically have a very small amount of RAM and if you don’t read the information when available, it will be lost

- Interleaving of CPU and I/O instructions
  - The I/O operation may take a while to execute; hence user process is stopped and CPU may be scheduled to execute another process

- Interrupts and instruction cycle
  - Synchronous or asynchronous execution of instructions and I/O
    - When external device is ready, I/O module sends an interrupt request to CPU
    - CPU responds by suspending current process and branches to interrupt handler; it returns to original process after servicing the request
    - Interrupt may occur at any point in the main program’s execution; it may not be dependent on the execution of a certain instruction
    - User process does not need any special code to accommodate interrupts; CPU and OS are responsible to suspend the process and resume it appropriately
• Interrupt stage
  – Added to the instruction cycle

  ![Diagram of the instruction cycle with Interrupt stage]

  – If there is an interrupt, indicated by an interrupt flag, CPU goes to an *interrupt handler* routine
    * Interrupt handler looks at the type of interrupt and executes appropriate function
    * Involves overhead but still better than the CPU waiting for I/O completion or other activities
  – Interrupt handler must satisfy the following constraints
    * Very fast: Interrupt handler activities contain an urgent part to be executed right away and a deferable part that can be handled later
      - Block of data arrives on network line
      - Kernel marks the presence of data (urgent part) and gives CPU back to process that was running before
      - Rest of processing can be done later (moving data to buffer where recipient will find it)
  – Interrupt vector
    * Table of pointers in memory that contains the addresses of interrupt service routines
    * At a fixed location for a given CPU

• Interrupt processing
  1. Device issues interrupt to CPU
  2. CPU finishes execution of current instruction
  3. CPU tests for pending interrupt request; if there is one, it sends an acknowledgment to the device which removes its interrupt signal
  4. CPU saves program status word onto control stack
  5. CPU loads the location of interrupt handler into PC register
  6. Save the contents of all registers from control stack into memory
  7. Find out the cause of interrupt, or interrupt type; invoke appropriate routine
  8. Restore saved registers from the stack
  9. Restore PC to dispatch the original process

• Multiple interrupts
  – Ability to handle nested exceptions
  – Disable the interrupts, with new interrupts getting queued and handled after the current interrupt is processed
    * Does not account for interrupt priority or time-critical needs
    * Data may be lost, especially if the I/O device buffer is small
  – Define interrupt priorities and allow a higher-priority interrupt to interrupt a lower-priority interrupt handler
– Interrupts must be disabled inside some critical kernel regions
  * Critical regions must be limited because kernel and interrupt handlers should be able to run most of the time to take care of any event
– Typically, interrupt handler will start with interrupts disabled\(^1\) (blue part in interrupt processing)
  * Allows exclusive use of a few registers, including PC that must be saved somewhere
– Some architectures allow an interrupt to be interrupted by a higher priority interrupt
  * ARM processors have two interrupt levels
    1. IRQ – Normal interrupts
    2. FIQ – Fast interrupts
  * Each type has its own register bank for PC and SP, and their own interrupt vector
    · IRQ handler starts with IRQ disabled but FIQ enabled
    · FIQ handler starts with both IRQ and FIQ disabled
    · FIQ can be triggered in the middle of handling an IRQ

• Types of interrupts
  – Maskable interrupts
    * All IRQs issued by I/O devices
    * Can be in two states: masked or unmasked
    * Masked interrupts are ignored by control unit unless unmasked
  – Nonmaskable interrupts
    * Generated by a few critical events, such as hardware failure
    * Always recognized by the CPU

• Types of exceptions
  – Processor-detected exceptions
    * Detected as an anomalous condition while executing an instruction
    * Further divided into three groups
      1. Faults
        · Can be corrected
        · Process restarts after correction with no loss of continuity
      2. Traps
        · Reported by the process after the kernel returns control to the process
        · Triggered when there is no need to reexecute the instruction that terminated
        · Used primarily for debugging
        · Signals the debugger that a specific instruction has been executed (breakpoints)
        · Execution continues from the following instruction, after the data is examined by the user
      3. Aborts
        · Used to report serious errors, such as hardware failures or invalid/inconsistent values in system tables
        · Handler forces the affected process to terminate
  – Programmed exceptions
    * Occur at the request of programmer
    * Handled by control unit as traps; software interrupts
    * Used to implement system calls and to notify debugger of a specific event

• Interrupts under Linux

\(^1\)This part from Gilles on Stack Overflow
– Known as IRQs – interrupt requests
– Short IRQs: Expected to take a very short period of time during which the rest of the machine is blocked and no other interrupts are handled
– Long IRQs: can take longer; other interrupts may occur (but not from the same device)

• Handling interrupts in Linux
  – When CPU receives an interrupt, it stops whatever it is doing; unless it is processing a more important interrupt in which case it will handle the current interrupt when the important one is done
  – Save parameters on stack and call interrupt handler
  – System is in an unknown state and therefore, certain things are not allowed in the interrupt handler itself
  – Interrupt handler should do whatever it needs to do immediately (read from or write to hardware), and then, schedule the handling of new information at a later time (known as the bottom half) and return
  – Kernel is then guaranteed to call the bottom half as soon as possible; at that time, everything that is allowed in kernel modules will be allowed
  – Each interrupt/exception identified by a number in the range (0,255) called a vector
    * Vectors of nonmaskable interrupts and exceptions are fixed
    * Vectors of maskable interrupts can be altered by programming the interrupt controller

• Signals
  – Mechanism to allow interactions between user mode processes
  – Notify processes of system events
  – A short message that may be sent to a process or a group of processes
    * The message contains only a number to identify the signal; there is no room for an argument or other accompanying information

Processor registers

• User-visible registers
  – Areas in CPU to be used as temporary memory locations
  – Accessed at CPU speed

• Control and status registers
  – Used to control CPU operation or privileged OS instructions

Memory Hierarchy

• Memory hierarchy based on storage capacity, speed, and cost
  – As capacity increases, applications grow to use it
  – Need for speed to keep up with the improvement in CPU speed
  – Higher the storage capacity, lesser the speed, and lesser the cost per bit

• Main memory or RAM
  – Implemented using a semiconductor technology called DRAM
  – Provides an array of bytes, with the index of the array providing the address of the byte
• Different memory levels, in decreasing cost per byte of storage

<table>
<thead>
<tr>
<th>Memory Level</th>
<th>Storage Capacity</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Few bytes</td>
<td>Almost CPU speed</td>
</tr>
<tr>
<td>Cache memory</td>
<td>Few megabytes</td>
<td>Nanoseconds</td>
</tr>
<tr>
<td>Main memory</td>
<td>Gigabytes</td>
<td>Microseconds</td>
</tr>
<tr>
<td>Magnetic disk</td>
<td>Terabytes</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>USB/Optical disk</td>
<td>No limit</td>
<td>Off-line storage</td>
</tr>
</tbody>
</table>

• Use hierarchical memory to transfer data from lower memory $M_2$ (away from CPU) to higher memory $M_1$ (closer to CPU) to be executed
  - Smaller/expensive/faster memory is supplemented by larger/inexpensive/slower memory
  - Hit ratio and access time
    * Hit ratio $H$ is the fraction of all memory accesses that are found in the faster memory
    * For higher values of $H$, the access time is much closer to that of higher memory
    * Let $T_1$ be the access time in smaller/faster memory and $T_2$ be the access time for larger/slower memory
    * The average access time $T_s$ is given by
      \[
      T_s = H \times T_1 + (1 - H) \times (T_1 + T_2) = T_1 + (1 - H) \times T_2
      \]
    * Consider two levels of memory with access times of $0.1\mu s$ and $1.0\mu s$
      - Average access time is given by
        \[
        0.95 \times 0.1\mu s + 0.05 \times (0.1\mu s + 1.0\mu s) = 0.095 + 0.055\mu s = 0.15\mu s
        \]

• Locality of reference
  - Most of the references in the memory are clustered and move from one cluster to the next
    * Holds true for both instructions and data
    * Clusters change over long periods but over a short period, CPU works with only a limited set of memory locations
  - Reasons for clustered references
    * Programs execute sequentially, one instruction after the next, and have relatively fewer branches in code; next instruction typically follows the current
    * Rare to have long sequences of function/procedure calls
    * Iterative constructs have a small number of instructions
    * Data structures such as arrays or sequences of records make references to nearby locations in successive instructions
  - Spatial locality
    * Reference to memory locations that are clustered
    * Sequential instruction execution
    * Accessing data location sequentially (arrays)
    * Exploited by prefetching mechanisms in pipelined architectures
  - Temporal locality
    * Accessing locations that have been used recently
    * Executing same instructions repeatedly in a loop (function calls)
  - Organize code and data across hierarchy
* Percentage of accesses to lower level should be substantially less than the accesses at higher level
* Move clusters of memory from higher to lower level

– Volatility

– Cache memory
  * Use of very fast memory (a few MB) designated to contain data for fast access by the CPU (close to CPU speed)
  * May not be visible to programmer or system
  * Used to temporarily hold data for transfer between main memory and registers to improve performance
  * Exploits temporal locality
  * Transfer a block of data from memory to cache, using locality of reference

– Virtual memory or extension of main memory

– Disk cache
  * Designating a portion of main memory for disk read/write
  * Improves performance by
    · Clustering disk writes; perform fewer large data transfers instead of many small transfers
    · Data destined for disk can be read back before it is written

• Memory management

– Memory management is one of the most important services provided by the operating system

– An operating system has five major responsibilities for managing memory:
  1. Process isolation
     * Should prevent the independent processes from interfering with the data and code segments of each other
  2. Automatic allocation and management
     * Programs should be dynamically allocated across the memory depending on the availability (may or may not be contiguous)
     * Programmer should not be able to perceive this allocation
  3. Modular programming support
     * Programmers should be able to define program modules
     * Programmers should be able to dynamically create/destroy/alter the size of modules
  4. Protection and access control
     * Different programs should be able to co-operate in sharing some memory space
     * Contrast this with the first responsibility
     * Make sure that such sharing is controlled and processes should not be able to indiscriminately access the memory allocated to other processes
  5. Long-term storage
     * Users and applications may require means for storing information for extended periods of time
     * Generally implemented with a file system
     – OS may separate the memory into two distinct views: physical and logical; this division forms the basis for virtual memory

• Memory bandwidth

– More core and bigger caches imply that processor speed increases at a faster rate than memory sub-system components
  * If a system does not move data between memory and cores fast enough, some cores sit idle
  * Increase in overall computation time, negating the benefits of multiple cores

– Performance improvement by using non-uniform memory access NUMA instead of front side bus FSB, reducing the bandwidth saturation
Multiprocessor and multicore organization

• Traditional view of computer as a sequential machine
  – Instructions executed in sequence, one at a time
  – Instruction execution cycle
• Instruction pipelines
• Multiprocessors have two or more processors in close communication
  – Share system bus, clock, memory, and peripheral devices
  – Result in increased throughput, economy of scale, and better reliability
• Symmetric multiprocessing / SMP
  – There are two or more processors of similar capability
    * Any process and any thread can run on any available processor
  – Processors share the same memory and I/O, through a common communications bus, leading to same memory access time for each CPU
  – All processors share the same I/O devices, either through the same channels or through different channels that provide paths to the same device
  – Shared memory is used to support communication and synchronization among processors
  – System controlled by an integrated OS providing interaction between PEs and processes at the job/task/file/data element levels
    * A very high degree of cooperation between PEs compared to cluster
  – Getting more popular due to advantages such as performance, availability, incremental growth, and scaling from hardware perspective
• Multicore computers
– Combine two or more processors or cores on a single die
– Each core contains all the components of a CPU, including multiple levels of cache
– Cores share the system bus, clock, memory, and peripheral devices
– Offer several advantages including increased throughput, economy of scale, and increased reliability
– Support three levels of parallelism
  1. Instruction level parallelism such as pipelining
  2. Multiprogramming/multithreading in each processor
  3. Single application executing in concurrent processes/threads across multiple cores

The last two types require support from the OS

• Parallelism within applications
  – Problem of splitting applications into independently executable tasks
  – Parallel design process supported by compiler and programming language
  – OS role in allocating resources among parallel tasks
  – Grand Central Dispatch (GCD)
    * Multicore support library to help break up an application into multiple concurrent parts
    * “Islands of serialization in a sea of concurrency”
    * Tasks expressed as function or block (anonymous function)
    * Tasks are queued up for execution and scheduled to execute on any of the available processor cores
    * Uses threads at lower level but abstracts them away from the programmer

• Array processing / SIMD
• General multiprocessing / MIMD
• Cluster computing and fault tolerance
  – Group of individual loosely-coupled computers equipped with their own OS and linked by a high speed interconnection
  – A virtual server accepts requests and directs them to one in a series of servers
  – High degree of availability
    * Each node monitors one or more of the other nodes over LAN
    * One node goes down, others can take over its work
  – Provides load balancing, fault tolerance, and higher scalability
  – Suitable for high-end, transaction processing applications
  – Typical applications include firewalls, web servers, domain name servers, and mail servers
  – Grid computing
    * Linking a number of separate fully-functional computers into a large cluster on demand
    * Applications run like a screen saver to take advantage of spare CPU cycles

• Cloud Computing
  – Software as a service, or Internet as a platform
  – Migration of data and programs from desktop PCs and corporate server rooms to the compute cloud
  – Google docs, Apple iCloud, Microsoft Office 365
  – No central computer as required in time sharing topology
  – No infrastructural investment in terms of OS updates and related changes to locally developed software; more mobility and collaboration
Introduction

– From vendor perspective, no need to develop software for diverse arrays of platforms (Turbo Tax); quick deployment of bug fixes and updates
  * Server side software still needs to interact with a wide variety of clients
– Cloud OS
  * User interface resides in a single window in web browser
  * OS inside a browser – eyeOS
  * Bypass the web browser; more capable software runs as a separate application on client computer and communicates directly with servers in cloud – AIR from Adobe
– Application scalability
  * Coordinate information coming from multiple sources
  * Many-to-many communication – each server talks to multiple clients and each client may invoke programs on multiple servers
  * Platform and language issues; and culture of users

Modern classification for OS based on use

• Low-end consumer desktop
  – Assumes no intelligence on part of the user
  – Focuses on hardware and software compatibility
• High-end desktop
  – User is smart but may not be very computer savvy
  – Must provide good performance for applications, such as CAD/CAM
  – Provide hardware and software compatibility
• Real-time systems
  – User is professional programmer
  – Focus on performance, defined response time, easy extendable hardware support, and programming control
• Embedded systems
  – Closed box system to do specific functions well
  – No expansion slots (standard machine has multiple standard interfaces)
  – Limited applications
  – Turn it on and use it (standard host enables all components, scans every interface, and is ready to run multiple apps)

Operating System Concepts

• Program
  – Collection of instructions and data kept in ordinary file on disk
  – Executable program, complete code output by linker/loader, with input from libraries
    * Generated from source program and object code
  – The file is marked as executable in the i-node
  – File contents are arranged according to rules established by the kernel
• Process
– Created by kernel as an environment in which a program executes
– May be stopped and later restarted by the OS
– Process executes a single sequence of instructions in an address space
  * Address space is the set of memory addresses that the process is allowed to reference
– Core image
  1. Instruction segment
  2. User data segment
  3. System data segment
    * Includes attributes such as current directory, open file descriptors, and accumulated CPU times
    * Information stays outside of the process address space
– Program initializes the first two segments
– Process may modify both instructions (rarely) and data
– Process context
  * All the metadata about process
  * Terminal, open files, register contents
  * Recorded in process table
– Process may acquire resources (more memory, open files) not present in the program
– Child and parent processes
– Communication between processes through messages
– Process id, UID, GID

• Threads
– Stream of instruction execution; running instance of a process’ code
  * Inefficient to create a separate process for some short-lived activities
  * Some applications may need to use memory between different non-overlapping activities
– A dispatch-able unit of work to provide intraprocess concurrency, without the overhead to create and manage separate processes
– A process may have multiple threads of execution in parallel, each thread executing sequentially
  * With threads, a process can execute multiple sequences of instructions in the same address space
– Threads may execute in time-sharing manner on a single CPU, on multiple cores concurrently, or on multiple CPUs

• Files
– Services of file management system to hide disk/tape specifics
– System calls for file management
– Directory to group files together
– Organized as a hierarchical tree
– Root directory
– Path name
– Path name separator
– Working directory
– File descriptor or handle – small integer to identify a file in subsequent operations, error code to indicate access denied
– I/O device treated as a special file
  * Block special file
* Character special file
  - Pipe – pseudo file to connect two processes

** System calls
  - Interface between user program and operating system
  - Set of extended instructions provided by the operating system
  - Applied to various software objects like processes and files
  - Invoked by user programs to communicate with the kernel and request services
  - Access routines in the kernel that do the work
  - Executing system calls in Linux
    * Process fills the register with appropriate values
    * Calls a special instruction to jump to a previously defined location (known as `system_call`) in the kernel
    * Done by interrupt `0x80` in Intel CPUs
    * Procedure at `system_call` checks the system call number to determine the service requested
    * Look into `sys_call_table` to see the address of kernel function to call
    * Call the function; on return, do a few system checks, and return control back to the process
  - Library functions corresponding to each system call
    * Machine registers to hold parameters of system call
    * Trap instruction (protected procedure call) to start OS
    * Hide details of trap and make system call look like ordinary procedure call
    * Return from trap instruction
  - Broadly divided into six classes: filesystem (`open`), process (`fork`), scheduling (`priocntl`), IPC (`semop`), socket/networking (`bind`), and miscellaneous (`time`)
    * The scheduling related system calls on p. 100 of the text are actually library functions (section 3RT in Solaris)
    * The socket/networking is implemented as library functions in Solaris but as system calls in FreeBSD
  - System calls vs library functions
    * System calls run in kernel mode on user’s behalf; provided by kernel itself
    * Library functions run completely in user space and provide a more convenient interface for the programmer to the functions that do the real work – system calls
    * System calls corresponding to `printf` (use `strace` on `opsys`)

** Shell
  - Unix command interpreter
  - Is a user program and not part of the kernel
  - Prompt
  - Redirection of input and output
  - Background jobs
  - For most commands, the shell `forks` and the child `execs` the command associated with the name, treating the remaining words on the command line as parameters to the command
  - Allows for three types of commands:
    1. Executable files
    2. Shell-scripts
    3. Built-in shell commands

** Kernel
– Minimal set of functions to manage system resources
– Permanently resides in the main memory
  * Loaded into memory by the bootstrap loader when the computer is powered on
  * Bootstrap loader is stored in ROM or firmware
– Controls the execution of processes by allowing their creation, termination or suspension, and communication
– Schedules processes fairly for execution on the CPU
  * Processes share the CPU in a time-shared manner
    · CPU executes a process
    · Kernel suspends it when its time quantum elapses
    · Kernel schedules another process to execute
    · Kernel later reschedules the suspended process
– Allocates main memory for an executing process
  * Allows processes to share portions of their address space under certain conditions, but protects the private address space of a process from outside tampering
  * If the system runs low on free memory, the kernel frees memory by writing a process temporarily to secondary memory, or swap device
  * If the kernel writes entire processes to a swap device, the implementation of the Unix system is called a swapping system; if it writes pages of memory to a swap device, it is called a paging system.
  * Coordinates with the machine hardware to set up a virtual to physical address that maps the compiler-generated addresses to their physical addresses
– File system maintenance
  * Allocates secondary memory for efficient storage and retrieval of user data
  * Allocates secondary storage for user files
  * Reclaims unused storage
  * Structures the file system in a well understood manner
  * Protects user files from illegal access
– Allows processes controlled access to peripheral devices such as terminals, tape drives, disk drives, and network devices.
– Services provided by kernel transparently
  * Recognizes that a given file is a regular file or a device but hides the distinction from user processes
  * Formats data in a file for internal storage but hides the internal format from user processes, returning an unformatted byte stream
  * Allows shell to read terminal input, to spawn processes dynamically, to synchronize process execution, to create pipes, and to redirect I/O
– Kernel in relation to other processes
  * In older operating systems, kernel executed outside of any process
  * Kernel has its own address space and its own system stack to control procedure calls and returns
  * In such systems, the concept of process only applies to user programs; OS code executes in privileged mode
– Micro-kernel architecture
  * Smaller set of operations in a limited form assigned to kernel
    · Each OS layer is a relatively independent program that must interact with other layers through well-defined and clean software interfaces
    · Interprocess communication, limited process management and scheduling, and some low-level I/O
    · Other OS services provided by processes, or servers, running in user mode
  * Less hardware specific because many system-specific operations are pushed into user space
· Existing microkernel OS can be easily ported to other architectures because all hardware-dependent components are encapsulated in microkernel code

* Simplifies implementation, provides flexibility, and is better suited for distributed environment
* Microkernel OS makes better use of RAM than monolithic because unneeded system processes may be swapped out or terminated
* Microkernels are slower than monolithic ones because they require explicit message passing between different layers of the OS, incurring a cost penalty

– Kernel in Unix

* Traditionally, the operating system itself
* Isolated from users and applications
  · Unix philosophy – Keep kernel as small as possible
  · A bug in an application should not crash the OS
* Kernel itself is not a process but a process manager
* Processes use a system call to request a kernel service
  · At the top level, user programs invoke OS services using system calls or library functions
  · At the lowest level, kernel primitives directly interface with the hardware
* Kernel itself is logically divided into two parts:
  1. File subsystem to transfer data between memory and external devices
  2. Process control subsystem to control interprocess communication, process scheduling, and memory management
* Operations include scheduling, process management, signaling, device I/O, paging, and swapping

– Kernel in Linux

* Monolithic in nature
  · One large block of code
  · Runs as a single process within a single address space
  · All functional components have access to all internal data structures and functions
  · If anything is changed, all modules and functions must be recompiled and relinked and system rebooted
  · Difficult to add new device driver or file system functions
* Structured as a collection of modules that can be automatically loaded and unloaded on demand (loadable modules)
  · Loadable modules overcome some of the problems of monolithic kernel
  · Dynamically linked
  · Stackable modules – arranged as a hierarchy
* Linux kernel modules
  · Intended to achieve theoretical advantages of microkernels without incurring performance penalty
  · Designed as object files whose code can be dynamically linked/unlinked to kernel at runtime
  · Object code is a set of functions to implement filesystem, device driver, or other features at kernel’s upper layer
  · Module does not run as a specific process but is executed in kernel mode on behalf of the current process, like any other statically linked kernel function
  · Advantages of using modules
    Modularized approach – well-defined software interfaces to access data structures handled by modules
    Platform independence – a disk driver module relying on SCSI standard works as well on a PC as on Alpha
    Frugal main memory use – module may be linked to a running kernel when needed and unlinked when no longer needed; useful for small embedded systems
    No performance penalty – after linking, the object code of a module is equivalent to the object code of a statically linked kernel; no explicit message passing is required to invoke module functions
  · Runs in it own memory space (memory is divided into kernel space and user space)
* Being monolithic, it includes low-level interaction with the hardware, and hence is specific to a particular architecture
* Linus Torvalds chose monolithic architecture for Linux because
  1. Micro-kernels were experimental at the time Linux was designed
  2. Micro-kernels were more complex than monolithic kernels
  3. Micro-kernels executed notably slowly than monolithic kernels

Kernel in Windows
* Core services such as memory management, process and thread management, security, I/O, and IPC provided by executive
* Microsoft calls it a modified micro-kernel architecture
  · Unlike pure micro-kernel, many of the systems functions outside the micro-kernel run in kernel mode for performance reasons
* Kernel works in cooperation with executive
  · Manages thread scheduling, process switching, exception and interrupt handling, and multiprocessor synchronization
    Kernel’s own code does not run in threads
    Only part of the OS that can not be preempted or paged
  · As with Unix, it is isolated from user programs, with user programs and applications allowed to access one of the protected subsystems
    Each system function is managed by only one component of the OS
    Rest of the OS and all applications access the function through the responsible component using a standardized interface
    Key system data can be accessed only through the appropriate function
    In principle, any module can be removed, upgraded, or replaced without rewriting the entire system or its standard application programming interface (API)
* Two programming interfaces provided by a subsystem
  1. Win32 interface – for traditional Windows users and programmers
  2. POSIX interface – to make porting of Unix applications easier
* Subsystems and services access the executive using system services
* Executive contains object manager, security reference monitor, process manager, local procedure call facility, memory manager, and an I/O manager

Process execution modes in Unix

• Two modes of process execution
  1. User mode
    – Normal mode of execution for a process
    – Execution of a system call changes the mode to kernel mode
    – Processes can access their own instructions and data but not kernel instructions and data
    – Cannot execute certain privileged machine instructions
  2. Kernel mode
    – Processes can access both kernel as well as user instructions and data
    – No limit to which instructions can be executed
    – Runs on behalf of a user process and is a part of the user process

Operating System Structure
• Minimal OS
  – CP/M or DOS
  – Initial Program Loading (Bootstrapping)
  – File system

• Monolithic Structure
  – Most primitive form of operating systems
  – No structure
  – Collection of procedures that can call any other procedure
  – Well-defined interface for procedures
  – No information hiding
  – Services provided by putting parameters in well-defined places and executing a supervisor call
    * Switch machine from user mode to kernel mode
  – Basic OS structure
    * Main program that invokes requested service procedures
    * Set of service procedures to carry out system calls
    * Set of utility procedures to help the service procedures
  – User program executes until
    * program terminates
    * time-out signal
    * service request
    * interrupt
  – Difficult to maintain
  – Difficult to take care of concurrency due to multiple users/jobs

• Layered Systems
  – Hierarchy of layers – one above the other

• Virtual machines
  – Basis for developing the OS
    * Create the illusion of having one or more objects to emulate the real object
    * Closely related to abstraction
      · Abstraction provides simplification by combining multiple simple objects into a single complex object
      · Virtualization provides diversification and replication by creating illusion of objects with desired characteristics
  – IBM System 370 – CMS, VM
    * Allowed a number of guest operating systems under the control of VM
    * Useful for cross platform development of software
    * Similar task is accomplished by VMware in recent times
· Provides support for different operating systems on the same machine
· Allows the different systems to run concurrently
  * Performs functions associated with CPU management and allocation
  * Provides synchronization and/or communication primitives for process communication

• Client-Server Model
  – Remove as much as possible from the OS leaving a minimal kernel
  – User process (client) sends a request to server process
  – Kernel handles communications between client and server
  – Split OS into parts – file service, process service, terminal service, memory service
  – Servers run in user mode – small and manageable

**I/O communication**

• Programmed I/O
  – Simplest and least expensive scheme
  – CPU retains control of the device controller and takes responsibility to transfer every bit to/from the I/O devices
  – Instruction set need I/O instructions for
    * Control: To activate and operate external devices (rewind tape)
    * Status: Test status conditions for I/O module and peripherals
    * Transfer: Read/write data
  – Bus
    * Address bus: To select a memory location or I/O device
    * Data bus: To transfer data
  – Handshaking protocol
  – Disadvantages:
    * Poor resource utilization
    * Only one device active at a time
    * Gross mismatch between the speeds of CPU and I/O devices

• Interrupt-driven I/O
  – CPU still retains control of the I/O process
  – Sends an I/O command to the I/O module and goes on to do something else
  – I/O module reads/writes data on the specified peripheral and places it on the data bus when instructed by CPU
  – I/O module interrupts CPU when it is ready to transfer more data
  – Possible to have multiple I/O modules on a machine

• Direct memory access
  – CPU trusts the DMA module to read from/write into a designated portion of the memory
  – DMA module (also called I/O channel) acts as a slave to the CPU to execute those transfers
  – DMA module takes control of the bus, and that may slow down the CPU if the CPU needs to use the bus

CPU and I/O overlap

• Hardware flag
  CPU is blocked if device is busy
• Polling by test-device-flag
• Memory-mapped I/O
  – Uses memory address register (MAR) and memory buffer register (MBR) to interact with I/O devices
  – I/O ports can be mapped into addresses of the physical address space
    * CPU can communicate with an I/O device by issuing memory instructions such as mov, and, and or
  – Modern hardware devices are suited to mapped I/O; it is faster and can be combined with DMA
• I/O-mapped I/O
  – Uses I/O address register and I/O buffer register to communicate with the I/O devices
  – CPU can read/write on an I/O port using special instructions called in, ins, out, and outs (I/O-mapped I/O)
    * CPU selects the required I/O port and transfers the data between a CPU register and the port

Multiprogramming

• CPU-bound system
• I/O-bound system
• Maintain more than one independent program in the main memory
• Sharing of time and space
• Multiprogramming OS
  – Requires addition of new hardware components
    * DMA Hardware
    * Priority Interrupt Mechanism
    * Timer
    * Storage and Instruction Protection
    * Dynamic Address Relocation
  – Complexity of operating system
  – Must hide the sharing of resources between different users
  – Must hide details of storage and I/O devices
  – Complex file system for secondary storage
• Tasks of a multiprogramming OS
  – Bridge the gap between the machine and the user level
  – Manage the available resources needed by different users
  – Enforce protection policies
  – Provide facilities for synchronization and communication

Multitasking

• Pre-emptive multitasking
  – Used in Linux
  – System preempts a process to gain control of the CPU
  – After gaining control of CPU, it selects and schedules another process to run
• Cooperative multitasking
  – Used in older Windows
  – Process stays on CPU until it decides to give it up

Multiuser systems

• Able to concurrently and independently execute several applications, possibly belonging to two or more users
• Concurrent applications
  – Active at the same time
  – Contend for various resources such as CPU, memory, and hard disk
• Independent applications
  – Each application can perform its task with no regard to what the other applications are doing
• Switching from one application to another slows down each of them and affects the response time
• Features of multiuser OS
  – An authentication mechanism to verify user identity
  – A buggy application should not block other applications running in the system
  – Malicious code should not be able to interfere with or spy on the activity of applications
  – Accounting to limit the resource usage for each individual
• Protection mechanisms implemented using hardware features associated with privileged OS execution mode, implemented through system calls interface
• Security in multiuser systems
  – Implemented through the mechanism of users and groups
  – Individuals are identified through a user identification (UID)
    * Helps in setting up private space on the machine for file storage and email messages
    * Allows setting up individual limits on the use of resources in a single application or across all applications
  – Selective sharing of data and resources is implemented through a group identification (GID)
    * Allows for controlled sharing of resources such as files
  – A special user in Unix, called root, is above the security mechanisms
    * Can access every file
    * Can manipulate every running process

Operating Systems as Virtual Machines

• Allows each user to perceive himself as the only user of the machine
• Fair share of available resources
• Time sharing (for CPU time)
• Abstraction
  – Availability of higher level operations as primitive operations
  – Virtual command language as the machine language of virtual machine
Introduction

- Virtual memory

But what about user interface?

- Not a concern for OS
- The only purpose of user interface is to shield the user from the system
- Unix takes the users inside the system, leading them through labyrinthine logic trails, labeling itself “unfriendly” in the process
- Is it GUI, or is it CUI?
  - Creeping featurism
    - Designing for beginners vs designing for experts
    - Getting into peoples’ head (playing dumb charades or pictionary)
  - Interface with other programs
  - Effects on complexity
  - Adaptability
  - Scalability (adding 1 user vs 100 users)
  - Who is in charge? User or computer?