OS Basics

OS Program b/w user & HW, executes user programs, gives convinient usage, ensures efficient use of HW.
OS Services user interface; prog execution; I/O operations; file-system manipulation; communication between Ps; error detection; res alloc; accounting – who uses how much of what; protection and security
System Call Interface Programming Interface to services provided by OS written in high-level lang (C or C++), below user interface. Each system call associated with a number, sys. call interface maintains a table of those numbers, calls Kernel to execute and returns status and output.
Direct Memory access Load data from/to I/O devices (e.g SSD) directly to/from main mem without involving CPU.
Interrupt request for the processor to interrupt currently executing code. The processor will suspend current activities, save its state and execute a function called **Interrupt Service Routine** which function address is accessed by **Interrupt Vector**. OS is interrupt driven.
Trap, Exception Software-generated interrupt. Caused by software error, system call, other process problems.
OS data structures OS needs Lists, stacks queues, trees, maps

Multiprocessor (MP) Systems

Generic Approach Each processor performs all (types of) tasks. OS shared among CPUs, each CPU has local private copy of OS data structures.
Asymmetric MP Each processor is assigned a special task. Master CPU runs OS, other Slave CPUs run user processes.
Symmetric MP Each processor performs all (types of) tasks. OS shared among CPUs.
Non-Uniform Memory Access (NUMA) Interconnected CPUs each with private mem. They logically share one physical mem space.
Clustered Systems Like MP, but multiple computers working together. Linked via some kind of network (e.g LAN).

OS Operations

Bootstrap Program Initializes system, loads OS kernel and starts execution at power-up.
Batch System, Multiprogramming multiple P's in mem; CPU changes P when waiting (for I/O)
Timesharing, Multitasking fast switching between P's; interactive; illusion of concurrency
Dual-Mode User mode, kernel mode. Goal: distinguish whether system is running user or kernel code with a HW provided **Mode bit**; some instructions privileged, only in kernel mode; sys. call changes mode to kernel; return from call resets it to user

OS Structures

System Calls

System Call Programming Interface to services provided by the OS written in high-level lang (C or C++), below user interface.
System Call Interface Each system call associated with a number, Sys. Call interface maintains a table of those numbers, calls OS to execute and returns status and output.
Parameter passing Either by passing parameters into registers (limited); or store them in a mem block and pass addresses in a register (unlimited length of parameter, limited amount of parameters); or push them onto stack (unlimited amount & length)
Syscall types File management, Device management, Information management, Communications, Protection

System Programs

System Program Provides convenient environment for program development and execution. Often they are user interfaces to system calls
System Program Types File management, Status information, Programming-language support, Program loading and execution, Communications, Background Services (daemons)

Application Programs

Application Programs designed to carry out a specific task other than one relating to the operation of the computer itself, typically to be used by end-users, e.g. web browsers.

Creation of processes

1. Preprocessing Reads c file, processes includes, expands macros, handles conditional compilation.
2. Compilation Produces object code (.o), i.e. sequences of bytes, loadable into any mem location
3. Linking Combines all object and library files into one executable file. Solves unresolved external references. Relocates machine addresses
Dynamic Linking Conditionally linked libraries. Loads system libraries only once.
Static Linking Necessary library functions are embedded directly in exe.
4. Loading Shell/click creates P, invokes loader, loads exe to RAM. OS allocates mem, relocates mem addresses.
5. Execution Program is a running P, CPU starts processing, upon completion returns status, releases resources, removed from mem
Executables across OSs Apps compiled on one OS are not executable on other OSs. (differing system calls, binary formats, instruction sets application binary interface). Can be solved via interpreted langs, virtual machines, use of standard API with compiler generating binaries in OS specific language (e.g. POSIX)

OS Structures

Monolithic Systems Includes everything between user prog and hardware. + fast kernel communication, + little overhead, + easy interaction between OS modules, - difficult to change, maintain, - single failure can cause system crash, - gets complex fast.
Loadable Kernel Modules Kernel can load independent modules such as device drivers when needed. Like layered Modules but more flexible as kernel can communicate directly. (Linux has this)
Layered Module Structure OS is divided into layers, each built on top of lower layers. Each layer implements service and communicates only to lower level layers.
Microkernel Systems Everything in user mode, except scheduling, virt. mem. and basic IPC in kernel mode. + easy to extend, + more reliable and secure, + easier to port, - more performance overhead of kernel and user space communication.
Hybrid Systems Combines microkernel and monolitic approach to address performance, security, usability needs. OS partially in kernel and user mode e.g. Linux

Process Management

Process Program loaded into mem, in execution
Program Passive entity stored on disk
Process states new, ready, running, waiting, terminated
Process Control Block Information about each P: P state, P number, PID, program counter, CPU registers, Mem.-management information (allocated mem for process), I/O status, CPU scheduling info (e.g. priority), Accounting info (e.g. CPU used)
Context Switch When CPU switches to another P, save PCB of prev. P, load PCB of new P.

Processes Layout

Stack temporary data: function parameters, return addresses, local variables
Heap dynamically mem allocated during execution
data Global variables
text executable code

Operations on processes

Process creation Parent P creates child (fork()) which can create own children → tree structure, every proc. has a unique P identifier (pid). Either parent and child share all, a part or no resources.
Process termination P's delete themselves w/ exit(). Parent deletes child w/ abort(). Parent waits for child to end w/ wait(). child → **zombie** if child terminates w/o or before parent wait(), child → **orphan** if parent terminates.

Interprocess communication

Shared Memory Communication is under control of user P (not OS), P's share mem. space. Issue: Producer-Consumer problem.
Message passing Communication controlled by OS, Messaging b/w P's without shared variables.
Direct Message passing A link is established b/w P's. They must name each other explicitly (send(P,message), receive(Q,message)). Only one communication link.
Indirect message passing Messages are sent and received from mailboxes (i.e. ports). May share several communication links
Ordinary Pipe Communication b/w parent and child, cannot be accessed from outside the proc. that created it. Have a read end (fd[1]) and write end (fd[0]). Exists until proc. completion.
Named Pipe, FIFO Several P's can use pipe for communication. Exist until deleted. Appears as a file.
Socket Endpoint of communication between different machines, concatination of IP and port. Data is sent via packets.

Scheduling

OS typically schedule threads and not processes

Basic concepts

Scheduling queue OS maintains a ready and wait (i.e. waiting for I/O or for child termination or for interrupt to be serviced) queue.
Short-term scheduler Selects P which should be brought from ready queue to mem and allocates CPU time.
Long-term scheduler Selects P which should be brought from job pool (possibly on disk) to ready queue
Dispatcher Component, which gives control of CPU to P. Jobs: switching context, switching to user mode, jumping to proper location in program.
Starvation A P has little to no CPU-time
Aging The waiting time of a P is taken into account.

Scheduling Criteria

CPU utilization $100 * cpu\_busy\_time/total\_time$
Throughput $number\_of\_finished\_Ps/time\_unit$
Turnaround time Amount of time to complete a particular P $waiting\_time + exec\_time + i\_o\_time$
Waiting time Amount of time a P is waiting in ready queue
Response time time it takes from when a request was submitted until first response is produced by P. (e.g. web server handling request)

Scheduling Algorithms

First Come, First Served (FCFS) Can lead to convoy effect, i.e. short P's waiting for long P's (starvation), which causes long avg. waiting time.
Shortest-Job-First (SJF) Length of CPU-burst estimated with previous CPU-bursts of P. Is optimal, bc minimizes average waiting time.
Shortest-remaining-time-first (SRFT) SJF with preemption, i.e. when P arrives with shorter burst-time than current running one, then current P is stopped and new P can run.
Priority scheduling (Prio) Each P has fixed priority number. Lower number = higher priority. Higher priority first.
Round Robin (RR) FCFS with preemption. each P can run for max fixed time q (**quantum time**). if q large ⇝ FCFS , q small ⇝ Context switching overhead.
Multilevel Queue Scheduling (MLQ) Ready queue partitioned into separate ready queues. With each queue a priority number is associated. Each queue has own scheduling algorithm (e.g. foreground tasks with RR and background tasks with FCFS). A P is permanently in a given queue. If MQL preemptive ⇝ starvation.
Multilevel Feedback Queue (MLFQ) Like MLQ, but Ps can move between various queues (e.g. Ps with short burst-time or long waiting-time go to higher-priority queues). This avoids starvation. MLFQ is the most general algorithm, since it can be configured in many ways.

Thread Scheduling

Process Contention Scope (PCS) Competition for CPU time is among Ts within the same P. The user-level T library schedules, OS not involved in scheduling.
System Contention Scope (SCS) Ts from different Ps, as well as Ts within the same P, compete for CPU time. OS is scheduling Ts. OS using one-to-one mapping model schedule Ts only using SCS.

Threads

Threads A basic unit of CPU utilization, executed within P. Shared with P: code, data, files. Private: registers, stack, PC and own copy of T-local-storage (i.e. copy of static data)
Multithreaded benefits responsiveness, resource sharing easier than with Ps, cheaper than P creation, lower overhead in context switching, scalability (even single P can take advantage)
Multiprocess benefits being isolated, are better suited for tasks that require a higher degree of separation and security.

Multicore Programming

Concurrency More than one task making progress, tasks running out-of-order or in a partial order. (Software Parallelism)
Parallelism Requires Concurrency and implies system can run more than one task simultaneously on multiple cores/nodes. (Hardware Parallelism)
Data Parallelism distribute subsets of data across multiple cores, same operation on each core
Task Parallelism Tasks across multiple cores, each task running unique operation(s).
Hybrid Parallelism Combination of task & data parallelism
Amdahl's Law Speedup from N cores compared to 1 core w/ serial portion S: < 1 / (S + (1-S)/N

Multithreading Models

User threads (UT) Ts in user space, which are not visible to kernel and managed without kernel support
Kernel Threads (KT) Supported and managed by OS kernel
Many-to-One Many UT mapped to one single KT. Not widely used, as one blocking UT blocks all UT and Ts don't run in parallel
One-to-One Each UT maps to one KT (two-level-concurrency: Ts in both user and kernel space are concurrent). Number of Ts restricted due to causing overhead in kernel
Many-to-Many Many UT multiplexed to many KT.OS creates as many KT as needed. Because of newer CPUs with many Ts, not that relevant anymore (starts to look like 1:1)
Two-level-Model Many-to-Many with one single One-to-One exception (which needs guaranteed level of service).

Implicit Threading

Thread pool Creation of a of Ts that can be assigned to tasks, creation overhead is reduced.
Implicit Threading Managing of Ts by frameworks. Programmer only has to identify tasks

Threading Issues

Semantics of fork() & exec() if T invokes exec(), it replaces whole process with all Ts. fork() sometimes duplicates process with all Ts and sometimes only calling T.
Signal handling (Signals are event based messages to a P) Problem: Where should a signal be delivered for a multithreaded P? Either same T is informed (sync), all Ts (async) or special signal T
Asynchronous cancellation T terminates immediately
Deferred cancellation T periodically checks if it should terminate (recommended)
Lightweight Process A data structure between kernel and user Ts to manage appropriate number of kernel Ts allocated to app.
Scheduler activations provide upcalls - a communication mechanism from the kernel to the app, to inform the app about events.

Synchronization

Cooperating P's: execute concurrently, may be interrupted, share data. Maintain data consistency through Sequencing and Coordination.

Race Conditions

Execution outcome dependent on order of concurrent access to shared data (ex: counter in prod-cons-prob, PID)
solutions: disabling interrupts (single core vs. multiprocessors (time-consuming), affects system clock)
preemptive kernel: P preempted in kernel mode, most common. more responsive, suitable for real-time programming
non-preemptive kernel: uncommon, P blocks CPU

Critical Section Problem

critical section is where a P accesses shared data (Structure: entry, critical, exit, remainder)
solution: mutual exclusion (no 2 P in CS at the same time), progress (selection cannot be postponed indefinitely), bounded waiting (limit on waiting → Starvation)
Peterson's Solution 2 Ps, int turn, bool flag[2] shared, acquire & lock, not guaranteed to work on modern computer architectures (requires atomic load/store, no instruction reordering)
Mutex Lock acquire() and release() lock (atomic), bool available (binary), require busy waiting (spinlock, no context switch required)
Semaphores integer variable, accessed through wait() and signal(). busy wait
- **counting** (init to nr of resources available, decr. in wait) vs. **binary** semaphore (like mutex lock).
- **implementation with suspension and waiting queues** instead of busy waiting S suspends itself, each semaphore has associated waiting queue. signal() removes one P from list and awakens it.

Monitors high level form of P sync. (ADT, internal vars only acessible within procedures)
- **condition variables** wait (suspends P) and signal (tells P to resume/condition could have changed) on condition var.
- **mutex locks**: acquire and release are procedures in monitor

Priority Inversion low-prio P holds lock needed by high-prio P. solution: **priority-inheritance protocol**: inherit higher priority until finished with resources

Bounded Buffer in Producer-Consumer-Problem

Buffer with n slots, each can hold one item
Semaphore Solution 3 S, init: mutex=1, full=0 (nr of fulls slots), empty=n.
- producer wait(empty);wait(mutex); //produce signal(mutex);signal(full);
- consumer wait(full);wait(mutex) //consume signal(mutex);signal(empty);

Dining Philosophers

Allocate several resources among several Ps in a DL- and starvation-free manner. ex: 5 ph, 5 forks, 3 states (thinking, hungry, eating). Init: 1 data set (bowl of rise), chopstick[5], initialized to 1 (available).
Simplest Solution remove one ph
Asymmetric Solution odd/even pick up chopsticks asymmetrically
Monitor Solution cond.var self[5], fun test(i) if hungry & neighbours not eating → self[i].signal()
- *pickup(i)* set i to hungry, call test(i). if not eating afterwards, self[i].wait()
- *putdown(i)* set i to thinking, test left and right neighbour

(starvation (requirement 3) still possible, can be solved by introducting time restriction)

POSIX Synchronization

pthread.h API is OS-independent (unix, macOS). provides mutex locks, named (accessible by multiple P's) and unnamed (need to be placed in shared mem) semaphores (include semaphore.h), condition variables (associated with a mutex lock).

Deadlocks

Deadlock 2 or more P's are waiting indefinitely for an event that can be caused only by one of the waiting Ps. 4 conditions: **Mut. Ex.**, **Hold and wait** P that holds min. 1 res waits for another res. **No preemption** res can only be released voluntarily by P holding it. **Circular Wait** Cycle in res-alloc graph
Resource-Allocation Graph Cycle necessary & sufficient condition (possibility if several instances)
Handling Deadlocks **1.** Ensure that sys never enters DL: DL prevention/avoidance (allow res-alloc only if no DL could happen) **2.** Allow DL, detect and recover from it. **3.** Ignore DLs (approach of most OS, user is responsible)
Deadlock Prevention : Eliminate at least 1 cond (only D4 practical to eliminate)
D1: Use shareable res (e.g. read-only files) **D2:** Only req. res if P doesn't hold other res (problem: low res util/ starvation) **D3:** (if res not available: first free all, restart if all needed res can be acquired at once) **D4:** Impose total ordering on all res-types. Requests allowed only in increasing order of enumeration.
Livelock P or T continously attempts an action that fails. (failure to succeed)

Memory Management

Main Memory

MM is the only storage the CPU can access directly. **Speed:** CPU Registers > Cache > Main Memory.

Base and limit registers smallest legal mem. address + size. Define **Logical Address Space (LAS)** of a P (set of all LA's). P can only access mem inside LAS (otherwise trap → fatal error). Only OS can access all registers/mem.

Logical Address / Virtual Address generated by CPU. visible to user. editable

Physical Address only seen by Mem. Unit. does not change. **PAS** set of all PAs correspoding to LAS

Address Binding mapping instructions and data to mem addresses. 3 schemes/stages: (in red)

Memory Management Unit MMU HW device, maps LA to PA during execution. → mapping methods (relocation register, contiguous paging)

Dynamic Loading : routine not loaded in mem until called. all routines kept on disk. no special os support needed **Static Linking**: Libraries and program code combined by loader. **Dynamic Linking** happens during execution. useful for shared libraries (standard C lib.) DLL: dynamically linked libraries.

Contiguous Memory Allocation each P is contained in a single section of mem that is contiguous to the section containing the next P. **Memory Protection:** through usage of Relocation & limit registers. degree of multiprog. limited by nr of partitions.

Dynamic Storage Allocation : OS maintains list of allocated and free partitions (**Holes**). First-fit (fastest), Best-fit (eq. to ff in storage-utilization, produces smallest leftover-hole), worst-fit (produces largest leftover hole)

Internal Fragmentation : physical mem organized into fixed-size blocks. happens if allocated mem larger than requested mem (internal to partition).

External Fragmentation : Total Mem Space for requests exists, but is not contiguous. 50% rule: 1/3 unusable. **Solutions** *Compaction*: moving data, can be expensive, only possible with dynamic address relocation (during ex. time) *Noncontinguous Allocation*: Strategy used in Paging

Paging PA can be noncontiguous, mem for P allocated wherever possible (no ex. fragmentation, but some internal → smaller pages eq. less int. frag but move overhead in page table). Physical mem is divided into fixed-size blocks called **frames**. Logical mem divided into same-sized blocks called **pages**.

Adress-Translation : page number and page offset in the per-P page table

OS keeps copy of each per-P page table + maintains frame table (for each physical frame)

Hardware Implementation : per-P page table kept in main mem. **PTBR** page-table base register (pointers) & **PTLR** page-table length register (size of page table)

Translation Lookaside Buffer TLB : associative mem. hw cache for page table. (page nr, frame)

Memory Protection protection bit (read-only, read-write) or valid-invalid bit (attached to each entry in the page table. indicates if legal (in LAS) or not)

Shared Pages : reentrant (unchanging) code shared among Ps. ex: Standard C libary

Structure of the Page Table

Hierarchical Page Table : ex: two-level page table, page the page table. (forwa[d mapping)

Swapping : moving P temporarily out of mem to a *backing store* and brought back for continued execution. (P roll out, roll in). system maintains ready queue. → transfer time too high, not used in modern OS

Swapping with Paging : pages of a P instead of whole P swapped. (page in, page out)

Virtual Memory

Abstracts main mem into an extremely large, uniform array of storage (LAS > PAS). Allows execution of partially-loaded programs. (more programs can run at the same time, increased CPU utilization & throughput, no increase in reponse/turnaround time, less I/O needed to swap processes)

Demand Paging : bring page into mem only when it is needed (when *page fault* occurs). **HW Support**: Valid-Invalid Bit (v: legal mem resident, i: not valid or not-in-mem). PPP-table, Secondary Mem with swap space, Instruction restart. **Pure demand paging**: extreme case where process starts with no pages in mem.

Copy-on-Write parent and child P initially share the same pages in mem. modifiable pages marked as CoW, copied only if page changes.

Free-Frame List Pool of zero-fill-on demand pages (frame cleared with 0's, before released to P). → no free frame? **Page Replacement**: select victim frame (requires 2 page transfers: page-out victim and page-in desired page, overhead can be reduced by using modify/dirty bit)

Page Replacement Algorithms Decide which pages are replaced, reduce page-fault rate (nr of page faults minimally decreases with more frames). **FIFO**: oldest page replaced (doesn't say anything about usage of page), suffers from **Belady's anomaly**: Adding more frames can increase the number of page faults. **Optimal algorithm**: Replace the page that will not be used for the longest period of time. not possible in practice bc it requires future knowledge. **LRU**: least recently used, replace page that has not been used the longest time.

Allocation of Frames How many frames are given to each process? **Fixed Allocation**: equal or proportional to process size. **Priority Allocation**: proportional to priority (sometimes size).

Replacement: select replacement frame from the set of all frames (**global**, no keeping track of which P replaced pages belong to) vs. own set of frames (**local**).

Thrashing P does not have enough frames to support the pages int he working set → high page-fault rate → low CPU utilization → OS increases multiprog (with global r. alg.) → more thrashing

- **Locality**: set of pages that are actively used together by a P.
- **Locality Model**: all progs will exhibit a basic patterned mem reference structure, page-faults occur only when it changes locality (of mem-reference). Thr. occurs when sum of locality sizes > total phys. mem, can be limited by using local replacement algo, or simply providing enough mem.

File Systems

mechanism for storing and accessing data and programs. **Files** contain data, **Directory structure** organizes and stores information about files.

Files

OS View: named collection of related info, logical storage unit, **User View:** smallest unit to store info in sec. storage.

Attributes (symbolic) **Name**, **Identifier**, **Type** (only if OS supports diff. types), **Location** (pointer to device + location in device), **Size** (in bytes/words/blocks), **Timestamps**, **Protection** (r, w, m)

Operations : **Create**: 1. find space 2. make entry in dir. **Open**: 1. evaluate file name 2. check access permissions (all ops ex. create& delete call open) **Write(filehandle, data to write)** / **Read(filehandle, pointer in mem to store data)** : keeps position pointer where next read/write must happen. 1 position pointer per process.

Reposition / Seek: changes position pointer, **Delete(dir, file name)**: releases allocated space, **Truncate**: erase contents of file

Structures : File types can indicate internal structure of files. (ex: ELF (executable and linkable file in Linux)), makes OS large and cumbersome

Types - File extension : extensions are used to indicate file types (ex: exe, c, xml, ...)

Access Methods Ways to retrieve/deal with information

- **Sequential**: information processed in order (read_next() or write_next()), developed for tape
- **Direct**: allow random access to any file block (file = sequence of records/blocks), developed for disk storage, read(n) or write(n) where n = block number
- **Indexing**: index contains pointers to blocks, search for a record in the file in index. (index can be kept in mem for faster access)

Directory

collection of nodes containing information about all files. (symbol tables that translate file names into file control blocks)

Operations Search, Create, Delete, List, Rename, Traverse

Structure **Single-Level** one dir for all users (grouping and naming problems), **Two-Level** separate dir for each user (UFD, efficient search, no grouping), **Tree-based** efficient search, grouping, abs/rel paths **Acyclic-graph** easy & good traversal algos, shared subdirs and files, need to guarantee no cycles, complicate searching and deletion **General Graph** allows cycles, requires garbage collection to recover unused disk space

Memory Mapped Files Map disk block (phys. mem) to page in virtual mem to directly access file on disk

Implementation

File System Structure 2 Problems: Define User View, Create Algorithms and Data Structures to map logical file system to physical secondary storage devices.

- **Disks**: rewritten in place, I/O transfers in units of blocks, direct access to any block of info.
- **NVM**: Non-volatile Memory

Layered File System **Application Programs** → **Logical File System** manages metadata and directory structure via *FCB* → **File Organization Module** tracks files and their logical blocks, includes free-space manager → **Basic File System** issues generic (read, write blocks) commands to device driver → **I/O Control** device drivers and interrupt handlers, transfer information between mem and dev

On-Storage Structures (Control Blocks) **Boot CB**: per volume, contains info how to boot OS from that volume. can be empty. **Volume CB**: per volume, contains volume details (nr of blocks, size of blocks, free-block count and pointers.) **File CB**: per file, organizes file **Directory Structure** organize files

In-Memory Structures : **Mount Table** info about mounted volumes **Dir-Structure Cache** info of recently accessed dirs **system-wide open-fil table** contains copy of FCB of each open file **per-process open-fil table** contains pointers to appropriate entries in sys-wide table **Buffers** hold FS-blocks

Directory Implementation **Linear List** easy to program, time-consuming in execution (requires linear search). optimizations: sorted list (but requires sort), binary tree. **Hash Table** linear list with hash data structure, decreases search time. beware of collisions. fixed size. optimizations: chained-overflow hash table.

Allocation Methods **Contiguous**: file occupies contiguous blocks on device. Simple (needs only block nr & length), performant. Problems: find space for file, knowing file size, external frag. Requires Compaction (Downtime). **Linked**: File = linked list of storage blocks. solves problems of cont. frag. dir contains pointer to first&last blocks. no direct access. (*File Allocation Table FAT*) **Indexed**: Each file has index block with pointers to data blocks. Random access, no ext. fragmentation, but overhead for index blocks (too big = overhead, too small = limits file size). *Linked Scheme* link several index blocks, *Multilevel Index* multiple levels of index blocks , *Combined scheme*

Free-Space Management File System maintains free-space list. Implementation: **Bit Vector/Map**: Each block represented by 1 bit, 0 = free, search for first 0 to find free space. **Linked List**: Link all free blocks together. Traversing time-consuming but seldom needed (always use first block). (**Grouping**: first free block contains adresses of another n free blocks)..

Internals

Storage Devices → Partitions → Volumes → File Systems

Mounting FS must be mounted before usage. **Mount point**: Location in dir structure where FS will be attached. Root partition gets mounted by the **boot loader** (set of blocks that contain enough code to know how to load the kernel) at boot time.

Partitions volume containing a FS. **Cooked** with FS, **Raw** w/o FS, raw sequence of bytes, **Root** contains the OS.

Virtualization

Abstract the HW of a single computer (CPU, RAM, Storage) into different execution environments.

Host underlying HW system. **Hypervisor/VM Manager** provides interface identical to host. **Guest**

History **Multicomputer Model** Each computer provides different service (+ reliable, isolation (sandboxing), - Maintenance, Scalability) **Virtualization** overcomes limitations. run multiple OS on one machine, Prototyping, support checkpointing and VM Migration.

Implementation of Hypervisors :

Type 0: HW-based solutions, support VM creation and management through *firmware* (VMM itself is encoded in firmware and loaded at boot-time)

Type 1: OS-like software or OS's, VMM runs in kernel mode (ex: Windows HyperV, ..), common in data centers, live migrations (balance performance, efficiency), snapshots, cloning.

Type 2: applications that run on standard OS (VMM is a P), limited hw feature exploitation. (ex: VirtualBox)

Emulators: allow apps written in one system architecture to run on different system arch.

Programming Environment Virtualization: no virtualization of real HW, but creation of optimized virtual system (ex: .NET, JVM)

Paravirtualization: avoids causing traps by modifying guest OS source code

Building Blocks

Exact Duplicate of HW difficult to provide (esp. with dual-mode operation (kernel/user mode))

VCPU Virtual CPU to represent the state of CPU per guest (as guest believes it to be).

Trap-and-emulate virtual user mode and virtual kernel mode (guest runs in physical user mode). privileged instruction (ex. sys calls) → trap to VMM → VMM emulates action → return control to guest (kernel mode slower → hw support)

Binary Translation For CPUs that do not cleanly differentiate privileged instructions. VCPU in user mode → run instructions natively. VCPU in kernel mode → inspect next few instructions, *Special Instructions* are translated and executed.

Hardware Support enables more stable, faster and feature rich virtualization. more CPU modes, hw support for Nested Page Tables, DMA, interrupts (ex: Intel VT-x, AMD: AMD-V)

Virtualization and OS Components

How do VMMs provide core OS-functions ?

CPU Scheduling VMM acts like multiprocessor system, schedules phys. CPU to VCPUs (using algos).

Overcommitment Guests are configured to use more CPUs/Mem than physically available.

Memory Management More Users of Mem → more pressure. Overcommitment common. VMM maintains **Nested Page Table** that translates guest page to real page table. (optimize without user knowing, own page-replacement-algos)

Storage Management need to provide boot disk & general data access. (support many guests, so partitioning not sufficient)

Type 0: store guest root disks &config as disk image in FS provided by VMM

Type 1: store as files in FS provided by host OS

Live Migration copy running guest to another system. (interesting: MAC must be movable (network). Limitations: Disks cannot be moved. (solve: make remote)

src connects to T(target) → T creates guest → send readonly pages → send read-write pages (clean) → send dirty pages (modified, repeatedly) → send VCPU's final state and start T → terminate source

Containerization

Application Containment run applications in isolated environment. create virtual layer between OS and applications. Each zone has own applications, Network Stack, Addresses, ... CPU and RAM divided between zones.

Containers Standardized Packaging for Software and its Dependencies. Multiple instances, Portable, Isolated (Security), Less Ressource Usage, Quick Startup, Fast(er) Live Migration, Consistent and Reliable Running Env.

Docker :

Image: represents full applications (store app)

Container: application service location and execution (run app)

Engineju: Creates, ships and runs Docker containers

Registry Service (Docker Hub): CLoud or server-based storage & distribution service for images

Dockerfile: Commands that build Image layer-by-layer (ex: alpine → python → ..)

Parallelization Steps

Sequential Program Sequential Steps + Programming (Application specification → executable program)

Parallel Program Parallelization + Programming.

Decomposition of apps into functional tasks / data blocks. (expose inherent concurrency). **Types: Data** exploits data parallelism, **Functional** expl. funct. parallelism. **Recursive** divide and conquer (exploits d or f). **Explorators** search problems, exp. data p. **Speculatory** exploit and employ if-statements. ex. f. dec.

Influencing Factors:

- **Application Type**: distinct steps or iterative block of computations (task parallel, data parallel)
- **Concurrency degree (max, avg)**: depending of degree of inherent parallelism. all/enough or none can be executed concurrently.
- **Granularity**: computational size of tasks / data blocks after decomposition. (fine, medium, coarse)
- **Target system**: affects costs of synchronization and communication. (*Shared-mem arch*: support fine-grain decomp. inexpensive more frequent s & c, *Distr. mem arch*: usually require coarse-grain decomp. more expensive and less frequent.)

Dependence Analysis is critical to identify how much parallelism exists & how it can be exploited.

Types of Dependencies:

- **Control**: describe control structure of f. dec. application. Impose precedence order on execution of tasks.
- **Data**: describe data transfers (d. dec. application). *flow (true) deps*: one task writes, another reads → also results in spec. order.
- **Name**: 2 tasks use same register/mem location without any data flow. (no true deps) *Anti-Deps*: one task reads, another writes. *Output deps*: both tasks write same var.

Mapping (assignment, part of scheduling) the exectuion of tasks / operations on data onto computing system.

- **Spatial assignment**: placing subtasks onto processing elements. (Where will a subtask be executed)?
- **Temporal ordering**: assign start time to subtask (When)?
- **Static/Dynamic Mapping**: offline (before exec.), online (during exec.). (How is a subtasks mapping performed)?

Programming or expressing parallelism in programming language.

Note: Performance now a programmers charge. (cannot rely on HW anymore).

Parallel Programming

How? Extend Compilers, **Extend languages**, Stack languages or New language & Compiler Set

Pros easiest, quickest, cheapest (effort, time). leverages existing compiler tech, new libs ready fast.

Cons lack of compiler support to catch errors (P creation, term., sync and communication), easy to write "bad" programs

Parallel Multithreaded Programming Multiple Ts (independent flow of control within one P with its own context (stack & register set)). shares P data and opened files. Lower overhead.

OpenMP

API for writing Multithreaded Applications: Compiler Directives, Library Routines. for C/C++ and Fortran. Standardizes SMP (symmetric multiprocessing). include "omp.h"

Pros

Cons

Components **Directives (Pragmas)**, **Runtime Lib routines**, **Env. variables**

- **Structured Blocks**: between {|. **Conditional Compilation** ifdef _OPENMP... **Continued lines** with \
- **Parallel regions (D)** fork when encountering parallel region (worker thread team/pool started). implicit join. sleep until needed again.
- **Work Sharing (D)**: OpenMP designed for parallelize loops (for-loops). pragma omp parallel for
- **RTL functions**: