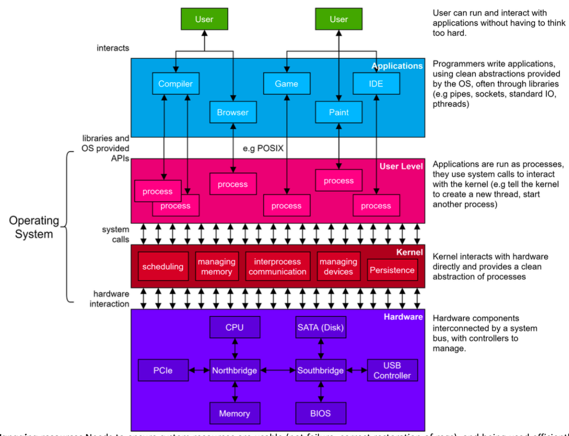
Operating Systems

Overview

Operating System Software that manages hardware and software resources as well as providing clean interfaces/abstractions for running programs to be managed and use these managed resources. OSes manage hardware and software to allow them to interact safely and efficiently so the system can operate.



Managing resources Needs to ensure system resources are usable (not failure, correct restoration of regs), and being used efficiently (switching to other tasks when one is blocked, waiting for IO etc.).
Clean interfaces Abstracts away/hides hardware complexity and variability.
Application Programming Interface (API) Interface allowing software to interact.
Portable Operating System Interface (POSIX) Supporting multiple OSes for an application is difficult because different OS has different interface. POSIX is an interface that could be used to write programs and it is implemented for many OSes. It is a unix like interface that covers OS interactions like process creation/control, pipes, signals, thread creation, control, synchronization (pthreads).
Sharing resources OSes share resources they manage (data, software, hardware) between processes/users. Resource allocation must be **fair** (processes can access system resources they need, no processes excluded from all resources/can monopolize), **efficient** (low overhead in determining how to manage resources, allocated resources efficiently), **simultaneous** (allow processes to run simultaneously/allocated different resources safely), **protective** (processes run logically in isolation/access resources without considering other processes, non can accidentally/maliciously corrupt/damage another).
Concurrency Where tasks can be run in overlapping timeframes, **logical/pseudo concurrency** where tasks run on single core and are scheduled/descheduled as needed, **real/physical concurrency** tasks run on multiple cores in parallel.
CPU can be interrupted at any time (non-determinism) and can switch activities at arbitrary times. Also, events can occur at unpredictable time/order. Need to provide synchronization primitives, allow processes to run in isolation without being interfered with (own address space).
Persistence
File System OS provides abstraction of file system, structure of directories/links/files stored in storage.
Access Controls OS enforces access control to determine which/when processes can run operations on files.
Failure Protection OS needs to detect data corruption, failing disks and create/manage recovery systems.
Organize Devices Storage devices (HDD, SSD, Tape) are organized into volumes/partitions/redundant arrays.
Portability OSes manage hardware; however we often want to use same libraries/programs/OSes on many different devices, not feasible to build OS where same binary can run on very different hardware/architectures, however we can make OS that needs little modification when adapting/porting to new architecture. Most of OS is written in high level language like C that can be recompiled for different hardware. Hardware specific components are separated (e.g., Hardware Abstraction Layer of Windows NT).
Kernel Mode Elevated privileged, full access to hardware. Read/write access to any memory, full access to CPUs privileged instructions/connected devices. OS kernel (implements basic OS functionality) runs in kernel mode – needs low level hardware access. Little restrictions can cause problems anywhere in memory.
User Mode Unprivileged, use interface provided by kernel (syscall). Can crash/restart, isolated from other processes, can't access privileged instructions, managed by kernel. Processes run in user-mode, considered to be safer but has reduced control over hardware as a result.
Kernel Core of OS, always in memory, implements most of core functionality (managing hardware, resource allocation, protection), runs in kernel/privileged mode with complete access to all hardware.
+- of splitting OS into user/kernel mode **+Protection** (Prevents system crashes/malfunction with no crash by only running code that needs low level access in kernel mode), **+Reliability** (Process running in user-mode can crash/restart without crashing system), **+Security** (Isolates applications as processes running in user space to restrict damage caused by malicious program), **-Performance** (Overhead associated with every transition through kernel-user mode boundary), **-Abstraction** (Programmers must use interface).
Monolithic Kernels Kernel as single executable with own address space. User mode processes use syscalls, all parts of kernel execute in one address space, most popular kernel style. **+Performance** (All kernel components are in one address space, fast communication), **+Easy to write** (Easy to write new kernel components, no abstractions), **-Complexity** (Lack of separation, very complex, cannot consider components in isolation), **-Correctness** (Very large and complex, so difficult to prove correctness/test), **-Robustness** (One part failing may lead to whole system crash, large kernel = more ways of crashing).
Microkernels Minimal kernel providing very basic functionality, with separate servers for file access, device IO, and over services running as processes in user-mode. Kernel provides interprocess communication (used to allow user process/server to interact), servers are processes and can therefore crash/restart/update/swap for different servers without having system down. **+Complexity** (Kernel remains small/simple. Tractable by single developer/student), **+Correctness** (Small/less complex so easy to test/debug, other OS components e.g., file systems/device drivers, can be tested in isolation, easy to extend), **+Robustness** (Servers can fail/crash without bringing down entire system), **-IPC Overhead** (OS separated into separate servers, so large amount of interprocess communication required).
Hybrid Kernels Combination of microkernel/monolithic kernel features, some components made into servers, some structured as servers, but run in kernel space to remove IPC overhead. **+Design Compromise** (More separation but lower IPC requirements), **-Performance** (User-level servers get lower performance).
Multiprocessor OS Efficiently manages/coordinates execution of programs on systems with multiple processors/CPU cores, enabling concurrent execution/resource utilization (Linux OS).
Server OS Specifically designed to provide services and manage resources for clients or other devices connected to network (Windows Server).
Mainframes OS Designed to support large-scale, high-performance computing on mainframe computers, typically used by organizations with high processing/data storage requirements (IBM z/OS).
Embedded OS Specialized software that manages/controls operations of specific device/system, typically with limited resources (FreeRTOS).
Mobile Devices OS Software platform that manages/controls operations of mobile devices, providing features and functionalities tailored for handheld usage (Android).
Real-Time OS (RTOS) Specialized software that manages/controls operations of system, ensuring timely and deterministic response to events and tasks (QNX).
Linux
History Bell Labs pioneers Multics but shifts to Unix, leading to its evolutions, the Unix Wars, the creation of GNU and MINIX, and the development of the Linux kernel with Tux as its mascot.
Features Monolithic kernel, interrupt handlers are primary means of interaction with devices, IO scheduler used to order disk operations, supports static in-kernel components and dynamically loadable modules, designed for portability, follow Unix philosophy, exposes many services and devices as files.
Unix Philosophy Summarized as "do one thing and do it well".
Dynamically Loadable Module Modules that can be loaded/unloaded from kernel space as part of kernel, without restarting/rebuilding kernel (often device drivers and reduce downtime when updating systems).
Windows
History Microsoft's journey includes partnering with IBM, evolving from Windows NT to Windows 11, expanding to mobile devices, introducing Windows Subsystem for Linux (WSL) for improved compatibility.
Features Hybrid kernel design, dynamic code libraries (DLLs) implement OS services modularly, executive layer contains most services, kernel layer contains thread scheduling/synchronization, traps/interrupt handlers, CPU management, HAL separates direct memory access (DMA) operations/BIOS config/CPU architecture specifics, device drivers are loaded into memory and dynamically linked.

Security

Goals prevent unauthorized access to system, permit authorized sharing of resources, data confidentiality, data integrity, system availability.
Policy vs Mechanism Security policy specifies what security is provided (what is protected, who has access, what access is permitted), security mechanisms (how to implement security policy, same mechanisms can support different policies).
People security Large no of computer crime by insiders, social engineering, people working around security measures for convenience, people with wrong security expectations.
Hardware security with physical access to computer/peripherals one can read contents of memory/disks, listen to network traffic including unencrypted passwords, alter contents of memory/disks, forge messages on network, steal machine/set it on fire. Hardware itself can contain exploitable security flaws.
Software security software bugs may allow attackers to compromise system (gain root privileges, crash application, steal data, compromise data integrity, deny access to system), attacks may exploit (buffer overflows, integer overflows, format string vulnerabilities).
Access Control Authentication – verification identity of users (principals), authorization – allow principals to perform action only when authorized.
Authentication – Verification of identity principal based on personal characteristics (e.g., fingerprints, voiceprints, retina patterns, can suffer from high equipment cost, false positives/negatives), possessions (e.g., keys, RFID cards, sensors, can suffer from impersonation, high equipment costs), knowledge (passwords, limitations include dictionary attacks, password reuse, password turnover, need to change regularly).
Password Protection One-way cryptographic hash: some Oss used to store user password in protected file (vulnerable to data theft, accidental disclosure/abuse by sysadmins), modern Oss store only encrypted versions of passwords (use one-way cryptographic has function for encryption, compare encrypted version of string entered by user A with encrypted password stored for A).
Password Encryption Encryption based on one-way hash functions (one-way function: easy to compute but computationally hard to invert, pre-image resistance: Given hash value h, it should be infeasible to find M such that H(M) = h), guessing is only feasible way to find cleartext password from encrypted password. UNIXs based on Data Encryption Standard (DES).
Rainbow Tables Given one-way function H, compute rainbow table of H(k)'s, for many popular passwords k, if H(password) leaks, compare it with all available H(k) in rainbow table, continue to improve rainbow table over time.
Password Protection: Salt Salt s: random value, often based on time, triple (userid, s, E(s, P)) stored in password file. At login, E(s, P) recomputed and compared with stored value. Use of salt prevents rainbow table attacks/reuse of dictionary attacks, duplicate passwords from being visible.
Authorization Specifies who can access, what they can access, how they can access. Policy decision: what should be the default authorization (no/all access?);
Principle of Least Privilege (PoLP) Gives user min rights required to carry out assigned task, unfortunately often more rights given by default for convenience.
Protection Domains Set of access rights defined as set of objects, operations permitted on them, principal executing in domain D has access rights specified by D.
Access Control Matrix Specifies authorization policy (rows represent principals e.g., users, user groups, …, columns represent target objects, e.g., files, devices, processes, …). **Implementation** Expensive to implement matrix as global 2D array, two options: access-control lists (ACLs)/ capabilities. Each column on access matrix stored as access control list (ACL), an ACL stores with each object, the principals that can access it, the operations each principal can perform on it.
Process Execution If A executes a program, the program runs with A's privileges and can access any file that A has access to. Only root can access password file.
SETUID programs SUID (set user id) bit – file switches effective UID to file owner when executed, increases privileges when using system programs; ***Process IDs*** Each process has 3 ids: real UID (id of user who started process), effective UID (effective ID of process which is used in access control checks), saved UID (saved ID to which the effective ID can be changed to), when a process starts effective UID = real UID, if setuid file, effective UID = ID of file owner, processes with elevated privileges may temporarily drop privileges changed their UID to an unprivileged value (EUID can be saved as saved UID), Non root processes can change EUID to real UID/saved UID.
Capabilities Row of access matrix can be associated with domain to give capability list, capability = possession of capability gives right to perform operations specified by it (similar to possession of key), Capabilities are protected objects (protected pointer to object specifying permitted operations on object, often not directly accessible by users but maintained by OS, alternatively give encrypted capability to user).
ACLs vs Capabilities Principle of least privilege (Capability), Revocation (ACLs), Rights transfer (Capabilities), Persistence (ACLs).
DAC vs MAC Discretionary Access Control (Principals determine who may access their objects), Mandatory Access Control (Precise system rules that determine access to objects)
Bell-La Padula Model Objects and principals have assigned security level (e.g., unclassified, confidential, top secret), two rules (read down, write up): Simple security property: Process running at security level k can read only objects at its level or lower, * property: Process running at security level k can write only objects at its level or higher. No info can leak from higher level to lower one.
Biba Model – Guarantees data integrity (read up, write down) – simple integrity principle: process running at security level k can write only objects at its level or lower (no write up), integrity * property: process running at security level k can read only objects at its level/higher (no read down).
Design Principles for Security Give each process least privilege possible (default no access), protection mechanism should be simple/uniform, scheme should be psychologically acceptable, system design should be public (security through obscurity is usually bad idea).

Processes

Process Abstraction
Process A running program. Processes contain state such as: process' own memory (address space) used for heap, stack, static data (such as instructions for program), registers (general purpose for operating on data, and special for PC, stack/frame pointer…), OS provided resources (e.g., process' currently open files). Run in user-mode, and use kernel provided interface for requesting resources, interprocess communication, and management.
Daemons processes that run in the background.
Positives: **Isolation** (Processes isolated from each other, can only affect each other through requests to kernel), **Safety** (Isolated and can fail in isolation without bringing down entire system).
Time Slicing CPU time is split into slices (quanta). At each interval, kernel's process scheduler determines which process gets next slice.
Context Switch CPU switches from executing one process, to another (context being memory map, registers, other process state). **-Overhead** Process state must be saved/restored. Could mean simply restoring registers/loading pages to/from swap. **-Caches** Some need to be invalidated and others will initially miss.
Process Control Block (PCB) Data structure storing info on a process and its state (process id, pointer to its memory map/page table, saved state, allocated resources).
Processes usually created on system initialization or by request of other processes and ends in one of 4 ways (normal completion, abnormal exit (error), aborted, never).
Process hierarchy Needed to control which processes can terminate/control other processes.
Unix processes
/* Creates copy of parent/calling process. Returns: -1 (parent process, error occurred), 0 (child process), pid (parent, pid of child) */ int fork (void);
Fork could fail if max no of processes/threads already reached, low on memory so kernel cannot allocate necessary structures, syscall could not be completed (e.g., killed).
/*Replaces program being run in current process by new program with new stack/heap. Params: path (full pathname of program to run), argv(arguments to pass to main), envp (environmental variable to pass (e.g., $PATH, $HOME). Returns: 0 (success), -1 (failure, error))*/ int execve (const char *path, char *const argv[], char *const envp[]);
Has many useful wrappers (execl, execle, execvp, execv) can be combined with fork to create child processes to run any program.
/*Terminates this process, returning status as exit code. Never returns process is killed. */ void exit(int status);
/*Sends any signal to process/process group. Params: pid: >0 (send signal to process pid), =0 (send to every process in current process' process group), =-1 (send to all processes the process is allowed to send signals to), <-1 (send to every process in process group with id -pid). Sig: =0 (no signal sent, success/failure of call checks for existence/permission to send signals), any (use signal specified by integer), returns: 0 (success), -1 (failure/error).*/ int kill(int pid, int sig);
/*Suspend execution of calling process until process pid terminates normally or receives signal. Params: pid: =-1 (wait for any child process), =0 (wait for any child in same process group of caller), <0 (wait for any child in process group =pid), stat: pointer to location to place info received on child, options: options to pass (e.g., WNOHANG – return immediately if child status not available). Returns: pid (pid of child waited on/that terminated (status stored at *stat)), 0 (if WNOHANG set and no terminated children), -1 (failure/error). */ int waitpid (int pid, int *stat, int options);
Could combine with fork to create child process and have some control over ordering operations.
Unix Signal An inter-process communication mechanism similar to hardware interrupts. Processes can send or receive signals, on receipt of signal a specified signal handler is called, processes can send signals to processes they have permission to, kernel can send signal to any process, signals contain very little info (type of signal), all signals except SIGKILL/SIGSTOP can be handled/ignored. SIGINT (Keyboard interrupt), SIGABRT (Abort signal), SIGFPE (Floating point exception), SIGKILL (Kill process), SIGSTOP (Don't kill, but remain suspended), SIGSEGV (Segmentation fault), SIGPIPE (Broken pipe), SIGALRM (Alarm timer signal), SIGTERM (Termination signal), SIGCONT (Resume/continue process).
Unix Pipe One way communication method between processes. Widely used in command lines to connect stdout of one process to the stdin of another. They can buffer, when buffer full writer blocked on write, when empty reader blocked on read. Unnamed pipes are created temporarily by processes to communicate. Named/persistent pipes are special pipes stored in file system (can be used like file) that outlive processes that created them.
/*Create new pipe. Params: fd (pipe places fds here. Fd[0] read, fd[1] write). Returns: 0 (success), -1 (failure/error)*/ int pipe (int fd[2]);

Threads

Thread Stream of instructions being executed. State defined by CPU (registers, program counter), each thread has own stack (for local variables, frames from function calls), uses address space that may be shared with many other threads (all use same heap). Process has at least one thread, can add more threads for using concurrency in program. **+Shared data** (threads in process share same address space, so can easily read/write from same memory to communicate), **+Cheap Switching** (switching between threads only need to change per-thread state (registers, stack pointer) and many caches remain unchanged (e.g., TLB)), **+Cheap Management** (Very basic, little state so creating/destroying is much cheaper than creating/destroying processes), **-Concurrency bugs** (Memory shared between threads, hence need to consider and manage/synchronize memory access), **-Fork issues** (copies calling thread into new process only, if another thread holds a lock, this lock can never be released in new process), **-Blocking** (If one thread blocks process then all threads in process are blocked).
POSIX Threads/PThreads Part of POSIX standards defining an interface for managing (creating, destroying, joining, …) threads. For pthreads functions #include <pthread.h>, For pthread types/structs #include <sys/types.h>
/*Create new thread. Params: thread (pointer to location to store new thread), attr (attributes for each stack, guard size, joinable (NULL for default)), start_routine (function to start new thread on), arg (arg to pass to start_routine). Returns 0 (success), EAGAIN (thread number limit/lack of resources fail thread creation), EINVAL (attr is invalid) , EPERM (caller lacks proper permission to create thread)*/ int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
/* Exit a thread (not the process) and provide value_ptr to any joins onto this thread. */ void pthread_exit(void *value_ptr);
Called implicitly at the end/return of the thread's start routine, If process is exited (e./g at end of main function, or any call to exit) then process terminates, terminating all threads regardless of pthread exit, If pthread exit is called by the main thread (e.g in main function) the process continues executing until exit is called, or the last running thread calls pthread exit.
/* Yield CPU to another thread. Returns: 0 -> Success!, n -> an error code */ int pthread_yield(void); If thread no longer needs CPU yield to another thread.
/* Block the current thread until thread terminates. Parameters: thread -> thread to wait on, * value_ptr -> location to put value of thread's termination (pthread_exit(void *value_ptr)). Returns: 0 -> Success!, EDEADLK -> Deadlock detected (thread joined with current thread already), EINVAL -> Thread is not joinable, EINVAL -> Another thread is already waiting to join, ESRCH -> Could not find the thread */ int pthread_join(pthread_t thread, void **value_ptr);
User level threads Kernel is unaware of threads, only interacts with processes, process maintains thread table used for thread scheduling, threads (scheduling, management) implemented by library. **+Performance (**Creation, termination, switching, synchronization are fast no expensive kernel involvement**), +Customization (**Each application can be written with own thread scheduler, best for that particular program**), -Blocking (**Any blocking syscalls on any thread block all threads in process**), -Page Faults (**OS uses page faults in memory management, during interrupt all threads blocked even though some may be runnable**), -Pre-emptive (**Implementing pre-emptive scheduling in user-mode is difficult (need to use clock interrupts – not always available, potentially needed for individual threads)**).** ***Kernel Level Threads*** Kernel can see and manage (block/schedule) individual threads within processes. **+Blocking (**Individual threads in process can be individually blocked**), +Simpler (**Scheduler easier to write though must be general**), -Performance (**Require kernel interaction for any thread related task (e.g., creation/termination/switching – mitigated by applications reusing threads (thread pools))**), -Customization (**No application specific schedulers**).** ***Hybrid Approach*** Take advantage of thread specific blocking but also be able to use cheaper synchronization, switching, creation/destruction. OS uses kernel level threads, program manages pool of kernel provided threads (scheduled by kernel), program schedules routines to run on kernel level threads. Programmer can use user-level synchronization (faster).

Scheduling

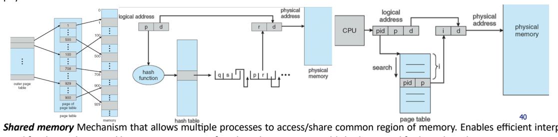
Pre-emption Where kernel interrupts/suspends a task before it completes/yields. Done through interrupts, when interrupt received CPU switches to executing routine specified as interrupt handler, on timer interrupt handler saves process/thread state, runs scheduler and potentially performs context switch to another scheduled process. ***Goals*** Fairness (Comparable process should get comparable CPU time), all run (avoid indefinitely postponing process), max utilization (do not leave CPU idle), min overhead (overhead from context switches, running scheduler), correctness (ensure scheduler correctly implemented).
Fair Share Scheduling Each user has ready queue used by scheduler. The scheduler round robins through users, scheduling processes from their ready queues, in order to fairly distribute CPU time.
Batch systems Optimize for throughput/turnaround time, ***interactive systems*** response time is critical, ***real-time systems*** run jobs to meet deadlines.
None Pre-Emptive Scheduler Processes run until voluntarily yield control to OS, must trust software, bad for interaction, good for batch systems.
Pre-Emptive Scheduler Requires timer interrupts, on timer interrupt kernel takes control and switches to another process. Good for interactive processes.
CPU Bound Process Processes that spend majority of time using CPU. Time to run limited by CPU performance. ***IO Bound Process*** Spend majority of time waiting for IO, briefly use CPU to issue IO requests, time to run limited by length of IO wait.
First Come First Served (FCFS) No pre-emption, assumes all processes scheduled eventually terminate/block (to allow other processes to run). **+All Run (**All processes eventually scheduled**), +Correctness (**Very simple/easy to implement**), +Minimize overhead (**Simple queue has low overhead**), -Turnaround time (**Small process can be kept waiting**).**
Round Robin (RR) Timer interrupt occurs periodically at each timer interrupt next process run and current put back in ready queue. **+Fairness (**All ready jobs get equal share of CPU time**), +Response Time (**Low response time for small no of jobs and small time slice**), +Turnaround time (**Low when run times differ**), +Correctness (**Just as with FCFS**), -Turnaround Time (**High when processes same/similar size, avg turnaround time large**), -Context Switches (smaller time slice/quantum mean more context switches occur, increases overhead).**
Shortest Job First (SJF) Schedule ready thread with shortest time remaining (assumes we know how long process will take, non-preemptive). **+Turnaround time (**optimally low quickest jobs run first**), +Correctness (**Simple algorithm**), +Context Switches (**Non preemptive**), -Response Time (**Non-preemptuve so response time can be high**), -Indefinite Postponement (**Many short jobs continually added could mean a longer job is never run**).**
Shortest Remaining Time (SRT) Pre-emptive version of shortest jobs first. **+Turnaround time** (low as job of shortest remaining time is always run), **-Indefinite Postponement (**If many short jobs continually added, longer job may never get scheduled**).**
Time Estimation Runtimes are rarely known in advance can use heuristics based on previous history (not always available), can use user provided estimates (could be inaccurate/malicious).
General Purpose Scheduling Favour short IO bound jobs, OS determines nature of jobs and adapts to changes. Good resource utilization, short response times, jobs often IO and CPU bound at different times.
Priority Scheduling Schedule jobs based on priority, always run job with highest priority, priorities can be externally defined, static or dynamic.
Multilevel Feedback Queues (MLFQs) Each priority level has a queue, highest priority process/thread is one being run (pre-empts if another process' priority higher), priorities recomputed periodically, can determine nature of current job, each priority level can use own system to determine next scheduled process/thread (usually each uses RR). **+Reactive (**Scheduler can react to changing behaviour of processes/threads**), +All Run (**Can use mechanisms such as aging , to ensure no process is indefinitely postponed**), +Fairness (**Mechanisms such as aging, and using recent CPU time as part of scheduler ensures all processes get some estimation of fair time**), -Inflexible** (Applications have little control/priorities no guarantess), **-Warm-Up** (Often need warm-up period), **-Cheating (e.g., if IO jobs get higher priority could add useless IO to boost priority), -No Donation (**Cannot donate priority to each other**).**
Lottery Scheduling Each job receives certain no of tickets, at each scheduling decision, one ticket chosen at random and holding process scheduled, share of tickets = share of CPU time, tickets can be transferred between jobs, can be used for resources other than CPU time. **+Meaningful (**Proportion of tickets held by job is equivalent to proportion of CPU time allocated**), +All Run (**No job can be starved, all jobs holding tickets must be scheduled**), +Donation (**Can donate tickets to unblock job**), -Proportions (**Meaningfulness of tickets relies on proportions, adding new tickets affects all jobs**), -Unpredictable (**Response time unpredictable due to randomness of scheduler**).**

Synchronization

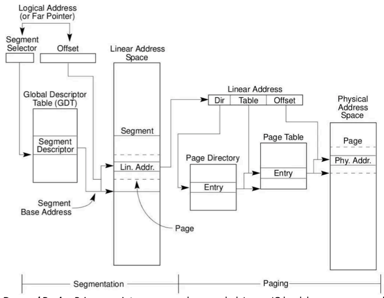
Advantages of concurrent programming Parallelism (Independent operations can be done in parallel reducing time taken), blocking (avoid blocking operations preventing progress we can separate out different IO and non-IO operations into separate threads/processes to allow progress even when some IO blocks).
Race Condition Type of bug when result of some program is dependent on some non-deterministic ordering of operations. Usually, this refers to order of access to some data by different threads in process (all share same address space), multiple processes can share data through shared memory.
Critical Section A section of code accessing some shared resource/data that requires mutual exclusion.
Mutual Exclusion Only one thread can access some shared resource/data (or being in critical section) at once. Requirements in the context of a critical section: only one thread can be in critical section at once, no thread from outside critical section can prevent other threads from entering it, no thread requiring access to critical section is delayed indefinitely, no assumptions are made about when threads will be scheduled/relative execution speed of threads.
Busy Waiting Where a program continually checks on some condition to wait rather than blocking/waiting to be awoken. Wastes CPU time (constantly rechecking) but avoids any overhead from context switching to and from, should be used when expected wait is small.
Disabling interrupts cli (Clear interrupt flag disables interrupts by clearing IF flag. Allows maskable interrupts to be ignored), sti (set interrupt flag, enable interrupts by setting IF flag, non-maskable interrupts are enabled after the next instruction). **+Simple (Already implemented by architecture), -Dangerous (**Only run in privileged mode on CPU**), -Pre-emption (**Keeping interrupts disabled for long time reduces ability of scheduler to pre-empt**), -Multi core (**Only disables interrupts on single core**).**
Strict Alternation Only one thread's turn at a time, can use any no of threads, but each must set turn to next thread to run, busy waiting used to wait until it's a thread's turn, thread outside critical section may block other threads from progressing to critical section. Must alternate so thread cannot return critical section until the other threads have and have set next turn.
Peterson's solution Allow threads to register interest, if other thread does not have turn/isn't interested in running then current thread can run. Does not require alternation, same thread can enter critical region multiple times without waiting for other, threads outside critical region cannot block others from entering (not interested), enforces mutual exclusion.
Atomic operation Single assembly instruction cannot be interrupted.
Semaphores Consists of counter representing no of threads that can enter and list of blocked threads waiting to enter that synchronize access to some critical section.
Mutual Exclusion Semaphore initialized with 1 to enforce mutual exclusion. ***Ordering*** Semaphored initialized with value 0 to order operations.
Locks enforce mutual exclusion. At most only one thread can hold lock at any given time, threads can attempt to acquire lock, if it is currently already held by another thread, the acquiring thread is blocked, only thread holding lock can release it, and on release one waiting thread is woken up and acquires the lock, a lock is much like semaphore initialized with value 1, except only holding thread can unlock.
Spin locks Spin locks are locks that busy wait (spin in while loop), does not require kernel involvement – does not need to use syscalls to block/unblock threads, need to ensure check on acquiring locks is atomic, can run into priority inversion problem. If expected wait is low busy waiting may be preferable than blocking threads.
Priority Inversion When using priority-based scheduling algorithm, it is possible for low-priority threads to block higher priority threads from running. Low priority thread holds some resource, high priority thread attempts to acquire the resource and is blocked/busy waits. Due to low priority compared with other threads, low priority thread is not run and cannot release resource (could use priority donation).
Read/Write Locks Race conditions can only occur when multiple threads access some shared data, and at least one thread is writing. Allows multiple readers to hold a lock, or a single writer. Reduced lock contention as when the read lock is held, any readers can acquire without blocking. Some read/write lock implementations allow a read lock to be upgraded to a write lock, or a write lock to be downgraded to read without requiring the lock to be released and reacquired.
Producer-Consumer Producer constraints: Can only deposit if there is space in buffer, can only deposit in buffer if mutual exclusion is ensured. Consumer constraints: can only fetch if buffer not empty, can only deposit in buffer if mutual exclusion is ensured. Buffer constraints: buffer has limited capacity (0 → N spaces).
Monitor A synchronization mechanism which enables threads to wait on condition, as well as signal to other waiting threads that a condition has been met. Threads outside monitor can call entry procedures and cannot access internal data without going through entry procedures, internal procedures can only be called from within the monitor lock, an implicit monitor lock ensures mutual exclusion within the monitor, usually implemented as a language construct.
Condition Variables A condition variable is a flag that represents some high-level condition. There are three main operations on condition variables. /*Release monitor lock and wait for c to be signalled.*/ void wait(condition_variable *c); /*Wake up one thread waiting for c.*/ void signal(condition_variable *c); /*Wake up all threads waiting for c.*/ void broadcast(condition_variable *c); Signals do not accumulate, if no processes are waiting on a signalled condition variable, the signal is discarded.
Hoare Implementation A thread waiting for signal is immediately scheduled. **+Simple (**Can easily reason about it as we are guaranteed the signalled condition's waiting thread is immediately run**), -Inefficient (**Thread that signals is switched out, despite not being finished with monitor**), -Scheduler (**Adds more constraints to scheduler**).**
Lampson Implementation Schedule signal and waking up from wait is not atomic. When condition variable is signalled, a waiting thread may be ready, but must wait until scheduled to run (at which point condition may no longer be true). **+Efficient (**Signalling thread is not switched away and no extra constraints on scheduler**), +Fault Tolerant (**If signal is erroneous, as condition is rechecked, it can be discarded**), -Complex (**Need to take extra care when waking from wait (recheck condition)**).**
Lock Overhead A measure of extra resources required for using locks (memory allocated for lock structure, time to initialize/destroy, time to acquire/release); ***Lock Contention*** Measure of number of threads waiting on lock (more contention = more threads blocked = less concurrency), ***Lock Granularity*** Measure of amount of data protected by lock.
Coarse grained Few locks with large amounts of data protected by each lock **+Simpler, +Lower Lock overhead, -higher contention, -less parallelism. Fine grained opposite.**
Deadlocks Set of processes/threads are deadlocked if each is waiting for event that only another can cause. Resource deadlock is most common. Can occur with single threads/processes (trying to reacquire re-entrant lock). Coffman conditions: Mutual exclusion (each resource is either available/assigned to 1 thread), hold and wait (thread can request new resources while holding other resources), no pre-emption (thread cannot have resources taken away), circular wait (closed chain of threads waiting on each other).

Memory Management

Memory is key component of computer. In Von Neumann architecture, all data/code stored in same memory system (as opposed to Harvard architecture where separate memories used for data/instructions).
Memory management needs to provide memory allocation/memory protection.
Requirements No knowledge how memory addresses generated (don't care about whether it is for instruction, indexing something, …), No knowledge of what memory address will be used for.
Memory Hierarchy Registers (small/fast), main memory (big/slower), disk (bigger/slower), cache (between registers/main memory has levels 1, 2, 3).
Logical Address Generated by CPU, virtual representation of memory location used by process. Address space seen by process; ***Physical Address*** Address seen by memory unit, refers to physical system memory.
Logical/Physical addresses Same in compile-/load-time address-binding schemes, different in execution-time address-binding scheme.
Memory Management Unit (MMU) Hardware device for mapping logical to physical addresses. User process deals with logical addresses only. Must be fast therefore implemented in hardware.
Contiguous Memory Allocation Main memory usually split into two partitions: resident OS (kernel) usually held in low memory with interrupt vector, user processes (user) held in high memory.
Low memory essential for system operations and ***high memory*** used for application execution.
Contiguous allocation with relocation registers **base register** contains value of smallest physical address, **limit register** contains range of logical addresses (each logical address must be less than limit register), MMU maps logical address dynamically (e.g., if process has base register = 300040, limit register = 120900, choosing 0 generates address 300040+0 = 300040, choosing 120901 is error – beyond address space).
Memory Protection Check memory access is valid to protect user processes from each other and from changing OS code/data.
Hole Available memory block; when new process arrives need to allocate memory from large enough hole.
OS maintains information about allocated partitions and free partitions (hole).
Dynamic Storage Allocation **First-fit** Allocate first hole that is big enough; **Best-fit** Allocate big enough smallest hole (must search list for smallest); **Worst-fit** Allocate largest hole (search entire list for largest).
External fragmentation Total memory exists to satisfy request but not contiguous; ***Internal fragmentation*** Allocated memory larger than requested memory, size difference internal to partition is not used.
External fragmentation reduced by compaction shuffle memory to get free blocks together, IO bottlenecks.
Swapping Number of processes limited by amount of available memory, only running processes need to be held in memory. **Solution**: Swap out processes temporarily, swap in for continued execution, requires ***swap space (swap out – memory to disk, swap in – disk to memory, holds processes that have been swapped out)*** can be file/dedicated partition on disk, transfer time is major part of swap time.
Virtual Memory – Separates user logical memory from physical memory, OS divides available memory into fixed-size ***pages*** and maps them to corresponding virtual addresses used by processes. Processes can operate as if they have access to larger amount of contiguous memory than physically available, by swapping data between RAM and disk when needed. Enables efficient management of memory resources.
Process's logical address space can be non-contiguous, allocated physical memory when available.
Frames Fixed-size blocks of physical memory (OS managed), contiguous memory to store data/instructions.
Page Fixed-size block, unit of virtual memory used by processes. Mapped to frames using page table.
Page Table Used by OS to map virtual addresses of process to corresponding physical addresses in memory.
Address translation For logical address space 2^m and page size 2^n, the first m-n bits are **page number** p, the remaining n bits are **page offset** d. p is used as index into page table, page table has base address of pages in physical memory f, f followed by d defines physical memory address sent to memory unit.
Fixed-size paging avoids external fragmentation, same size and allocated pages not always contiguous.
Memory Protection
Valid-invalid bit attached to each page table entry, **valid** = legal page (associated page in process's logical address space), **invalid** = missing page (page not in process's logical address space – ***page fault***, need to load page from disk – ***demand paging***, incorrect address – programming error?).
Page table kept in memory, Page-Table Base Register (PTBR) points to page table; ***Page-Table Length Register (PTLR)*** indicates size. Inefficient – every data/instruction requires two memory accesses.
Associative Memory Use special fast-lookup hardware **cache** as associative memory. Supports parallel search. For address translation (p, d) if p in associative register, get frame number otherwise page table.
Translation Look-Aside Buffer (TLBs) Some store Address-Space IDs (ASIDs) in entries to uniquely identify process to provide address-space protection for that process. Need to be flushed after context switch.
Effective Access Time (EAT) Associative lookup = ε, assume memory cycle time = 1μsec, **hit ratio (fraction of times page found in associative registers, ratio related to number of associative registers)** α, EAT = $(\varepsilon+1)\alpha+(\varepsilon+2)(1-\alpha) = 2 + \varepsilon - \alpha$.
Hierarchical Page Table Break up logical address space into multiple page tables. Simple technique: **two-level page table (assuming 32-bit machine with 1K page size, logical address divided into p1 (12 bits), p2 (10 bits), d (10 bits), where p1 indexes outer page table, p2 displacement within page of outer page table).**
Page Table Size increases as machine size increases, use hashed page table/inverted page table to not store entry per page but per frame.
Hashed Page Table Hash virtual page number into page table. Page table contains chain of elements hashing to same location, search for match of virtual page number in chain, extract corresponding physical frame if match found.
Inverted Page Table Each entry represents a frame in order, look through to match pid and p, the number of spaces moved down is i which is combined with d to get physical address.



Shared memory Mechanism that allows multiple processes to access/share common region of memory. Enables efficient interprocess communication by eliminating need for data to be copied between processes. After shared memory is established, no need for kernel involvement.
shmget Allocates shared memory segment; ***shmat*** Attaches shared memory segment to address space of process; ***shmctl*** Changes properties associated with shared memory segment; ***shmdt*** Detaches shared memory segment from process.

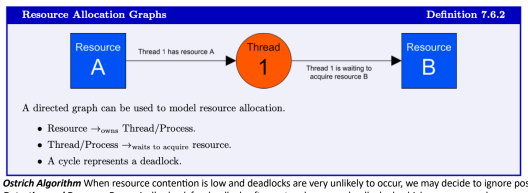


Segmentation has independent address space from 0 to some maximum, can grow/shrink independently, supports different kinds of protection, unlike pages, programmers are aware of segments. Memory allocation harder due to variable size (external fragmentation) but good for shared libraries.
Working Set Model – **Working set of pages**: W(t, w) set of pages referenced by process during process-time interval t–w to t. Add time of last use to Clock Replacement algorithm. At each page fault, examine page pointed to, if R=1, then set R=0 and move to next page, if R=0, calculate age: if age<working set age w, continue (page in WS) , if age>working set age w: if page clean, replace; otherwise, trigger write-back, continue. Processes transition between working sets: OS temporarily maintains in memory pages outside of current working set, goal of memory management to reduce misallocation. Could also observe page fault frequency if many faults, allocate more page frames.
Demand Paging Bring page into memory when needed. Lower IO load, less memory needed, faster response time, more users. Reference page needed, invalid reference (abort), not-in-memory (bring). Use valid-invalid bit (1 in memory and 0 not), initially set all entries 0, 0 during address translation page fault.
Page Faults First reference, trap to OS (page fault), OS looks at another table to decide whether it is invalid reference (abort) or valid but just not in memory (handle request), to handle valid request: get empty frame, swap page into frame, reset tables, valid bit = 1, restart last instruction.
Performance: Demand Paging Page Fault Rate 0 <= p <= 1.0, if p = 0 no page faults, if p = 1 every reference causes page fault. **Effective Access Time (EAT) = (1-p)*memory access + p(page fault overhead + [swap page out] + swap page in + restart overhead).**
Copy-On-Write (COW) Allows parent and child processes to initially share same pages in memory, if either process modifies shared page, then copy page. Efficient process creation (copy only modified pages), free pages are allocated from pool of zeroed-out pages.; ***Memory-mapped files*** Map file into virtual address space using paging, simplifies programming model for IO.
fork() Creates new child process by making exact copy of parent process image, but copying entire address space is expensive. ***Copy On Write (COW)*** Give child own page table pointing to parent's pages marked as read only, when any process writes to a page protection fault causes trap to kernel, kernel allocates new copy of page so both processes can have own private copies, both copies marked as read/write.
Page Replacement Find some unused page in memory to swap out if no free frame, strategy for page replacement: minimize number of page faults to avoid bringing same page into memory several times, prevent overallocation of memory, use modify (dirty) bit to reduce overhead of page transfers (only modified pages written to disk).
Basic Page Replacement Find location of desired page on disk, free frame ? use : select victim frame. Read desired page into (newly) freed frame, update page/frame tables, restart process.
Page Replacement Algorithms Want lowest page-fault rate, page faults decrease with more frames.
First-In-First-Out (FIFO) Algorithm Replace oldest page (may replace heavily used page).
Belady's Anomaly See frames/FIFO replacement, more frames = more page faults but goes down again.
Optimal Algorithm Replace page that won't be used for longest period of time (can't do in practice, used for measuring performance of algorithms).
Least Recently Used (LRU) Algorithms Each entry has counter, page referenced = copy clock into counter, to replace choose lowest counter. Proper LRU is expensive, use approximations instead. Reference bits initially set to 0, when referenced set to 1, replace page with r=0 if exists. Periodically reset reference bits (does not provide proper LRU order).
Second chance (or clock) page replacement Combines round robin replacement with reference bit r, if page to be replaced in RR order has r=1, set r=0 and increment RR frame index (leave page in memory), repeat subject to same rules, r=0 replace page and increment RR index.
Counting Algorithms **LFU algorithm** replace page with smallest count, may replace page just brough in, never forgets heavily used page (use aging/reset counters), **MFU algorithm** replace page with largest count.
Locality of Reference (Programs tend to request same pages in space and time) System must maintain program's favoured subset of pages in main memory otherwise **thrashing (excessive paging activity causing low processor utilization)** may occur.
Local Strategy Each process gets fixed allocation of physical memory, need to pick up changes in working set size; ***Global Strategy*** Dynamically share memory between runnable processes, initially allocate memory proportional to process size, consider page fault frequency (PFF) to tune allocation (measure page faults per sec and increase/decrease allocation).

Continuation of Linux

Ext2 inode (represents files/dirs in FS, stores info relevant to single file/dir e.g., time stamps, permissions, owner), ext2 inode pointers (first 12 directly locate 12 data blocks, 13th is indirect pointer locates block of pointers to data blocks, 14th is doubly indirect pointer locates block of indirect pointers, 15th is triply indirect pointer locates block of double indirect pointers), provides fast access to small files, while supporting very large files, ***block groups*** (clusters of contiguous blocks, FS attempts to store related data in same block group, reduces seek time for accessing groups of related data), ***superblock*** (critical data about entire FS), ***inode table*** (contains entry for each inode in block group), ***inode allocation bitmap*** (inodes used within block group), ***block allocation bitmaps*** (blocks used within group), ***group descriptor*** (block numbers for location of inode allocation bitmap, block allocation bitmap, inode table, accounting information), ***data blocks*** (remaining blocks store file/dir data – dir info stored in dir entries, each dir entry is composed of: inode no, dir, entry length, file name length, file type, file name). ***Accessing Files on UNIX/Linux*** Users are principals (each user has unique user id (uid), superuser root has UID 0 and can access any resource), files are objects, groups (each user can belong to 1+ groups, each file can only belong to one group), access rights are read (R), write (W), execute (E). For dir, it means read = can list contents of dir, write = can create/delete owned files, execute = can enter dir/get access to files.

Continuation of Synchronization



Ostrich Algorithm When resource contention is low and deadlocks are very unlikely to occur, we may decide to ignore possibility of deadlock.
Detection and Recovery Dynamically check for deadlock, after system becomes deadlocked, which resources and processes/threads are in cycle and recover. Build RAG, use DFS to look for cycles, find members of cycle and recover from deadlock by removing one (revoking resource). **Pre-emption (**Take resource from owner and give to another temporarily**), rollback (**periodically take snapshot of system's state and rollback to older checkpoint at deadlock**), killing processes (**select random process and kill it – dangerous**).**
Prevention Remove one of the four Coffman conditions: mutual exclusion – share resource (+no overhead, +more concurrency, -more race conditions), hold and wait – require all processes to specify which resources they will need (-redundant requests, -advanced knowledge), no pre-emption allow resources to be revoked (-correctness) or allow attempt to get resources, request could be revoked (-complexity), circular wait ensure resources are acquired and released in same order (+simple, +efficient, -ordering).
Livelock – Processes/threads are not blocked, but system as a whole does not make progress.

Device Management

Objectives Fair access to shared devices (allocation of dedicated devices), exploit parallelism of IO devices for multiprogramming, provide uniform simple view of IO (hide complexity of device handling, give uniform naming/error handling).
Device Independence from device type and device instance; ***Device variations***: **Unit of data transfers**: character/block; **Supported operations**: read/write/seek; **Synchronous/Asynchronous operation**; **Speed differences**; **Shareable/Single user**; **Error conditions**.
Character device device file that allows sequential, unbuffered access to data at character level, representing devices that process data character by character; ***Block device*** allows data to be read/written in fixed-size blocks, such as hard drives/SSDs.
Interrupt Handler Processes each interrupt, for block devices on transfer completion, signal device handler; for character devices when character transferred, process next character.
Device Driver Handle one device type but may control multiple devices of same type, implements block read/write, access device registers, initiate operations, schedule requests, handle errors.
Device Independent OS Layer provides device independence: mapping logical to physical devices (naming and switching), request validation against device characteristics, allocation of dedicated devices, protection/user access validation, buffering for performance and block size independence, error reporting.
Dedicated device (e.g., DVD writer, terminal, printer, …) Simple policy: open fails if already opened, alternatively, queue open requests, allocated for long periods, only allocated to authorized processes.
Shared device (e.g., disks, window terminals, …) OS provides file system for disks.
Device Allocation: Spooling uses intermediate storage to improve efficiency of IO operations with slower devices by queueing (and processing e.g., formatting/scheduling/prioritizing…) data in a spool. Increases device utilization, allows concurrent processing, support background operations, facilitates error recovery.
Buffered IO **Output**: User data transferred to OS output buffer, process continues/suspends when buffer full; **Input**: OS reads ahead, reads normally satisfied from buffer, process blocks when buffer empty. Used to smooth peaks in IO traffic/caters for differences in data transfer units between devices. ***Unbuffered IO*** Data transferred directly from user space to/from device (read/write causes physical IO, device handler used for each transfer), high process switching overhead (e.g., per character). ***User-level IO interface*** **IO operations**: open, close, read, write, seek. OS IO library procedures to set up parameters must be device independent. Synchronous/asynchronous, blocking/non-blocking, unix: Access virtual devices as files.
Memory-Mapped IO Device addressed as memory location.
Blocking IO IO call returns when operation completed, process suspended so IO appears "instantaneous", easy to understand, leads to multi-threaded code; ***Non-blocking IO*** IO call returns as much as available (e.g. read with 0 bytes), turn on for fd using fcntl syscall, provides application-level polling for IO.
Asynchronous IO Process executes in parallel with IO operation (no blocking in interface procedure), IO subsystems notifies process upon completion, supports check/wait if IO operation completed, very flexible/efficient, harder to use/potentially less secure.

Disk Management

Sector Layout Surface divided into 20+ zones, outer zones have more sectors per track, ensures sectors have same physical length, zones hidden using virtual geometry.
Disk addressing **Physical hardware address: actual geometry is complicated, hidden from OS.** Modern disks use ***logical sector addressing*** (or logical block addresses LBA) – Sectors numbered consecutively 0..n, makes disk management easier, helps work around BIOS limitations.
Disk Formatting **Low level format** Disk sector layout: preamble, data, ECC, cylinder skew, interleaving; **High level format** boot block, free block list, root directory, empty file system.
Typical Disk **Sector size** 512 bytes, **seek time (adjacent cylinder)** <1ms, **seek time (average)** 8ms, **rotation time (average latency)** 4ms, **transfer rate** upwards of 100MB/s.
Disk scheduling Minimize seek/latency times, order pending requests with respect to head position, seek time ~ 2x/3x latency.
Disk performance Seek time (t_{seek}), latency time (rotational delay) $t_{latency} = 1/2r$, transfer time $t_{transfer} = b/(rN)$ where b = number of bytes to be transferred, N = number of bytes per track, r = rotation speed in revolutions/sec. ***Total access time***: $t_{access} = t_{seek} + t_{latency} + t_{transfer}$.
First Come First Served (FCFS) No ordering of requests: random seek patterns, ok for lightly loaded disks but poor performance for heavy loads, fair scheduling.
SCAN Scheduling Choose requests which result in shortest seek time in preferred direction, only change direction when reaching outermost/innermost cylinder (or no further requests in preferred direction), most common scheduling algorithm, long delays for requests at extreme locations.
Cylinder Group of tracks located at same radial distance on each platter of HDD***; Surface*** Single side/layer of platter where data is magnetically recorded in a HDD***; Sector*** Smallest addressable unit of data storage***; Track*** Concentric circular path on platter where data is magnetically recorded***; Seek time*** Time taken for arm to move between different tracks***; Rotation time*** Time taken for specific sector/data point to rotate under head***; Latency*** Time delay between moment request made to access data and moment data is available for processing/retrieval***; Transfer time*** Time taken to read/write once head positioned.
Solid State Drive (SSD) storage device that uses integrated circuitry to store data persistently, providing faster access times and improved reliability compared to tradition HDDs. More bandwidth (1GB/s read/write vs 100MB/s), smaller latencies (microseconds vs milliseconds).
Detailed trade-offs Cost, MB/s, IOPS (Input/Output operations Per Second).
Redundant Array of Inexpensive Disks (RAID) Use array of physical drives appearing as single virtual drive, stores data distributed over array of physical disks to allow parallel operation (called striping). Use redundant disk capacity to respond to disk failure (more disks = lower mean time to failure (MTTF)).
RAID levels with different properties (performance characteristics, level of redundancy, degree of space efficiency (cost)). ***RAID level 0*** (Striping) uses multiple disks, spread out data, disks can seek/transfer data concurrently (also may balance load across disks), no redundancy so no fault tolerance. ***RAID level 1*** (Mirroring) Mirror data across disks, reads can be serviced by either disk (fast), writes update both disks in parallel (slower), failure recovery easy (high storage overhead (high cost)), ***RAID level 2 (Bit-Level Hammering)*** parallel access by striping at bit-level (use hamming ECC, corrects single-bit errors (and detect double-bit errors)), very high throughput for reads/writes (but all disks participate in IO requests (no concurrency), read-modify-write cycle), only used if high error rates expected (ecc disks become bottleneck, high storage overhead), ***RAID level 3 (Byte-Level XOR)*** Only single parity strip used (parity = data1 XOR data2 XOR data3…, reconstruct missing data from parity and remaining data), lower storage overhead than RAID L2 (but still only one IO request can take place at a time), ***RAID level 4 (Block-level XOR)*** parity strip handled on block basis (each disk operates independently), potential to service multiple reads concurrently, parity disk tends to become bottleneck (data/parity strips must be updated on each write), ***RAID level 5 (Block-Level Distributed XOR)*** distributed parity information (most commonly used), some potential for write concurrency, good storage efficiency/redundancy trade-off (reconstruction of failed disk non-trivial (and slow)).
Disk Cache Use main memory to improve disk access, buffer in main memory for disk sectors (contains copy of some sectors from disk, OS manages disk in terms of blocks – multiple sectors for efficiency), buffer uses finite space (need replacement policy when buffer full).
Least Recently Used (LRU) Replace block that was in cache longest with no references, cache consists of stack of blocks (most recently referenced block on top of stack, when block referenced (or brought into cache), place on top of stack, remove block at bottom of stack when new block brought in), don't move blocks around in main memory (use stack of pointers instead).
Least Frequently Used (LFU) Replace block that has experienced fewest references, counter associated with each block (counter incremented each time block accessed, block with smallest count selected for replacement), some blocks may be referenced many times in short period of time (leads to misleading reference count, use frequency-based replacement). ***Frequency-Based Replacement*** Divide LRU stack into two sections: new and old (block referenced is moved to top of stack, only increment reference count if not already in new), BUT blocks "age out" too quickly (use three sections – new, middle, old – and only replace blocks from old).`

File Systems

Objectives Long term, non-volatile, online storage, sharing of info or software, concurrent access to shared data, organization and management of data. ***File*** Named collection of data of arbitrary size.
File User Functions **Create**: Create empty file, allocate space and add to directory, **Delete**: Deallocate space, invalidate/remove directory entry, **Open**: Search directory for file name, check access validity and set pointers to file, **Close**: Removes pointers to file, **Read**: Access file, update current position pointers, **Write**: Access file, update pointers, **Reposition/seek**: Set current position to given value, **Truncate**: Erase contents but keep all other attributes, **Rename**: Change file name, **Read attributes**: creation date, size, archive flag, …, **Write attributes**: protection, immutable flag, …
FS Support Functions logical name to physical disk address translation, management of disk space, file locking for exclusive access, performance optimization, protection against system failure, security.
File Attributes basic info: file name (symbolic name, unique within directory), file type (text, binary, executable, directory), file organization (sequential, random), file creator (program which created file). Address info: volume (disk drive, partition), start address ((cyl, head, sect), LBA), size used, size allocated. Access control information: owner (person who controls file), authentication (password), permitted actions (read, write, delete for owner/others). Usage info: creation timestamp (date/time), last modified (could include user id), last read, last archived, expiry date (when file will be automatically deleted), access activity counts (number of reads/writes).
Space Allocation Dynamic space management (file size naturally variable, space allocated in blocks), choosing block size (too large – wastes space for small files – more memory needed for buffer space, too small – wastes space for large files – high overhead in terms of management data, high file transfer time: seek time greater than transfer time.
Contiguous File Allocation Place file data at contiguous addresses on storage device, **+Successive logical records typically physically adjacent, -External fragmentation, -Poor performance if files grow/shrink, -File grows beyond size originally specified and no contiguous free blocks available (transfer to new area, leads to additional IO operations).**
Block Linkage (Chaining) When locating data block chain must be searched from beginning, if blocks dispersed throughout disk, search process slow (block-to-block seeks occur), wastes pointer space in each block, wastes pointer space in each block, insertion/deletion by modifying pointer in previous block, large block sizes (can result in significant internal fragmentation), small block sizes (data spread across multiple blocks, dispersed through disk, poor performance due to many seeks).
Block Allocation Table Stores pointers to file blocks (directory entries indicate first block of file, block. Number as index into block allocation table – determines location of next block for centralized chaining, if current block = last block, set table entry to null), file allocation table (stored on disk but cached in memory for performance), reduces number of lengthy seeks to access given record (files become fragmented – periodic defragmentation, table can get very large).
Index Blocks each file has 1+ index blocks (contains list of pointers that point to file data blocks, file's directory entry points to its index block, chaining: may reserve last few entries in index block to store pointers to more index blocks), **+searching takes place in index blocks themselves, +place index blocks near corresponding data blocks for quick access to data, +cache index blocks in memory.**
Free Space Management Use free list (linked list of blocks containing location of free blocks, blocks are allocated from beginning of free list, newly-freed blocks appended to end of list), low overhead to perform free list maintenance operations files likely to be allocated in non-contiguous blocks – increases file access time.
Bitmap contains one bit (in memory) for each disk block (indicates whether block in use, ith bit corresponds to ith block on disk), **+Can quickly determine available contiguous blocks at certain locations on secondary storage, -May need to search entire bitmap to find free block, resulting in execution overhead.**
Filesystem Layout Fixed disk layout (with inodes) (boot block, superblock, free inode bitmap, free block (zone) bitmap, inodes+data), Superblock (contains crucial infor about FS) (no of inodes, no of data blocks, start of inode/free space bitmap, first data block, block size, max file size, …).
Filesystem Directories Directory Maps symbolic file names to logical disk locations (helps with file organization, ensures uniqueness of names).
Multilevel (Tree) Directory Structure Hierarchical file system (root indicates where on disk root directory begins, root directory points to various directories – each of which contains entries for its files, file names need to eb unique only within given directory).
Pathname file names usually given as path from root directory to file; ***Absolute and relative pathname; Directory operations*** **open/close** directory, **search** for file in directory, **create/delete** files/directories, **link** create link to file, **unlink** remove link to file, **change directory** opens new directory as current one, **list** lists/displays files in directory, **read attributes** reads attributes of file, **write attributes** of file, **mount** creates link in directory to directory in different file system.
Links Reference to dir/file in another part of FS (allows alternative names), ***hard link*** reference address of file (only supported in Unix), ***symbolic (soft) link*** Reference full pathname of file/dir (created as dir entry), **File deletion – search for links and remove them, looping: dir traversal algorithms may loop.**
Mounting Mount operation (combines multiples FSs into one namespace, allows reference from single root dir, support soft-links to files in mounted FSs but not hard links), Mount point (directory in native FS assigned to root of mounted FS), FSs manage mounted dirs with mount tables.

Linux

32-bit Virtual Memory Layout Processes map kernel to 3-4GB virtual address range (user processes can make system calls without TLB flush); kernel maps lower 896MB of physical memory to its virtual address space (All memory access must be virtual but need efficient access to user memory + DMA in low memory, create temporary mappings for >896MB of physical memory in remaining 128MB VM). ***Paging*** IA-32: 4KB page size, 4GB virtual address space; x86-64 (4-levels): 4KB, 2MB, 1GB page sizes, 48-bit addresses: 256TB. Two-level page table for 32-bit system (or 3 levels with Physical Address Extension (PAE)), also contains page status bits: dirty, read-only, … ***Page Replacement*** Uses variation of clock algorithm to approximate LRU page-replacement strategy, memory manager uses two lined lists (and reference bits), active list contains active pages (most-recently used pages near head), inactive list contains inactive pages (least recently used near tail), replace pages in inactive list; **kswapd (swap daemon)** pages in inactive list reclaimed when memory low, uses dedicated swap partition/file, must handle locked/shared pages; **pdflush kernel thread** periodically flushes dirty pages to disk. ***Loadable Kernel Module (LKM)*** provide device drivers (contain object code, loaded on-demand: dynamically linked to running kernel; require binary compatibility – modules written for different kernel versions may not work), **Kmod** Kernel subsystem managing modules without user intervention, determines module dependencies, load modules on demand. ***Two basic LKM functions***: int init_module(void) – for initialization code, void cleanup_module(void) – for clean shutdown. ***Load module***: insmod module.o (normally restricted to root). ***IO Management*** Kernel provides common interface for IO system calls, devices grouped into device classes (members of each device class perform similar functions, allows kernel to address performance needs of certain devices (or classes of devices) individually), major and minor id numbers (used by device drivers to identify devices, devices with same major num controlled by same driver, minor nums enable system to distinguish between devices of same class). ***Device Drivers*** Most devices represented by device special files, entries in /dev directory that provide access to devices, list of devices in system can be obtained by reading contents of /proc/devices. Character devices: mem, pty, tty$, cua, misc, input, lvm, pts, raw, usb; Block devices: ramdisk, fd, ide0, loop, sd, md, lvm, sd. ***Device Access*** Device files accessed via virtual file system (VFS) system calls pass to VFS, which in tuen issues calls to device drivers, most drivers implement common file operations (read, write, seek); Linux provides ioctl system call which supports special tasks: ejecting CD-ROM tray (ioctl(cdrom, CDROMEJECT, 0), retrieving status information from printer). ***Character Device IO*** Character device (transfers data as stream of bytes; represented by char_device_struct containing driver name, registered major/minor numbers, pointer to driver's file_operations structure; all registered drivers referenced by chrdevs vector), **file_operations structure** (maintains operations supported by device driver, stores functions called by VFS when system call accesses device special file). ***Block device IO*** **Block IO subsystem** Kernel's block IO subsystem contains number of layers, modularize block IO operations by placing common code in each layer. Two primary strategies used by kernel to minimize amount of time spent accessing block devices (caching data, clustering IO operations). ***Block Device Caching*** When data from block device requested, kernel first searches cache (if found, data copied to process' address space otherwise typically added to request queue), Direct IO (O_DIRECT) Driver bypasses kernel cache when accessing device, important for databases/other applications, kernel caching inappropriate/may reduce performance/consistency. ***IO classes*** Character (unstructured): Files/devices; Block (structured): Devices; Pipes (message): Interprocess communication; Socket (message): Network interface. ***Sockets*** allow bidirectional communication, can be used to exchange information both locally and across a network (unlike pipes which are identified by machine specific fds), two types of sockets (TCP – stream sockets, UDP – datagram sockets). ***IO API*** **IO calls** fd = create(filename, permission) opens file for reading/writing; fd is index to file descriptor, permission is used for access control; fd = open(filename, mode) mode is for read (O_RDONLY/0), write (O_WRONLY/1), read/write (O_RDWR/2), direct IO, close(fd) close file or device, numbytesread = read(fd, buffer, numbytes) read numbytes from file/device referenced by fd into memory buffer, returns number of bytes actually read in numbytesread, numbyteswritten = write(fd, buffer, numbytes) write numbytes to file referenced by fd from memory buffer, returns number of bytes actually written in numbyteswritten. ***IO User Interface API*** pipe(&fd[0]) creates pipe, fd is array of two integers fd[0] for read, fd[1] for write, newfd = dup(oldfd), dup2(oldfd, newfd) duplicate fd, ioctl(fd, operation, &termios) used to control devices, e.g., &termios is array of control chars, fd = mknod(filename, permission, dev) creates new special file (e.g., character/block device). ***File Descriptors*** each process has own fd table, 0 == stdin, 1 == stdout, 2 == stderr, by default these three fds refer to terminal from which program was started. ***Disk scheduling*** IO requests placed in request list (one request list for each device in system, bio structure: associates memory pages with requests), block device drivers define request operation called by kernel (kernel passes ordered request list, driver must perform all operations in list, device drivers do not define file read/write operations), some device drivers (RAID) order own requests (bypass kernel for request list ordering). ***Disk Scheduling Algorithms*** **Default**: variation of SCAN algorithm (kernel attempts to merge requests to adjacent blocks, but synchronous read requests may starve during large writes), **deadline scheduler: ensures reads performed by deadline** (eliminates read request starvation), **anticipatory scheduler: delay after read request completes** (process will issue another synchronous read operation before its quantum expires, reduces excessive seeking behavior, can lead to reduced throughput if process does not issue another read request to nearby location – anticipate process behaviour from past behaviour). ***File System Calls*** fd = open(file, how, …) open file for read/write, s = close(fd) closing an open file, n = read(fd, buffer, nbytes) read data from file to buffer, n = write(fd, buffer, nbytes) write data from buffer to file, pos = lseek(fd, offset, …) move file pointer, s = stat(name, &buf) get file's meta-data, s = fcntl(fd, cmd, …) file locking and other operations. ***File attributes can be accessed using syscall stat*** returns info about specified file in stat. ***Inodes*** (index blocks), on file open, OS opens inode table (inode entry created in memory), structured as inode on disk but includes disk device no., inode no. (for rewrite), num of processes with opened file, major/minor device no. ***Directory syscalls*** s = mkdir(path, mode) creates new directory, s = rmdir(path) remove dir, s = link(oldpath, newpath) create new hard link, s = unlink(path) unlink file, s = chdir(path) change working dir, dir = opendir(path) open dir for reading, s = closedir(dir) close dir, dirent = readdir(dir) read one entry from dir, rewinddir(dir) rewind dir to reread. ***Ext2fs*** High-performance, robust FS with support for advanced features., typical block sizes: 1024, 2048, 4096, 8192B, safety mechanism (5% blocks reserved for root – allows root processes to run when malicious process consumes all space).