Operating System Overview

Special layer of software that provides application software access to hardware resources
Referee: Manage sharing of resources, protection, isolation
Illusionist: Easy to use abstractions of physical resources
e.g. ∞ memory, dedicated machine; files, users, messages, virtualization
Glue: storage, window system, networking; sharing, authentication; look + feel
Trends in OS from hardware ↓, humans ↑; long lineage of OS

Four fundamental OS concepts

Threads
Single unique execution context with program counter, registers, execution flags, stack
Executing on processor when resident in registers
PC has addr of executing instruction, registers have context of thread (stack pointer, frame pointer, heap pointer, data)
Registers hold root state, rest in memory

Address Space with Translation
The set of accessible addresses and state associated
Programs execute in address space distinct from phys mem space
Code → static data → heap ⟷ stack
Read/write cause nothing, memory op, ignores writes, I/O, fault/exception

Process
Execution environment with restricted rights
Own address space with ≥1 threads, file descriptors, file system context
Encapsulates ≥1 threads sharing process resources
Provides protection between processes and to OS, memory protection
Tradeoff between protection + efficiency (w/in, between)
Application made up of ≥1 processes

Dual Mode Operation
Hardware provides at least two modes: "kernel"/supervisor, "user" mode
Only the "system" has access to certain resources
Protect/isolate from user programs by controlling translation of program virtual address to machine physical address
3 Types of Mode Transfer:
Syscall – process requests system service, get address + exec
Interrupt – external asynchronous event triggers context switch – Timer, I/O device, etc. independent of user process
Trap/Exception – Internal synchronous event in process – e.g. Protection violation (seg fault), divide by zero
Handle using interrupt vector + interrupt handler
Safety by: carefully pack up user process state, impossible for user to cause kernel corruption, interrupt processing not visible to user
Kernel has own stack independent from user for interrupts, syscall copies user args to kernel space before invoking function
Kernel System call handler
① Locate arguments (in registers or on user stack)
② Copy arguments (user → kernel memory), protect code from malicious code evading checks
③ Validate arguments (protect kernel from user code errors)
④ Copy results back into user memory

Concurrency
Hardware resources (CPU, DRAM, I/O), processes believe they have exclusive access to shared resources
OS must coordinate activity (multiple processes, I/O interrupts)
Use VM abstraction: simple machine abstraction for processes and multiplex the abstract machines
Results of multiprogramming
All virtual CPU's share same non-CPU resources (I/O, memory)
Each thread can access other thread data (sharing, no protection) and all threads can share instructions

Protection
OS must protect itself from buggy/malicious user programs
Reliability (compromising OS → crash), security (limit scope of process actions), privacy (limit process to permitted data), fairness
Limit translation from program addr space → physical memory addr space
Use privileged instructions, in/out instructions, special registers
Syscall processing, subsystem implementation

Interrupt Control
Interrupt handler invoked with interrupts disabled
re-enabled upon completion, non-blocking, pack up in thread to OS thread to do hard work
OS kernel may enable/disable interrupts (only), with atomic select next thread/process to run, return from interrupt/syscall
HW may have multiple levels of interrupt
Handling Interrupts Safely
Interrupt vector – limited number of kernel entry points
Kernel interrupt stack – handler works regardless of user code state
Interrupt Masking – handler is non-blocking
Atomic control transfer – "single instruction" to change PC, SP, memory protection, kernel/user mode
Transparent restartable execution – user does not know that interrupt occurred

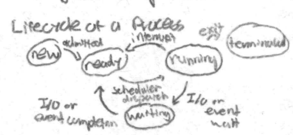
Interrupt Controller
Interrupts invoked by interrupt lines from devices
Controller chooses request to honor (mask enables/disables, priority encoder picks highest enabled, software interrupt set/cleared by software, interrupt identified specified I/O line)
CPU can disable all interrupts with internal flag,
Non-maskable interrupt line can't be disabled

The Process

Process Control Block (PCB)
Status (running, ready, blocked ...)
Register state (when not ready)
Process ID (PID), User, executable, priority
Execution time, memory space, translation
Single and Multithreaded Processes
Threads encapsulate concurrency: "active" component
Address spaces encapsulate protection: "Passive"
Allows us to switch between user processes + kernel, kernel can switch among user processes, protect OS from processes

Process Management
fork – system call to create copy of current program, start running
Return value: >0 – running in Parent, pid of child
0 – running in new child process
<0 – error in process creation, running in parent
exec – system call to replace current process image with new process image, exits process once finished
wait – class of C functions calling syscalls to wait on (external) state changes to child processes (terminated, stopped by sig)
signal – software interrupt to communicate about process, OS hardware which can change flow of program
int signal(int signum, void (*handler)(int)) – handle signal signum using function given by handler

Lifecycle of a Process



new: Process being created
ready: Process waiting to run
running: Instructions being executed
waiting: Process waiting for event
terminated: Process finished execution

Thread/Process Terminology
Thread: a sequential execution stream within a process ("lightweight process")
– Process still has single address space, no inter-thread protection
Multithreading: single program made up of many different concurrent activities
"thread" part of process (concurrency), "address space" (protection)
Heavyweight process – process with only one thread

Threads
Thread state consists of shared (content of memory – global variables, heap – I/O state → file descriptors, network connections) and private (CPU registers + PC, execution stack w/ parameters, temp var, return PCs)

Thread Operations
thread_fork(func, args) – create new thread to execute func(args) in same addr space, start executing from specified function
thread_yield() – relinquish thread voluntarily, to ready queue w/o blocking
thread_join(thread) – in parent, wait for forked thread to exit
thread_exit() – quit thread and clean up, wait for joiner

Thread Dispatcher

```
Loop {
  RunThread();
  ChooseNextThread();
  SaveStateOfCPU(curTCB);
  LoadStateOfCPU(newTCB);
}
```

Run thread by load state (PC, reg, SP) into CPU, load environment (virt mem space), jump PC
Dispatcher regains control by:
Internal: thread returns control voluntarily
External: thread gets pre-empted

Internal Events: Block on I/O, wait on signal, execute yield()
External Events: Interrupts (hw/sw signals → sup → kernel), Timer (alarm)

Thread Control Block (TCB)
Execution state: CPU registers, program counter (PC), stack pointer (SP)
Scheduling Info: state, priority, CPU time
Various Pointers (for implementing scheduling queues)
Pointer to enclosing process (PCB) – user threads
(keep track of TCB's in "kernel memory")

– Kernel vs. User-mode threads
Kernel threads natively supported by kernel, can run/block independently, one process may have several waiting but expensive (go to kernel mode to schedule)
User threads: scheduler and thread package by user program, may have several user threads/kernel thread, can be scheduled non-preemptively vs. each other + cheap but all block when block on I/O, kernel cannot adjust scheduling among threads → scheduler activations (kernel → user about block)

Some Threading schemes
① user-level library, within single-threaded process (early Java) – library → context switch, kernel time slices between processes
② (SunOS, Linux/unix variant): green threads where user-level library does thread multiplexing
③ (Windows): scheduler activations – kernel allocates processes to user-level library, thread library implement context switch, system call I/O that blocks triggers upcall
④ (Linux, MacOS, Windows): use kernel threads
System calls for thread ops, kernel does context switching, simple → lot of transitions between user, kernel mode

Thread Cooperation
Allows for ① sharing of resources, ② speedup (overlap I/O, computation thread-ahead, multiprocessor → parallel program)
③ Modularity (larger program into smaller pieces)
Independent vs. cooperating threads
Indp – no state shared, deterministic, reproducible, no order dependence
Coop – shared state, non-deterministic, non-reproducible ⇒ errors/bugs
Thread pools used to bound level of multiprogramming

Context Switch Comparison

Processes
switch overhead: high
kernel entry: low
CPU state: low
Memory / I/O: high
Process creation: high
Protection
CPU: yes
Memory/IO: yes
Sharing overhead: high (requires context switch)

Threads
switch overhead: medium
kernel entry: low
CPU state: low
Thread creation: medium
Protection
CPU: Yes
Memory/IO: No
Sharing overhead: low (thread switch overhead low)

Multi-core
switch overhead: low (only CPU state)
Thread creation: low
Protection
CPU: Yes
Memory/IO: No
Sharing overhead: low (thread switch overhead low, may not need switch)

Hyper-threading
switch overhead between hw threads very low
Contention for ALU/FPU may affect
Duplicate register state into second "thread" to allow more independent instructions

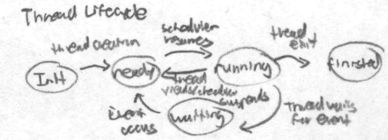
Synchronization

Threads allow us to perform overlapped/concurrent I/O and computation w/o breaking apart code, but shared state can become corrupted
Atomic Operations: always runs to completion or not at all indivisible, cannot be stopped/modified in middle
Synchronization: using atomic operations to ensure cooperation correctness between threads
Mutual Exclusion: ensure only one thread does anything at a given time
Critical section: piece of code only executed by one thread
Lock: Synchronization variable providing mutual exclusion
Before critical section/shared data, unlock after, wait if locked and sleep if waiting a long time
Tradeoff between complexity + atomicity
Implementing Locks
① Disable interrupts, then re-enable when done
user cannot perform, real-time system no timing guarantee, can miss I/O + important events, only one
② Disable interrupts during set of variable
critical section short, re-enable interrupts on next thread when sleep
cannot give to user, issues on multiprocessor
③ use atomic instruction sequences
test&set to try and set value of lock to 1, busy waits, inefficiency + priority inversion
but machine can receive interrupts, user code okay, multiprocessor
test&test&set to busy wait value (read stays in cache) and then try to grab lock
test&set w/ guard, guard is placed on lock so period of busy is short (sleep must reset guard)

Semaphore: non-negative integer value with operations
P(): atomic operation waits to become positive, then decrements (wait)
V(): increments semaphore by 1, waking waiting P (signal)
Atomic to prevent <0 and sleep P will not miss V wakeup
Use for mutual exclusion (value = 1), scheduling constraints (initial value = 0) → bounded buffer
Bounded buffer use semaphore for read/receiver + mutex
Drawback in dual-purpose, can easily deadlock

Condition Variable: queue of threads waiting inside a critical section (allows sleeping within critical)
wait(&lock): atomically release lock + sleep, re-acquire later
signal(): wake up one waiter, remove from queue
broadcast(): wake up all waiters in queue
Must hold lock when performing operations
Associated w/ lock (monitor), boolean condition, or queue of threads waiting to be true
Hoare – signaller gives lock, CPU to waiter and thread runs immediately, give lock/process to signaller when exit critical or wait again
Mesa – signaller keeps lock and processor, waiter on ready queue but possible for condition to change between thread unblock and CPU acquire

Thread Lifecycle



I/O, Sockets, Networking

Key Unix I/O Design Concepts
- Uniformity - file operations, device I/O, interprocess communication through open, read/write, close (composition)
- Open before use - opportunity for access control, arbitration, set up underlying data structures, etc
- Byte-oriented - addressing always in bytes
- Kernel-buffered reads - streaming/block devices look the same → read blocks process ⇒ yield to other
- Kernel-buffered writes - out-going transfer completion decoupled from application, allow to finish
- Explicit close

File System Abstraction
- Files live in hierarchical namespace of filenames
- File: Named collection of data in a file system
  - File data - text, binary, linearized objects
  - File metadata - size, mod time, owner, security info, access control
- Directory: "Folder" containing files and directories
  - Hierarchical (graphical) naming - path through dir graph

C File API
- Applications operate on "streams" - sequence of bytes + position
- open file w/ r, w, a and use block, character-oriented stream ops or formatted
- Stream API also permits positioning to preserve abstraction of stream, + adds buffering for performance
- Standard Streams
  - stdin, stdout, stderr opened on program execution enabling composition of programs (stdin/stdout)

C Low-level I/O
- Operations directly on file descriptors - OS object representing state
- File descriptor: index into file-descriptor table stored by kernel, create in response to open and associated with abstraction of file object

Device Drivers
- Device-specific code in kernel, interact directly with device hw
- Supports standard internal interface, same kernel I/O system interacts easily with different device drivers, config via ioctl()
- Two Parts: Top half - access in call path from system calls
  - cross-device calls like open(), close(), read, write, ioctl
  - interface to device driver (kernel), start I/O → thread to sleep
- Bottom half - run as interrupt routine
  - Gets/transfers next block of output
  - May wake sleeping threads if I/O complete

Sockets
- An abstraction of a network I/O queue
- Serve as mechanism for inter-process communication, embodies one side of communication channel
- Data transfer similar to files (read/write), any network
- Socket creation
  - Client: ① Create a socket using socket() system call
    - ② Connect socket to address of server using connect()
    - ③ Send and receive data with read()/write(), etc.
  - Server: ① Create socket with socket() system call
    - ② Bind socket to addr using bind(), to port # on host machine
    - ③ Listen for connections with listen() system call
    - ④ Accept connection with accept(), block until connect
    - ⑤ Send and receive data
  - 5 values: [client addr, client port, server addr, server port, protocol]
    - Client port "randomly" assigned, server port well-known
  - Protect self via fork, and allow concurrency w/o wait
- Namespaces
  - Hostname (www.eecs.berkeley.edu), IP address (128. ...)
  - Port Number
    - 0-1023 ("well known"), superuser privileges to bind
    - 1024-49151 "registered" ports, assigned by IANA
    - 49152-65535 "dynamic"/private, auto-allocate as "ephemeral"

Deadlock

Starvation: thread waits indefinitely (low vs high priority)

Deadlock: circular wait for resources, implies starvation but requires external intervention

Requirements: Mutual Exclusion - only one thread at time
- Hold + wait - thread holding ≥1 waits to acquire
- No preemption - resources released voluntarily only after thread finished
- circular wait - exists an ordering of circular wait

Types: Bridge crossing - two halves must be simultaneously open
- can be resolved with rollback, starvation possible
- Train wormhole - each wants to turn right, force ordering
- Dining lawyers - do not take chopstick if no one has two chopsticks afterwards, make one give up

Deadlock Detection: repeatedly try to terminate tasks

No waiting, infinite resources, no sharing of resources

Scheduling

Deciding which threads allowed to access resources to optimize desired parameters of system

Assume one program/user, one thread/program, programs indp
- programs forced to give up CPU at varying time with bursts of CPU and I/O

Goals/Criteria
- Minimize Response Time - elapsed time to complete job, user sees, real-time tasks necessary
- Maximize Throughput - operations/second, minimizing response time ⇒ more context switching
- Must minimize overhead and use resources efficiently
- Fairness
  - Share CPU among users, tradeoff between average response time and system fairness

Scheduling Schemes
- First-Come, First-Served (FCFS)
  - order of arrival, keep CPU until thread block
  - simple but short jobs get stuck (convoy effect)
  - sometimes better if cache state, context switch
- Round Robin (RR)
  - Each process gets unit of CPU time (quantum) around 10-100 ms → preempted after expires → end
  - n process, quantum q ⇒ each gets 1/n CPU time, wait no more than (n-1)q time
  - Large q → FCFS, small q → interleaved, q must be large with respect to context switch
  - Overall better for short jobs, fair but context switching adds up for long jobs
- Strict Priority Scheduling
  - Execute highest priority, RR in highest queue level
  - Leads to starvation (lower priority jobs), priority inversion
  - Fix by adjusting priority by heuristics about interactivity, locking, burst behavior
  - Implement fairness by giving each some CPU, or increase priority of un-serviced jobs
- Lottery Scheduling
  - Give job tickets of priority and randomly call
  - Assign tickets more to short to approximate SRTF, avoid starvation by giving everyone at least one
  - Adjusts well as load changes
- Knowing the Future
  - Shortest Job First (SJF) - run least amt computation job
  - Shortest Remaining Time First (SRTF) - preemptive version of SJF, compare remaining time + preempt
  - Best to minimize response time
  - All jobs same length → FCFS, varying length ✓
  - Many small jobs can lead to starvation for long jobs
  - Optimal response time, but hard to predict, unfair
    - ① Predict length of CPU burst with exponential averaging, Kalman filter → SRTF
    - ② Multi-level feedback scheduling, multiple queues with own priority/scheduling algorithm
      - Job start at highest priority, ↓ timeout expire, ↑ no timeout expire
    - Approximates SRTF, either fixed priority or timeslice between queues but user can perform useless I/O
- Real-time Scheduling
  - Performance guarantees / worst case bounds
  - Hard Realtime (EDF, least laxity first, rate-monotonic, deadline-monotonic)
  - Soft Realtime - attempt to meet deadlines with high probability, minimize miss ratio / maximize completion ratio
  - Earliest Deadline First (EDF) - consider periodic tasks with period P, computation C (give priority based on time to deadline)
  - Schedulability $\sum_{i=1}^{n}\left(\frac{C_i}{D_i}\right) \leq 1$
- Linux Completely Fair Scheduler (CFS)
  - Track virtual time by process (scale by weight / priority)
  - Targeted latency (T_L) - after which every process runs a little
  - Red-black tree for sort by vruntime
  - Increasing priority by 1 scales CPU time by same

Address Translation

Memory Multiplexing
- Controlled overlap - separate state of threads should not collide in physical memory unless shared
- Translation - processor → virtual addresses (illusion of full mem) and used to avoid overlap
- Protection - prevent access to private memory of other proces, special behavior, kernel data protected from user, prog from self
- Base and Bound leads to fragmentation problem, no support for sparse address space
- Multiple segment model leads to holes in virtual address, page fault for stack + heap to grow
  - Protection for code, shared data + stack, seg table in CPU
  - may have to swap many times

Paging
- Divide physical memory into fixed "pages" and handle translation from virtual to physical addresses
- One page table/process, virtual addr + offset
- Simple but bad if address space sparse, table big
- Two-level page table for sparseness (can page the page table)