Basics

Motivations Manage resources and coordination through process synchronisation and resource sharing, **simplify programming** through abstraction of hardware, **enforce usage policies**, **security and protection**, **user program portability**, **efficiency** through sophisticated implementations

OS Roles Abstraction, Resource Allocator, Control Program

Implementation Machine independent HLLs, machine dependent HLLs, machine dependent assembly code

Structures
Monolithic - kernel as one big program
Layered system - generalisation of monolithic system, with components organised into hierarchy
Microkernel - kernel only provides basic and essential facilities, uses IPC
Client-server - microkernel, with client process requesting service from server process, server processes built on top of microkernel

Hypervisors

Type I	Type II
Runs directly on hardware	Runs on host OS
Can directly access hardware	Negotiates with host OS
Isolated	Not isolated

Motivation for VMs Running multiple OSes, debugging & monitoring

Process Abstraction: Memory context

Text (for instructions), **Data** (for global vars, and static vars), **Heap** (for dynamic allocs), **SP** (at top of stack), **Stack** (for function invocations)

Stack frame Return address of caller, arguments (parameters) of function, storage for local variables, other information
Return address (allows for nested calls), other information could be saved registers

Stack pointer points to top of stack

Frame pointer points to a fixed location in a stack frame (platform-dependent)

Calling convention

Step 1: Preparation to make function call
Caller passes parameters with registers and/or stack
Caller saves return PC on stack

Step 2: Transfer of control to Callee
Callee saves register used by the Callee, and saves old FP and SP
Callee allocates space for local variables of caller
Callee adjusts SP to point to new stack top

Step 3: Execution of call

Step 4: Returning from call
Callee restores saved registers
Callee places return result on stack (if applicable)
Callee restores saved SP

Step 5: Transfer of control back to Caller
Caller utilises return result (if applicable)
Caller continues execution in caller

Register spilling Limited amount of GPRs → save registers in stack memory

Heap memory

Memory blocks only allocated at runtime, cannot be placed in **Data**. Memory blocks no definite deallocation timing, cannot be placed in **Stack**.

Challenges variable size of memory, variable of allocation/deallocation is variable

Process Abstraction: OS Context

Distinguish between each other PIDs

5-Stages
New Process just created
Ready Process is waiting to run
Running Process is being executed on CPU
Blocked Process is waiting for event, and cannot execute until available
Terminated Process finished execution, may require cleanup

Transitions
Create (NIL → New) Create process
Admit (New → Ready) Process ready to schedule
Switch: Scheduled (Ready → Running) Process is selected to run
Switch: Scheduled (Running → Ready) Process gives up CPU voluntarily/preempted by scheduler
Event wait (Running → Blocked) Process request event/resource/service not available/in progress
Event occurs (Blocked → Ready) Process can continue after event requested occurs

Process Control Block and Table

Challenges Scalability (amount of concurrent processes), Efficiency (minimum space wastage)

Process Interaction with OS

Unix system calls Function wrapper, or function adapter

System call mechanism
1. User program invokes library call →
2. Library call places system call number in a designated location like register →
3. Library call executes special instruction to switch from user to kernel mode (TRAP) →
4. Determine appropriate system call handler →
5. Execute system call handler to carry out request →
6. System call handler ended, control return to library call and switch back to user mode →
7. Library call returns to user program

Exception Synchronous, exception handler
Interrupt Asynchronous, interrupt handler

Unix

fork() returns 0 for child, PID for parent
Child differs in process ID, parent.

exec*() replaces current executing process image with new one, only replaces code (PID and other remains)

init() Root processes - PID = 1

exit() Status returned - 0 for normal termination. Becomes a zombie process

Zombie process
parent process terminates before child → child adopted by init and becomes **orphan** process
child process terminates before parent and parent does not call wait() → child becomes **zombie** process
zombie processes still occupy resources

Process Scheduling

Criterias Fairness (fair share of CPU time per process/user, no starvation), balance (all parts utilised)

Preemptiveness Process is given fixed time quota to run

Scheduling for Batch Processing

Criteria Turnaround time $finish - start$, Throughput (number of tasks/unit time), CPU utilisation

FCFS No starvation as number of tasks in front of task is always decreasing
Disadvantages Not optimal waiiting, convoy effect

SJF Shortest Job First - needs prediction of CPU time (exponential average)
Advantages Reduces avg. waiting time
Disadvantages Short processes can block long processes

SRF Shortest Remaining Time - **preemptive** selects job with shortest remaining (expected) time.

Scheduling for Interactive Systems

Criterias Response time, predictability

Timer interrupt Interrupt goes off periodically
Time quantum Execution duration given to process

Round-Robin Preemptive FCFS - $(n-1)q$ response time. Bigger quantum = better CPU utilisation, longer waiting time (No starvation)

Priority scheduling Preemptive - higher priority process can preempt and non preemptive - late coming has to wait for next round of scheduling
Can starve if high priority process keeps hogging
Priority inversion: Medium priority process block high priority process requiring resources blocked by low priority process
Solutions Decrease priority gradually, temporary increase in priority

MLFQ Adaptive, minimising both response and turnaround time
$p(A) > p(B) \implies A\ runs, p(A) = p(B) \implies RR$
New job given highest priority, reduce priority if TQ fully utilised time slice, retain priority if not
Disadvantages Can be exploited: Process designed to give up right before TQ, bad response time if jobs homogeneous

Lottery Scheduling Randomised - gives out lottery ticket
Responsive - allows for immediate participation in next lottery, gives good level of control, no starvation*

Threads

Unique information thread ID, registers, stack

Benefit Multiple threads require much less resources (economy), resource sharing, appear responsive, scalable w.r.t CPUs

Problems Parallel system calls possible

Thread Models

User thread User library - kernel unaware
Pros OS-independent, operations are library calls, more configurable and flexible
Cons OS not aware of threads (scheduling at process level), single thread blockage = process blocked = all blocked, prevents optimal utilisation of multiple CPUs

Kernel thread OS - kernel aware
Pros Scheduling on thread levels, allows more than 1 thread to run simultaneously
Cons Thread operations as syscall makes it slower and more resource intensive, less flexible

Hybrid OS scheduling on kernel, user thread binding to kernel threads

POSIX threads (pthread)

pthread_create (pthread_t*, ptrhead_attr_t*, void* (void *), void*) - threadId of created thread, threadAttributes, startRoutine, args for start routine

pthread_exit (void *) - pointer to exit value to return to whoever syncs (If not used, terminates when end of startRoutine reached)

pthread_join (pthread_t, void**) - threadID to wait for, exit value returned by pthread

IPC

Shared memory Share memory and communicate through region
Pros Efficient, easy to use
Cons Synchronisation, hard to implement
In POSIX: Create/locate region, attach to space, read from/write to space, detach, destroy (from one process)

Creating shared mem: shmget(IPC_PRIVATE, size, IPC_CREAT — 0600)
Attaching shared mem: shmat(id, NULL, 0)
Detaching shared mem: shmdt(addr)
Destroy shared mem: shmctl(id, IPC_RMID, 0)

Message passing Process sends message, and other process receives
Allows for naming and synchronisation
Pros Portable, easier synchronisation
Cons Inefficient, harder to use
Naming schemes: direct (name other party), indirect (from common message storage)
Synchronisation
sync (blocking) - simplify programming, ensures sync but can be inefficient, and may block indefinitely
async (non-blocking) - responsive, efficient, no deadlock/unresponsiveness, but complex if operations cannot complete immediately, and can lead to busy waiting/spinning

Unix Pipes

Input written into end of pipe, output read from other end. Implicit synchronisation (writers wait when buffer full, readers wait when buffer empty)

Variants Multiple readers/writers, half-duplex (unidirectional, one W, one R), duplex (bidirectional, both RW)

pipe(int pipeFd[2]) Takes in array of file descriptors, [readEnd, writeEnd] (0 success, !0 error)

close(pipeFd[i])

write(pipeFd[write_end], str, len(str) + 1

read(pipeFd[read_end], buf, sizeof(buf)

dup(oldfd), dup2(oldfd, newfd) creates copies of given file descriptor.
dup2() can be used for input/output redirection

Signal Kill, Stop, Continue, Memory Error, Arithmetic Error

Synchronisation

Race conditions Situation where order of events is important, but system doesn't enforce order

Possible issues Deadlock, livelock, starvation

Critical section Only one process can execute, prevent other processes from entering while executing

Properties of correct CS

Mutual exclusion (no two process simultaneously in critical section)
Progress (no process in CS, one waiting process should be granted access)
Bounded wait (no process should wait forever to enter CS)
Independence (process running outside critical section cannot block)

Implementations of CS

Assembly Level Implementations TestAndSet
Peterson's Algorithm Keep track of Turn and Want - an array to represent which process wants the process. Can cause busy waiting, not general (only addresses mutex)
Want[n] Array Independent, but deadlock
Turn Violate independent, but mutex
Disable interrupts Buggy CS can stall, permissions needed

Semaphore protected integers/list to keep track of waiting processes
$S_{init} \geqslant 0 \implies S_{cur} = S_{init} + \#signal(S) - \#wait(S)$
Conditional variables related to monitoring

Synchronisation Problems

Producer-Consumer Producers share bounded buffer, and only produces items when buffer not full, and consumers remove items when not empty.

Readers-Writers Processes share data structure where readers retrieve and writer modifies. Writers must write alone.
No reader should be kept waiting until writer has obtained permission
Once writer is ready, performs write as soon as possible.

Dining Philosophers 5 philosophers with 5 chopsticks. Philosopher thinks → tries to pick up two chopsticks closest → eats without releasing → continue
Remedies Limited Eater (allow at most 4 to sit simultaneously), Mutex (only pickup if both available), Assymmetric (odd pick up left than right, even vice-versa)

Implementations in UNIX

POSIX Semaphore Initialise semaphore, perform *wait()* or *signal()*

pthread Has additional broadcast method