Intro

If programs had to share hardware themselves, very difficult; conflicts easily; interfere with each other, etc. OS' goal is to run programs easily, effectively, securely etc. OS also handles concurrency, provides persistence, and supports networking, system security
Process = running program; Program = stored instructions; Address Space = Programs Data;
PCB (Process Control Block) is used to keep track of process info
Every process has a parent; Starts at initial ends in final (zombie) state. If Parent finishes before child, child becomes orphan and root becomes parent
pid_t fork() → creates copy of current process; continuing at execution from return of fork().
Returns pid of child to parent, 0 to child, and -1 if failed (no child created)
pid_t wait(int w_status) → waits for a child process of the current running process;
Returns pid of child that terminated, w_status gets filled with info on termination status
If process terminates without parent calling wait(), becomes zombie.
int exec(const char *pathname, char *const argv[]) → replaces the current program with a new one, command line arguments being passed in argv.
Returns: On success does not return (nowhere to return to) and runs new program. On fail returns -1
int pipe(int p[2]) → creates a communication channel, normally called before fork in order to communicate
Returns: On success returns 0, p[0] is read, p[1] is write. On failure returns -1
int dup(int fd) → changes pointer of file descriptor
Returns: On success returns new file descriptor and moves pointer. On failure returns -1

Schedulers

User mode versus Kernel Mode: User mode does not have access to read/write I/O devices, or use memeory outside its space, Kernel mode has full access; In order to use I/O as user, use system call, causes a trap;
Time-sharing → multiple processes run together on a single machine as though they are in sole control
multiprogramming - when process waiting for I/O, OS can have another process use CPU
multitasking - each process gets a time slice, a time limit before being forced off CPU
Time needed by CPU for process is job or CPU burst; time to wait is an I/O burst
preempting = swapping before process is done
$T_{turnaround} = T_{completion} - T_{arrival}; T_{response} = T_{firstrun} - T_{arrival}$
FIFO = First In First Out → easy to implement (simple queue), no preemption, not very good. Relies on arrival order
SJF = Shortest Job First → priority queue based off job length, no preemption, optimal average turnaround when arriving at same time. Relies on arrival order
STCF = Shortest Time-to-Completion First → priority queue based off job length, preempts if job appears with shorter time left. Short jobs don't have to wait for long jobs. Can still be bad and starve with long jobs / lots of short jobs
SJF and STCF require an oracle → a way to decide how long jobs take
RR = Round Robin → FIFO queue, Preemption as each job gets single time slice, Low response time, High turnaround time.
Multi Level Queue → Process that need fast response but little CPU should be highest, Process that need long CPU and little I/O lowest priority Ruleset: 1) If Priority(A) > Priority(B); A runs,
2) If Priority(A) = Priority(B); A&B run in RR using time slice of given queue,
3) When process enters it starts at highest priority → Feedback
4) Once a process has used its time slice its priority gets reduced → Confront gaming the system and adding feedback
5) After some time period S, move all processes to topmost queue → Priority Boost

Lottery → each process gets tickets, scheduler randomly picks a ticket at each time slice, that process gets the time slice. Can gift tickets out inversely to completion time, nondeterministic, no job can be starved, a short job can get unlucky. Fair in long term, stateless.
Stride Scheduling → deterministic but with same fairness as lottery. Fair in short and long term. Requires state

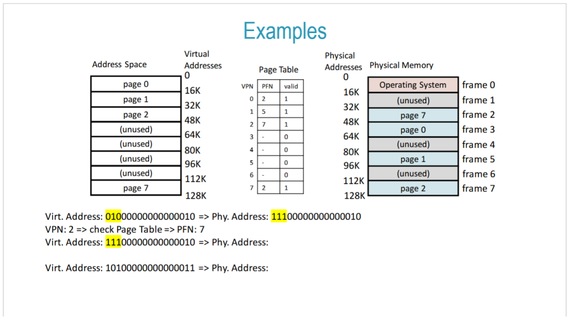
Linux → sched_latency (time period for all runnable processes once.), divided into n slices where n is the number of processes. min_granularity is the minimum size of time slice. For each slice picks the process with the smallest runtime that still needs to run. Uses priority system with niceness (high nice = lower priority). Updates time slice to worry about priority. vruntime k = vruntime k + runtime * weight 0 / weight k. uses states. new process is set to the shorted vruntime of the existing processes. small vruntime = larger weight = longer physical runtime. uses red-black tree for balancing.

Memory

Single Programming: splits memory between OS and user process. Addresses assigned at compile time (not used in modern computers) Multi Programming: Processes have to share main memory; Simple solution is to assign contiguous memory for each process. Memory actually gets fragmented and split, may even have to use disk Programmers view: Program Code @ 0 (static values as well), Heap starts at top and goes down, Stack starts at bottom and moves up. Heap = malloc, stack = declaration Memory virtualization gets from messy fragmentation to programmer view. 32 bit = 4 GB of memory. Registers, instructions, address bus and data bus = one word. Word size dependent on architecture, usually a power of 2.
How? limited direct execution but since it happens at every stage also hardware-based address translation. Loader may statically translate before execution

$$physicaladdress = virutaladdress + base$$

where *base* is start of process. *bounds* maximum address. bounds may be used on vr or pr. MMU = Memory Management Unit; hardware responsible for all memory translation. part of the CPU but sits between core and address bus. Problems with base and bounds is that if memory needs to grow, lots to update. We want sparse address space, segemenation is how we get that. parts are independent. Divided into 3 segments; Code, Heap, Stack. Last 3 bytes are offset, first 2 bits are segment bits.
allows for change in how it grows and where, etc. Can also allow for sharing of memory if read only
Can result in external fragmentation, need to compact in order to reclaim.
Page table, uses valid bit to show if that entry is valid (not all memory needs allocated), converts VPN to PFN If total address space is 2^m bytes and page size is 2^n then VPN is $m - n$ bits and offset is n bits If physical memory is 2^m bytes and (frame) page size is 2^n then PFN is $m - n$ bits and offset is n bits Thus PhysAddr = PFN × frame_size + offset

Example: 

VPN	PFN	valid
0	2	1
1	5	1
2	7	1
3	-	0
4	-	0
5	-	0
6	-	0
7	2	1

Virt. Address: 01000000000000010 => Phy. Address: 11100000000000010
VPN: 2 => check Page Table => PFN: 7
Virt. Address: 11100000000000010 => Phy. Address:

Virt. Address: 10100000000000011 => Phy. Address:

Linux page size is 8KB, Linux also uses mutli level structure, lookup is slow though
TLB = cache for page table entries. May use LRU (least recently used) or Random for misses. context switches make misses more often.
Page table includes present bit, to state if in physical memory or swap (disc). Page fault occurs when attempting to access page not in memory
Cache styles:
FIFO → easy to implement and exactly as sounds, can lead to Beladys anomly in which larger cache leads to worse response
Random → as it sounds; LRU → better than FIFO, no anamoly, slow.
Thrashing is when a process spends more time on page-fault than executing.

Concurrency