1) The OS

OS provides **a clean interface to user apps**; abstracts away the complex interface of low level hardware devices, so u-apps needn't be concerned with specific hardware details e.g. RAM type, by providing **high-level abstractions.** The **Kernel** is part of the OS **is always loaded into Memory.** It runs in a **privileged mode** –access all hardware resources.

1.1) Jobs of the OS:

1) Managing Resources: e.g. Processors/ Memory /IO Devices. Resources are limited and must be shared between users. **2) Share**: OSes must share data, programs, hardware between users simultaneously. Done with **time and space multiplexing** (slicing processor time and RAM s.t. each user has a piece). Must also offer **Resource Allocation**: Fairly and efficiently use CPU Time, disk space; **Mutual Exclusion** for shared resources; Protect against data corruption. **3) Support Concurrency:** Run multiple activities in parallel. Must be capable of **Context Switching**. **Ensure Fairness** – don't starve processes; prioritise important ones. **Offer Safe Concurrency** with **Concurrency Primitives,** Programs should have their "own space" that is preserved. **4) Store Data Well**: Easy access to files with **user-defined names** by managing directory structures, links, shared disks. **Enforce Access Controls** on read/write/remove/exec /copy perms. **Protect against unexpected failures** i.e: with backups. **Manage Storage Devices** by organizing disks into volumes, partitions, RAID. **5) Handle non-determinism**. Also provide **Simplified I/O** (disk/dvd or remote file server access – OneDrive), **Virtual Memory**, **File Systems, Program Interaction & Communication** (msgs, pipes, sockets, shared mem), **Networking, Security,** a **HCI**, **Administration and Management**.

1.2) OS Structures:

1) Monolithic Kernel: Single executable with own addr. space. Acts as a single black box. The structure is not visible but it can be interacted with by pushing parameters to the stack and trapping to kernel mode to exec sys calls. Most popular. **Advantages:** Efficient calls within kernel. Easy to write Kernel Components due to shared mem. **Disadvantages:** Complex design, many interactions. No component protection; system crashes if one component fails. **2) Microkernel:** Small kernels that have functionality in **User Level Servers** (a ULS is a program that runs on a microkernel supporting higher level functionality like networking, filesys access, GUIs). Uses Inter-Process Communication (IPC) between servers and have separate servers for device IO, file access, and process scheduling. **Advantages:** Less complex kernel, less error. Servers have clean interfaces. **Disadvantages:** High overhead from IPC within the kernel. **3) Hybrid:** Combines features from both the above. **Compromise between clean design and performance overhead**. **Advantage:** More structured design. **Disadvantage:** User-level servers are slower than that of Microkernel

Linux is Monolithic. Linux SysCalls are done by pushing args into regs or the stack, and issuing traps to switch from a user context to a kernel context. Linux supports many programs due to the GNU project – Shells, desktop environments and utility programs. **Interrupt handlers** are the main means to interact with devices. They stop curr process, save state, start driver and then return to the saved state. The IO scheduler orders disk operations. Linux supports in-kernel components and dynamically loadable modules. **Windows is Hybrid:** The Windows NT kernel consists of two layers: Executive: most services. Kernel: thread scheduling and synchronisation, traps, interrupt handlers, and CPU management. Programs are built on top of dynamic code libraries (DLLs), and DLLs implement the OS services in a modular fashion. NTOS provides system calls, loaded from ntoskrnl.exe at boot. Hardware Abstraction Layer (HAL) abstracts out DMA operations, BIOS config, CPU types. Device drivers are loaded into kernel memory and are dynamically linked against NTOS and HAL layers.

2) Processes

A process is an instance of a running program. They can be thought of as **virtual CPUs** with their own **memory space and state**. They are useful because: 1) Provide an illusion of Concurrency, 2) Provides isolation between programs, simplifying programming 3) Allow for better resource usage (diff processes use diff resources at specific times) 4) Processes can have multiple threads.

2.1) Time Slicing

The CPU Scheduler provides the ability to support many processes by **context switching** every few ms, so all processes get CPU time. Note True/Pseudo Concurrency and **Fairness. Context Switch –** the processor switches from executing one process to another, in response to **interrupts** (non deterministic event) or after a time slice. **PCBs** store the process state so that we don't lose data when context switching. **Since context switches are expensive** (must store process state and CPU caches including the TLB are lost) they should be minimized.

2.2) Process Control Block (PCB)

Processes have own virtual CPU, address space (stack & heap), open FDs. This must be preserved after context switches. The PCB stores: **1) Registers, 2) Process management info** (PID, parent process, recent_cpu, priority), **3) File management info** (root dir, working dir, open fds).

2.3) Process Creation

Two process types: **Foreground:** Process that interact with users, **Background:** Daemons (handle mail, printing requests, networking, etc). Processes are created on startup, by user requests or syscalls. **Process termination** is caused by **1) Normal Completion 2) Syscalls** `exit()/ExitProcess()` **3) Abnormally 4) Aborted** (SIGQUIT) **5) Never** (servers).

In UNIX we have **Process Hierarchies** – parent / child, etc. Windows instead gives a handle token to the parent when it spawns a new process.

2.4) Processes in UNIX

1) `int fork(void)` creates a new child process by copying the parent process and its resources. The child begins executing concurrently with its parent. Fork returns the child PID in the parent, or -1 if it failed, and 0 in the child.

2) `int execve(const char *path, char *const argv[], char *const envp[])` **execve** executes a command. Instead of copying a parent process a completely new process is made. **path** is the full pathname of the process to be executed, **argv** the list of arguments to be passed to the program. **envp** is a list of environment variables.

3) `int waitpid(int pid, int* stat, int options)` suspends execution of the calling process. If pid=-1, then we wait for any child, if pid=0, we wait for a child in the same process group as the caller, if pid = -gid we wait for any child with process group gid. The return value is the pid of the terminated child process, or 0 if WNOHANG was set in options, or -1 on error (with errno set to indicate the error).

4) `void exit(int status)` terminates the calling process, returning the exit status to the parent process (i.e: through `int* stat` in **waitpid**). `void kill(int pid, int sig)` sends signal sig to process pid.

5) Windows uses `CreateProcess` which is the equivalent of a UNIX `fork` followed by `execve`. CreateProcess is an alias of CreateProcessA which has many arguments.

2.5) Process Communication

Processes can **communicate in many ways**: e.g: Files, Signals (UNIX), Events and Exceptions (Windows), Pipes, Sockets, Shared Memory and Semaphores. In **UNIX**: Signals are an IPC Mechanism. They notify a process when an event has occurred and are delivered similarly to how interrupts are. A **process requires permissions** to send signals to another process (or must have matching user ID), but the kernel can send signals to any process. **Signals are generated** when an exception occurs, when the kernel wants to notify a process of an event, in response to certain key commands, or the kill syscall. UNIX has many signals, examples include SIGPIPE (floating point exception), SIGKILL (kill prog), SIGBUS (bus error), and SIGSEGV (segfault). The default response to most signals is to terminate the process but we could also ignore the signal or handle it with a **signal handler:**

```
#include <signal.h> #include <stdio.h>
void sigint_handler(int sig) {fprintf(stderr, " => SIGINT caught!\n"); }
int main(int argc, char *argv[]) { signal(SIGINT, sigint_handler); while (1) {}}
```

2.6) UNIX Pipes

A pipe connects the stdout of one process into the stdin of another. This grants one-way communication channel between processes. Two types of pipes: **1) Unnamed:** created, used and destroyed in the lifetime of a process. `int pipe(int fd[2])` **pipe** takes two file descriptors: `fd[0]`: the read end of the pipe (sender should close). `fd[1]`: the write end of the pipe (receiver should close). If a receiver reads from an empty pipe, it blocks until data is written at the other end. If the sender writes to a full pipe, it blocks until the data is read at the other end and the pipe is cleared. **2) Named (FIFOs):** Persistent pipes that outlive the process that created them. Stored on the file system, can be opened like a regular file. More efficient than files as they **only exist in memory**. To use these we just do mkfifo <file_name>.

3) Threads

A thread is a separate flow of control through a program. Threads share the same address space. When multithreading is used, a process has one ore more threads. **Per Process we have:** Address space, global vars, open files, child processes, signals. **Per threads:** PC, Registers, Stack. Characteristics of threads: **hard to communicate between them bc diff addr spaces. Can block, causing entire app to be switched out.** Expensive to **context switch** as we change the entire addr space visible to the executing program. Expensive to create and destroy. Advantages vs Processes: Threads are much lighter, with less state. Shortcomings: **Shared addr space** / can read and write to other thread stacks causing memory corruption and concurrency bugs. **Unclear semantics forking inside a thread** – should we make a new process with same number of threads, or 1 thread? **Unclear signal handling semantics –** which thread should handle the signal?

3.1) PThreads

Operates like in True Concurrency. `#include <pthread.h> #include <sys/types.h>`

`int pthread_create(pthread_t *a, const pthread_attr_t *b, void *(*c) (void*), void *d)`

Creates a new thread stored in a. Function returns 0 if successful, errcode otherwise. b specifies thread attrs (eg min stack size, guard size) – NULL for default attrs. c is a function pointer – the start routine of the thread. d is an argument. `void pthread_exit(void *val_ptr)` terminates the calling thread and makes *val_ptr available to any successful to join with the exiting thread. Exit is called implicitly when the thread's start routine returns (except the initial thread which called main). If main terminates before other threads have exited, then they all get terminated. If pthread_exit is called in main then the process keeps executing until the last thread is terminated, or exit is called. `int pthread_yield(void)` release the thread's hold on the CPU to let another thread run. Returns 0 on success, errcode otherwise. Always works on Linux. `int pthread_join(pthread_t thread, void **val_ptr)` blocks the calling thread until thread terminates. The *val_ptr passed to pthread_exit by the terminating thread will be stored at the joining thread's **val_ptr.

3.2) User-Level and Kernel-Level Threads

Two ways to implement threads. 1) **User Level Threads:** The kernel is not aware of the threads – each process manages its own threads. **Advantages:** Better performance – creation/termination/switching are fast as none of these operations require kernel involvement, and each app can have its own scheduling algo (more flexible). **Disadvantages:** Blocking syscalls stop all threads in the process – denies the core motivation of using threads. Non-blocking IO is harder to use and understand. ***During a page fault the OS blocks the entire process even though other threads might be runnable.*** 2) **Kernel Level Threads:** Managed by the kernel – allows for True Concurrency. **Advantages**: Blocking system calls are easy to implement – if one thread blocks or causes a page fault the OS can schedule another thread from the same process. **Disadvantages:** Thread creation (can be mitigated with thread pool), termination, switching requires syscalls, more expensive. (much cheaper than process switches tho – as its in the same addr space). Apps can't have own scheduling algos. **A Hybrid Approach** is possible – using kernel threads and multiplexing a large number of user-level threads onto the kernel threads. Provides the **True Concurrency** of kernel threads, and lightweight switching of user threads.

4) Scheduling Processes

Goals: Fairness, Avoid Indefinite Postponement, Enforce Policy (like priorities), Maximise Resource Usage (CPU and I/O Devices), Minimise Overhead from context switches. **For Batch Systems**: We want high throughput and low turnaround. **For Interactive Systems:** We want low Response Time. **For Real Time Systems:** We have soft deadlines that should be met and **hard deadlines** which must be met. There are two types of scheduling – **Preemptive (time sliced context switches)** and **Non-preemptive**. **CPU** bound processes are those that mostly spend their time using the CPU. **IO Bound Processes** are those that mostly spend their time waiting for IO actions. **Scheduling Strategies: 1) FCFS:** We have a ready queue of scheduled process. Non preemptive. Once the running thread releases the CPU it gets blocked and added to list of waiting processes. Once its blocking operation is done its added to back of ready queue. **Pros:** Easy. No process starves forever. **Cons:** No concept of priority. If there is a long job followed by many short jobs, they will all have to wait a long time – could cause the system to hang. **2) RR:** We have a ready queue. **Preemptive** - After a process runs for a given amount of time we put it back in ready queue and switch to another process. **Pros:** Fair. Response time is fast for small number of jobs. High if we have many jobs – as queue will be long. Turnaround time is low when runtimes differ. **High if runtimes similar.** We can choose a **large/small** quantum (time slice) – **smaller/larger** overhead, **worse/better** response time. Most OS do 10-200ms – good tradeoff. **3) Shortest Job First: Non-preemptive** – at each stage we select process with lowest estimated/known runtime. Maximises throughput. **4) Shortest Remaining Time: Preemptive version of 3.** RTs must be known beforehand – at each step choose process with shortest remaining RT. When a new process arrives with less RT than the curr's then immediately switch over. When pre-emption occurs, switch over if there's a process with less RT. Two ways of getting RT – computing CPU Burst (time spent on CPU until no longer ready) estimates on heuristics (e.g. prev exec history) / user provides estimate (penalising users for underestimating). **5) Fair-Share:** Takes into account who owns a process before execution. For example: User 1: Processes: A, B User 2: C. Both get equal turns. The scheduler will give User 2's individual process twice the processing time as the first processes - as we alternate the top of each user's process queue, A, C, B, C,...) **6) Priority Scheduling:** Jobs are always ran on priority (externally defined or computed by a metric e.g. CPU burst). Priorities can be dynamic. **Processes could get starved** if one has a high priority but takes long. We might want priority decay or dynamic priorities (priority donation). **7) General Purpose Scheduling** – What modern OSes do. **Favour short and IO bound processes** – gets good resource use and response times. **Quickly determines the nature of a job and adapts** – processes can move from being IO bound to CPU bound, vice versa. **8) MLFQS – One queue per priority level**. Run the job on the highest nonempty PQ. Priority takes into account CPU/IO bound. Must avoid starvation of low priority (ageing?). Recompute priorities periodically (based on recent_cpu & ageing). **Pros**: Each Q can have own Scheduling Algo. Easy. **Cons:** Inflexible – apps have low control, priority makes no guarantee. System must be up for a while for good results. Processes can cheat by adding useless IO. No Priority Donation. **9) Lottery Scheduling – Preemptive.** Jobs receive lottery tickets for each resource type, like CPU time. A ticket is chosen at random and the job owning the tickets gets CPU time. **Pros:** Number of tickets is meaningful – jobs holding p% tickts should get p% resources. **Very responsive** – a job can be given more or less tickets and immediately have this reflected in the next decisions. **No starvation.** Tickets can be donated/exchanged. Adding and removing jobs affects others proportionally. **Cons:** Unpredictable, an unlucky process might wait a long time.

5) Synchronisation

Terminology: Critical Section (CS) – a section of code in which a process accesses a shared resource. **Mutual Exclusion (ME) –** Ensuring that if one process is executing a CS, no other does. Requirements for ME: **1)** No two processes must be in the same CS simultaneously **2)** No process outside the CS can prevent others entering the CS **3)** Processes needing to enter the CS can't be delayed forever **4)** No assumptions about speed of each process. **Race Conditions**, **Thread Interleavings** (number of possible exec. orders), **Memory Models** are the interactions of threads through memory and shared data. They depend on hardware behaviour and compiler optimizations. **Sequential Consistency** – The operations of each thread are executed in sequential order atomically. **Weak MM –** The hardware is permitted to reorder memory writes. A **Happens Before Relationship** (HBR) is a framework used to reason about race conditions in concurrency. We introduce a partial order between events in a trace. If a occurs before b, and they're in the same thread **OR** a is release(lock_1) and b is acquire(lock_1) then a -> b and we consider them ordered. -> is irreflexive (a -> a impossible), antisymmetric (if a -> b, then b -/-> a), transitive. Under this framework, a data race only occurs between a and b if they access the same memory location, one of them is a write and they're unordered according to HBR

5.1) Methods of gaining ME

1) Disabling Interrupts. Wrap our ME with STI() (set interrupts) and CLI() (clear). **Cons:** Only works with 1 processor systems, loss of performance, buggy processes might not yield CPU. **Only use it when necessary e.g. Kernel code.**

2) Strict Alternation. Processes take turns accessing shared resource. Each gets some time before yielding to the other.

```
char turn = 'A'; void process_work_a(void *arg) {while (true) {while (turn != 'A') {}
critical_section(); turn = 'B'; non_critical_section_a(); } void process_work_b(void *arg) {while
(true) { while (turn != 'B') {} critical_section(); turn = 'A'; non_critical_section_b(); } }
```

Cons: Busy waits. Can't pre-empt a thread that gets stuck forever. Assumes A and B are similarly fast – A could loop around fast enough that B doesn't get a turn. We might need A to work twice in a row but can't – **Peterson's Method** fixes this by using **a flag variable** (if a process isn't interested in taking its turn the other one takes its turn instead) as well as the turn variable.

3) Locks: A Synch Primitive that restricts access to a resource. `void acquire(*int lock) { while (TSL(lock) != 0) {} *lock = 1; } void release(*int lock) { *lock = 0; }` **Busy waits.** Also, we had to use TSL as it's atomic. Runs into priority inversion problem (fix with PD). As Locks get more Fine-Grained, **Lock Overhead** increases, **Contention** decreases, complexity increases. **Coarse-Grained** opposite. To minimise contention and max concurrency, do a fine-grained approach and release locks ASAP. **3.1) RW Locks** – Allow for Concurrent Read-only access. If one process is a write, then all others must block until it finishes. **4) Semaphores:** A structure which allows a progress to stop and continue only when it receives a signal. `init(s, i)`, `down(s)` (aka P()), `up(s)` (aka V()). Consists of a counter i and a queue of waiting processes. A semaphore init at 1 = lock, A sema at 0 is blocked until it is upped. So we can order events by initing at 0, running process A then upping it allowing other process in finally. It can also be used for **producer-consumer** relations: A producer adds items to a shared buffer if theres enough space and ME is ensured, a consumer removes them if ME is ensured and the buffer is not empty.

```
semaphore_t *i_+; semaphore_t *i_-; semaphore_t *b_lock; item_t *b[10];
void prod_work(){
    while(1){item_t *i=produce();down(i_-);down(b_l);deposit(b, i);up(b_lock);up(i_+);}
}
void consumer_work(){
    while(1){down(i_+);down(b_;);item_t *i=fetch(b,i);up(b_;);up(i_-); consume(i);}
}
```

We initialise i_- at 10, as we can have max 10 items in b. b_l is init at 1 so only 1 item can do prod_work or consumer_work. Explanation: for prod_work, we down(i_-) and b_l, to ensure we are only adding when we have space and for ME. At the end we up b_l and i_+ to signify we're no longer in a critical section, and that we can remove an item (consumer_work calls down(i_+) immediately).

5) Monitors: High level Synch Primitive - Allows ME and the ability to wait for **a signal or condition to become true or false.** A Monitor has: Shared data, Entry procedures, Methods that can only be called inside the monitor, An Implicit Monitor Lock and one or more cond vars. Only 1 process can be inside the monitor at once and has access to all its methods and shared data. They are a language construct – not provided by the OS, built ontop of the provided primitives. In Java every object/class is associated with a monitor, and synch blocks/methods use monitors.

6) Cond Vars are associated with a high level condition e.g. "some data has arrived in the buffer". wait(c) – releases monitor lock and waits for c to be signalled. signal(c) – wakes a process waiting for c to be signalled. broadcast(c) – wakes all waiting for c. Two ways to respond to a signal **Hoare:** A process waiting for a schedule is immediately scheduled. Easy to reason about, but is inefficient (proc immediately switched out), scheduler needs to do extra work. **Lampson:** Sending signal, and waking from wait not atomic. More efficient, more error tolerant, harder to reason with. Code wise – we would use an if check for Hoare, as we only get scheduled once the condition is met. While check for Lampson as we would busy check the condition multiple times whenever the process gets scheduled again.

5.2) Deadlocks: A state where processes are waiting for an event that can only be caused by a waiting process. Resource deadlock is most common – and it requires these conditions to occur: **ME, Hold and Wait –** a process can request resources while it holds another it requested before, **No pre-emption** – resources can't be forcibly removed, **Circular Wait** – two or more processes in a circular chain where each is waiting for a resource held by the next. We can use **Resource Allocation Graphs (RAG)** to model dependencies on resources and processes. Resource -> Process means the Process owns that resource. Process -> Resource, Process waiting for resource. Cycle = Deadlock.

5.2.1) Dealing with Deadlocks 1) Ignore – if it's rare this can work. **2) Dynamic Avoidance** – we evaluate every request to see if giving out a specific resource is ok. Every request provides info about resource usage. **3) Prevention** ensure one of the 4 criteria don't hold. **4) Detection and Recovery** – Dynamically build RAG, search for cycles. For recovery, can use one of 3 techniques **1) Preemption** – temporarily take resources from owner and give it to waiter. Could corrupt state **2) Rollback** – processes are checkpointed, on deadlock rollback before it. **3) Killing Processes** – select a process and kill it. The safety of this technique depends on workload – ok for compilers, not ok for databases.

6) Memory Management

Logical Addresses (LAs) – Generated by CPU, these are the Mem Addrs used by the system software including OS & Apps. **Physical Addresses (PAs) –** Refers to actual locations in the computer's main memory (the RAM). They're the same in compile-time and load-time address binding schemes but different in execution-time address binding schemes.

6.1) Memory Management Unit – To achieve the binding from LAs to Pas at high speed, this mapping is done in hardware. This means user processes only have to deal with LAs. Example MMU func: Add value (14000) stored at relocation register (RR) to every LA passed into it before accessing memory. LA 346 → PA 14346. RR holds lowest PA.

6.2) Memory Allocation and Protection. 1) Contiguous Memory Allocation

Works by splitting main mem into two parts. **Kernel** – held in low mem with interrupt vector, **User** – high mem for User Processes (UPs). UPs are given **contiguous ranges** of memory. MMU is then given two registers. **Base** – contains smallest PA for a particular range (as the RR did). **Limit** (contains the size of the range of addrs). We can do **Memory Protection** (each process only access own mem) by ensuring base <= addr from < base + limit. If not, segfault. **Multiple-Partition-Allocation**: Holes will appear in mem as processes are freed. The OS stores info about holes. We allocate to holes to new processes via specific techniques: **1) First-fit:** allocate in the first hole large enough. **2) Best-fit:** allocate in smallest hole large enough. Sorted list ideal. **3) Worst-fit:** allocate in largest hole.

6.2) Fragmentation: caused by inefficient use of memory, reduces capacity&performance. **External Fragmentation** - there's enough free memory for the allocation request but it's not contiguous. **Internal –** not enough free memory. **Compaction** fixes external frag by shuffling memory contents so that the free memory is 1 large block. Expensive.

6.3) Swapping: Number of processes is limited by mem. size. But many processes might not be running, they can be swapped out of mem to disk. We require dedicated swap space on disk, and swapping in/out mem is a lot of swap time.

6.4) Paging: Allows for LAs to be non-contiguous. **Frame:** a fixed-size block of phys. mem. **Page:** a block of logical (virtual) memory, the same size as a frame. When a progam with n pages is ran, we find n free frames, load the prog into memory and setup a pagetable to translate LAs to PAs.

6.4.1) Page Table (PT): Translation mechanism from logical mem to phys mem. Stores Mapping of Pages to frames. When translating an address the PT finds the frame the data exists in but we also need the offset into the frame. LAs consist of: **A Page Number** – an index into the PT, which stores the **Frame Number** corresponding to a page in phys mem (which is a base Addr). & **Page Offset** – combined with base addr the MMU can compute the exact page. Given a LA size 2^m and page size 2^n the page number has m-n bites and offset has n. e.g. m = 32, n = 12, LA = 0x12345678, page num = 0x12345, offset = 0x678. To keep track of which frames are free we can use a **free frames list**. Can be searched when a process needs to palloc, and removed and entered into the processes' page table. **Memory Protection** – To stop invalid accesses of logical memory, each entry has a valid-invalid protection bit. Valid = legal page, Invalid = Indicates page is missing because: page isn't in processes' LA space/incorrect access/need to load page from Disk (lazy loading). **Implementation:** PTs are large so are kept in Main Memory. The MMU has two regs: PT Base Register (points to base of table), and PT Length Register (stores size of it). Inefficient as each data/instruction access needs two mem accesses – one for PT, one for instr. **Associative Memory** is Fast lookup cache implemented in hardware. Mapping from Page Num to Frame Number. The cache is checked first before the page table in mem. Associative Memory stores recent virtual-to-physical. Massively improves performance.

6.5) Translation Lookaside Buffers (TLBs)

These high speed memory caches that speed up virtual mem addr translation are known as TLBs. They *usually* are flushed after Context Switches, and we'd get poor performance from TLBs until the cache warms. To avoid this, we store **address space IDs** in entries, which uniquely identify each process to provide address-space protection for them. This lets the TLB be shared across processes so we don't have to flush them during context switches. Kernel memory used to be in the v addr space of each process – more efficient as these pages didn't have to change in the TLB on a syscall. But it's been moved to a separate vaddr space recently for security so the TLB might have to be flushed on syscalls. **Effective Access Time**: Associative lookup = eμs, Memory cycle time = 1μs, Hit rate = a (fraction between 0 and 1) of times the page is found in associative regs. Therefore, the EAT = (e+1)a + (e+2)(1-a) = 2 + e – a. (e+1) = TLB hit time, (e+2) = TLB miss time.

6.6) Types of Page Table. 6.61) Hierarchical Page Tables (HPTs)

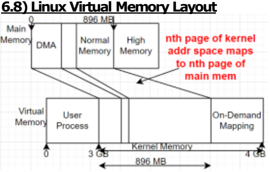
In PTs, the allocated array must be proportional to the size of the v addr space (as we need an entry for each frame). This is wasteful as it is usually only sparsely populated. We solve this with **HPTs**. **Two Level Paging:** We use an outer page table in conjunction with many inner PTs. The first level consists of a fixed size page-dir, with each entry being a pointer to an inner PT. The second level PT contains pointers to the actual pages of data stores in the main memory or on storage devices. **Thus we get to store a variable number of PTs** – as some of the pointers can be null until needed. **Each inner PT is a fixed size (all equal) and covers a range of phys mem.** An LA was divided into Page Num and P offset. Now that the PT is paged, we divide the Page Num up into p1 and p2 – index into the outer PT, and displacement into the page of the inner PT respectively. On a TLB miss we have 1 extra lookup. We can do n level paging – same principle. The size of the PT grows with the size of the v addr space. The size of the PT grows with the size of v addr space. Assuming a 4KB page size, offset size = 12. 32 bit machine – 20 bit for Page Num = 8 x 1MB per process. 64 bit = **30PB per process!** To fix, we can store entries per frame rather than process, as HaPTs and IPTs do.

6.62) Hashed Page Table – We have multiple v_addrs going to the same key index, and store a chain of p-addrs they map to. Done in hardware (to traverse the chain); MMU is more complex. Needs good hash func for min conflicts.

6.63) Inverted Page Table – **Not** per process. p_addrs are used as the key, and we store a list of v_addrs (and note down which process owns them). Reduces the memory needed to store it but lookups take longer (like for HaPTs).

6.7) Shared Memory – We can share memory between processes by mapping a page from each processes' virtual mem to the same frame. This frame is now **shared memory** – no need for kernel involvement but has no synchro. **SystemV's** (a UNIX version) API for memory sharing 1) **shmget** - allocs a shared mem segment 2) **shmat** - attaches a shared mem segment to the addr space of a process 3) **shmctl** - changes properties of a shared mem seg **4) shmdt** – detaches a mem seg from a process. **6.8) Segmentation** – The practice of dividing a computers primary memory into segments. (separate addr space for code, data, stack). Each segment is used to store a different type of data or execute a specific task. They are an independent address space from 0 to some max which can **grow / shrink** independently, support different protections (R/W/Exec), programmers can be aware of them. Not used much these days – paging is more popular. x86 uses paging and segmentation (for legacy reasons, used it when it was 32 bit). Memory allocation of segments can be harder due to their variable size, which may cause external fragmentation.

6.8) Linux Virtual Memory Layout



The bottom section is the view an executing user process has of its v_addr space. First 3 GBs in v_mem is where code, stack, heap are. Processes map the kernel to the 3-4GB range, for syscalls without TLB flushing. The kernel maps the lower 896 MB of phys mem to its virtual address space.

6.9) Linux Paging

x86-32: 4KB Page Size, 4GB v_addr space, two level PT. **x86-64:** 4KB/2MB/1GB, up to 4 level PT. Supports 48 bit addrs – 256TB. The 2 lvl PT uses offset bits in addr to store page status (dirty/read-only) **Meltdown Attack:** attack that let user program to access kernel_addr mapped in v_mem. Caused by **speculative execution** – a compiler optimization which means code is executed concurrently before it's needed. Should pagefault on the kernel-mem addr access.

```
s = a + b; t = s + c; u = t + d; v = u + e;
if (v == 0) {w = kernel_mem[addr]; x = w & 0x01;
          y = x * 4096; z = user_mem[y]; } /*…*/
if (v == 0) {w = w_; x = x_; y = y_; z = z_;}
```

The lines afterwards are ran and stored in intermediate vars: w_, x_, y_, z_, before it faults or even if v =/= 0. By arranging user_mem[0..4096] so it isn't cached, the access user_mem[y_] will cause either user_mem[0] or user_mem[4096] to be in cache after being speculatively executed. A program then can time how long it takes to access user_mem[0] and user_mem[4096]. If user_mem[0] is significantly faster then it must be in cache so x_ was a 0, otherwise if user_mem[4096] is significantly faster then x_ was 1. We've learned the value of a single bit in kernel memory. Scaling up we can leak lots of kernel memory, including the root password. New versions of linux fix this by moving k_mem to diff addr space. Expensive.

6.10) Demand Paging - Lazy Loading from PintOS. Reduces IO load when app is launched, less memory is used if some parts of exec never run, faster response times. Uses valid-invalid (VI) bit **1)** Set all the valid-invalid bits to 0 (not in memory). **2)** During address translation, if this bit is 0 then we trap to the kernel + page fault. **3)** The OS then checks another table to decide if the reference is valid or invalid. **4)** If the ref is invalid then the program is aborted. Else, the reference is valid but the page is not loaded into memory. Then: Get the empty frame, swap the page into the frame, update the page table by setting the VI bit to 1, restart last instruction. **EAT** = (1-page_fault_rate)*mem_access_time + page_fault_rate(page_fault_time + swapping_page_out_time + swapping_page_in_time + restart_instruction_time)

6.11) COW - allows parent process to share frames with children as read only. When we write to them, a copy is made. Saves cost of having to copy pages upfront, but child processes might be slow later on if they start to write a lot. Works by giving the child process its own PT pointing to the parent's frames that are marked as read-only. When either process attempts to write to one of these frames, the page protection causes a trap to the kernel. Then the kernel allocates each process its own private copy of the frame, and it marks them as read-write.

6.12) Memory Mapped Files (MMF): An MMF is a file whose contents on disk are mapped in the virtual address space of a process. When reading / writing to the file, the process can simply refer to the addresses in memory directly rather than using system calls. Through demand paging, these actions get translated into disk accesses. Provides an elegant programming model for I/O operations.

6.13) Page Replacement Algorithms: Strategy for Page Replacement should: Min PFs, prevent over-alloc of mem, use the dirty bit to reduce overhead of Page Transfers (unmodified pages = deleted, modified write back). **Belady's Anomaly:** Number of PFs can increase when number of frames increases. But as num frames continues to grow, it'll go down. Can be seen with replacement string 1,2,3,4,1,2,5, 1,2,3,4,5 with FIFO. 3 frames - 9 page faults, 4 – 10. 1) **FIFO** – Remove oldest page. 2) **LRU** – evict the page referenced longest ago – using timestamps or ref bits which are cheaper. 2.1) RefBit – Initially set to 0. When a page is ref, set to 1 (done in hardware). During replacement we pick a page which is 0. OS should reset pages periodically. 3) **Second Chance** – Improves LRU – we keep pages in an ordered (when brought into memory) circular linked list . **We give pages a second chance** – if they are old but used they're kept. Oldest page with unset bit is exited. 4) Page Refcounting Algorithms: **4.1) LFU** – Replaces page with lowest count. Could be a page which was just brought and didn't have a chance to get used. We could use reset counters or use ageing. **4.2) MFU** – the idea is to preserve those brought just into memory. **4.3) Working Set Clock Algorithm** – Builds on Second Chance but also uses a working set of **size w** to keep track of pages. At each page fault we examine the page pointed to: If the ref bit is 1, set to 0 and go to next page. Else, calculate the age: if age <= w, continue (its in our working set) else check if its dirty – if so write it back else evict it. The optimal size W changes based on what we're doing. For best performance we change it during runtime, we do this by observing the number of page faults. If it's a lot, we should increase W. **Local Strategy**: Each process is given a fixed number of frames and by observing fault freq we can change W. **Global Strategy**: Memory is dynamically shared between runnable processes. Initially memory is allocated proportionally to the process size, and by considering the page fault frequency the allocation to each process can be tuned. Linux uses Global; Windows Local. **Linux Page Replacement:** Uses a variation of clock. We have an **active list** with active pages, MRU at front. **Inactive list** has LRU at rear – only these are replaced. **kswapd** reclaims pages in the inactive list when memory is low in background, uses a dedicated swap partition/file. Must handle locked&shared pages. **pdflush** is a kernel thread. Periodically flushes dirty pages to disk. This happens in the background as it is an expensive process.

7) Device Management Objectives: 1) **Fair access** for processes to shared devices 2) **Exploit Parallelism** of IO devices for multithreadness. 3) **Provide an API for IO operations.** 4) **Uniform naming and error handling.** **Device Independence:** means the OS should abstract away from specific device types (terminal/disk/drive) and the device instance – nowadays all devices can be addressed as though they're indexable memory! **Device Properties:** 1) **Unit of Data Transfer:** Character – we write a stream of characters or Block - we access fixed size blocks of data at a time (Buffer cache has data written to it in a stream until it becomes a block. Faster than char) performance for this). 2) **Supported operations** (e.g. read, write, seek). 3) Synchronous or asynchronous operation. 4) Speed. 5) Shareability: Sharable (e.g. disks). Single user (e.g. printers, DVD-RW). 6) Type of error condition

7.1) IO Layering

User Program -> OS -> Specific Device Drivers -> Specific Device Controllers -> Individual Devices
^User Space^ ^ Kernel Space ^ ^Hardware^ ^Devices^

Controllers are implemented in the hardware. Interfacing with controllers is done by Device Drivers in kernel space. The OS provides abstractions on top of the interface provided by Device Drivers, used by user programs.

Layers: User-Level IO software <- device-independent OS software <- Drivers <- Interrupt Handlers <- Hardware

1) Interrupt Handler Layer: Process interrupts caused by devices. **Block:** usually on transfer completion, **Character:** after characters are transferred: e.g. keyboard **2) Device Driver Layer:** A device driver handles one or more devices of a type. They must: 1) implement block reading / writing 2) access device registers 3) initiate operations. 4) schedule 5) handle errors e.g reporting corrupted disk blocks. **3) Device Independent OS Layer:** Provide a Uniform Interface for device drivers. A standard driver interface can make it easy to implement device drivers – crucial as they're often written by third parties. The device independent layer provides device independence. They should: **1)** have a mapping from logical -> physical devices, e.g. Rather than physically referring to disk we refer to partitions/volumes. **2)** validation of requests against device characteristics. **3)** Allocating devices correctly: ME for processes using that device. **4)** dealing with user access validation. **5)** buffering & caching blocks for speed & block size independence **6)** error reporting. **3.1) Dedicated and Shared Device Allocation:** Dedicated devices have simple policies (e.g. fails if opened twice), are usually allocated for long periods of time & are only allocated to authorised processes. Shared devices are used by many user processes concurrently. **3.2) Spooling** – Spooling is a queueing system for non-shareable devices. For an allocated spooled device, when jobs are submitted, the OS instead can save the job to disk. The spooler daemon is the only process allowed to directly control the device, it picks up the safe jobs and submits them to the device, one by one. Allows sharing of non-sharable devices, and reduces I/O time. **3.3) Buffering** - Buffering allows blocks of data to be stored in the buffer cache, so requests to a disk for data can be cached and retrieved rather than the device having to get the data again. **3.3.1) Buffered and None Buffered IO:** Buffered I/O can be used for both reading and writing helping smooth peaks in I/O traffic and accommodating differences in data transfer units between devices. When writing, data is first stored in a buffer until it's full, then flushed to the device. When reading, the OS may read ahead to prefetch additional data for faster access. Some applications prefer unbuffered I/O, which transfers data directly from user space to the device, incurring high process switching overhead such as Databases which may opt for unbuffered I/O to implement a more effective caching strategy.

4) User I/O Interface Layer: Here, the act operations that a process may want to use (e.g. open, dose, read, write, seek for a disk) are exposed. I/O libraries also configure other types of I/O params, e.g. blocking or not I/O, synch or asynch I/O, etc. This layer must provide some kind of abstraction of how to expose devices. Unix, does by representing them as files

7.2) Device Drivers 7.2.1) MMIO: To implement drivers we need to interact with hardware. This is done with MMIO. This means particular device registers reside in mem addr space. For example to disable the I2S clock on a Raspberry Pi:

```
*(clk + 0x26) = 0x5A000000; *(clk + 0x27) = 0x5A000000;
```

The MMU recognises some particular addresses as devices connected to the CPU. **7.2.2) IO Methods: 1)** Programmed I/O: directly writing to registers to control the device - inefficient as I/O devices are much slower than the CPU, wasting CPU cycles. **2)** Interrupt-Driven I/O: CPU gets interrupted by the I/O device when the device driver that it can interact with the device again - may result in many interrupts. **3)** Direct Memory Access (DMA): the CPU hands more control over to the DMA controller, relieving the CPU from handling many interrupts, as data is transferred to and from the DMA controller without interrupting the CPU.

7.2.3) Linux – Loadable Kernel Modules (LKM): LKMs provide device drivers. They contain object code that is loaded on-demand, dynamically linked to the running kernel. Made by third parties. Require binary compatibility: Modules written for different kernel versions may not work. Kmod is the kernel subsystem for managing modules without user intervention, it determines module dependencies, and loads modules on demand. Every LKM module consists of two basic functions (at a minimum): int init_module() and void cleanup_module(). insmod module.o loads a module – usually a root privilege. **Blocking IO**: IO calls only return when done. Processes get suspended – they only continue once the operation has returned. The process sees the operation as instant. Easy to understand, but leads to multi-threaded code. **Non-Blocking IO**: IO calls return as much as available at that moment. fcntl sys call turns it on for an fd. Provides application-level polling for I/O (monitoring the application) **Asynchronous IO:** The process executes in parallel with some daemon working with the IO app. When the IO app finishes the process is notified. Supports checking / waiting until its done – flexible and efficient.

7.3) Linux IO API - distinguishes operations in terms of the class of device: 1) **Character** (unstructured): files and devices. 2) **Block** (structured): devices. 3) **Pipes** (message): IPC 4) **Socket** (message): network interface. Sockets are an I/O abstraction that allow for bidirectional communication between processes on a single machine or devices over a network. They are a separate I/O class due to the unique nature of network communication, which can involve variable amounts of data (packets) and potential packet loss. There are two types of sockets: TCP (stream sockets) and UDP (datagram sockets), each with their own protocols to handle communication.

`int create(const char* path, int perm)` Returns an fd, takes a file path and opens it for reading and writing (makes file object). perm for access control

`int open(const char*name,int mode)` Return fd. Mode 0: R, 1: W, 2: RW

`void close(int fd)` Closes the file or device referenced by fd.

`int read(int fd, void *buffer, int numbytes)` Returns the number of bytes read. Reads numbytes from file open as fd into buffer.

`int write(int fd, void *buffer, int numbytes)` Returns the number of bytes written. Writes numbytes from buffer into file open as fd.

`void pipe(int* fds)` Creates a pipe. fd is an array of two ints: fd[0] is for reading, fd[1] is for writing. dup and dup2 duplicate fds, dup returns it, dup2 takes in the return param.

`void ioctl(fd,op,&termios);` Used to control io devices, termios = array of control chars.

`fd = mknod(filename, permission, dev);` mknod creates a new special file e.g. a character or block device

Processes have unique fd tables. When a process is created it has 3 fds, stdin, stdout, stderr. By default they all refer to terminal from which the program started.

8) Disk Management - Capacities (and demand for it) have increased a lot but access speeds haven't. Disks -> split up into platters -> split up into tracks (rings) -> split up into sectors (parts of a ring). The sectors store the actual data. The platters are stacked on a spindle and a read/write head is able to move across all of the different surfaces simultaneously and can read all of the tracks that are vertically above each other. Once the platters are spinning, this lets the read-write head read all the information on that track. This multi-layered track is referred to as a cylinder – the unit of information that can be read from or written to in a single operation. **HAMR: Futuristic HDDs**: Heat Assisted Magnetic Recording allows for information to be stored in a more compact part of the platter surface. Heating up the medium before storing the bit allows for a more precise write. Can reach density 1.5Tb/inch^2. **8.1) Sector Layout:** Nowadays, Disks have virtual geometries seen by the software, abstracting from the complex physical geometry. Thus we can maintain uniform storage density on platter surfaces, despite them having sectors with varying capacities. Previously, the OS would refer to a particular cylinder number, a surface, and then a sector, uniquely selecting 512 bytes of information (assuming this is the sector size). Modern discs use LBA, sectors are numbered from 0 to n nd can be directly referred to by their sector numbers. Makes disk management much easier, and it helps to work around BIOS limitations. **Disk Formatting:** Imposing structure on the disk. **Low Level Format:** Gives just enough structure so sectors can be read reliably. Each sector indicates the start, has 512 bytes of data, has an error correcting code at the end. Cylinder skew and interleaving were also imposed by low level formatting in the past. **High Level Format:** Done when adding file systems or partitions to the disk. Includes: boot block, free block list, root directory, empty filesys. **Disk Performance:** Hard Disks are slow as they rely on moving parts. Delay is composed of by **1) Seek Time 2) Latency Time 3) Transfer Time**. A typical disk has Sector size 512B, seek time (adjacent cylinder/track) < 1ms, avg seek time 8 ms, rotation time 4ms, transfer rate >100MBps. b = bytes to be transferred, N = bytes per track, r = rotation speed in revolutions/s: Latency time is 1/2r, transfer time is b/rN and **Total Access Time:** seek time + 1/2r + b/rN. **8.2) Disk Scheduling** - For disk requests we want to minimize seek and latency times as they're the biggest factor to delays. Either the OS or disk controller will order the requests with respect to head's position. 1) **FCFS**: Requests are ordered, causing random seek patterns. This is okay for lightly-loaded disks, but has poor performance for heavy loads. Scheduling is fair. 2) **Shortest Seek Time First** (SSTF) Requests are ordered according to the shortest seek distance from the current head position. Discriminates against the innermost / outermost tracks. It has unpredictable and unfair performance. 3) **SCAN Scheduling:** Requests are chosen which result in the shortest seek time in a preferred direction. The RW head only changes direction when it reaches the outermost / innermost cylinder (or there are no further requests in the preferred direction). This can cause long delays for requests at extreme locations. 4) **C-SCAN** Requests are only served in one direction only. When the head reaches the innermost request, it jumps to the outermost request. This lowers the variance of requests on the extreme tracks, but it may delay requests indefinitely. 5) **N-Step SCAN** C-SCAN, but requests are only serviced if they were waiting when the sweep began. This has the benefit of it being impossible to delay requests indefinitely. 6) **What Linux Does**: I/O requests are placed in a request list. There is one request list per disk block device in the system. The block device driver for a disk must implement a function that can execute a request. The kernel can then pass an ordered request list (according to how it wants to schedule those requests) to the device driver. Linuxuses a variation of the SCAN algorithm. The kernel attempts to merge requests into adjacent blocks so they can be done together. The kernel also wants to avoid the scenario where synchronous reads may starve during large writes. Linux uses a deadline scheduler, which ensures that read requests will be performed by some deadline, even if it results in a longer seek time. This eliminates read request starvation. The Linux Scheduler also uses Anticipatory Scheduling: if a process issues one read request, there's a high likelihood that it will submit another read request after a short period of time. We thus have a small delay afer each read. This can help reduce excessive seeking behaviour, but it can also lead to reduced throughput if the process does not issue another read request to a nearby location. **SSDs** - Non-volatile memory device. Array of transistors. No moving parts. Organized as Dies containing Planes containing Blocks containing Pages of memory. Enables parallelism – multiple parts of the SSD can be accessed in parallel (num concurrent RW ops depends on bus width). **SSDs vs HDDs:** obvious. **8.3) RAID** - **Level 0:** Data is spread out across multiple disks, allowing for concurrent seeking / transferring of data. Can also balance the load across disks. No redundancy – fragile. **Level 1:** Data is mirrored across multiple disks. This is to introduce a redundant backup of data, providing easy failure recovery. Reading from the disks is fast as each copy may be used to service the request, but writing is slower as the data needs to be written across all disks in parallel. Storage overhead cost is high in RAID 1. **Level 5:** Parity strips (block of data that is calculated based on XOR of the corresponding blocks from the other drives in the RAID array. Allows for the recovery of data in the event of a single disk failure) are distributed but parity strips are distributed across all the drives in a round robin fashion. This means that there is no longer a single parity disk that becomes the bottleneck. There is potential for concurrent writing, and there is a good storage efficiency / redundancy trade-off. However, data reconstruction of a failed disk is non-trivial and is slow. **8.4) Disk Cache** – We cache sectors of the disk with a buffer in main memory to reduce access times. Since the space is limited we require replacement policies: **1) LRU:** Replace block that was in cache longest without references. The Cache contains a stack of blocks - the most recently referenced block is at the top. We simply remove that block at the bottom. This method does not keep track of a block's popularity - we might remove some goated block. **2) LFU:** Replace the block with the fewest reference (each block has a counter) – each time reffed increment. We replace lowest ref. If a block is referenced a lot in a short time and not used again it'll be kept. We can use ageing to fix this (*second chance*). **3) Freq Based Replacement:** To prevent items that get a sudden burst of accesses from lingering too long in the cache we only increment blocks not in new. Only blocks in old are evicted. To prevent pages incrementing too quickly in old we could have a "mid" which allows for incrementation but not eviction.

9) File Systems – organise info. Objectives: Non-volatile storage, sharing info, concurrent and convenient access, easy management of data, security. **File Naming:** Any name, extension (indicate file usage). **File Types: 1)Hard Links:** A file that aliases the data of another file (refers to location of data). **2)Soft Links:** A link which aliases the path of the file to another (file points to another file, which points to the data). **In** command soft links files. **3)Character Special:** Provides access to a char IO device. **4) Block Special**. File system support functions: Name translation (convert paths to logical disks and blocks for driver), File locking for exclusive access. The obvious: caching/buffering/manage disk space/protect against failure/security. Files can have metadata:Owner, authentication (pw), actions: RW, Delete, usage info: (creation, last modified, expiry dates). File size is variable and is allocated on the disk in **blocks**, if its too small then we have high overhead for large files, high transfer time. If its too small we have internal frag, and lots of wasted space. Caching becomes harder (based on blocks). Usually do 512-8192B. **9.1) Ways of accessing blocks of a file:** 1) **Contiguous File Allocation:** Place file on contiguous stretch of addrs on a device. **Pros:** Reduces seek time between successive logical blocks. **Cons:** External frag, if unable to alloc new block as file grows, must transfer to new section (expensive). **2) Block chaining:** Each file has a linked list of blocks which is traversed. **Pros:** Files grow without realloc. No external frag. **Cons:** Pointers take space. Lots of seeking to traverse files. Small block sizes massively reduce performance. **3) Block Allocation Table:** A directory maps files to their first block, and a table maps blocks to their next. **Pros:** Fast lookup – no long seeks. No external frag. **Cons:** Requires periodic defrag. Tables can become very large and no longer cacheable in main mem. Windows uses this for FAT16/32, table cached in mem. Modern systems use **4) Index Blocks (IBs)** – Each Block Allocation Table has 1 or more IBs. They contain an array of pointers to data blocks & pointers to subsequent IBs. **Like PTs but for file systems. Pros:** Can search through index blocks to find location of file blocks easily. Index blocks are per-file, can load a file's table, rather than have an enormous global table as with FAT. For small files with good performance for big files. No external fragmentation & can extend files easily. Index blocks can be cached in memory just like data blocks. Linux does this through **Inodes 5) Inodes:** when a file is opened we also open an inode table and create an inode entry in memory. The struct contains: **Type and access control, number of links, User ID, Group ID, access/modification/inode change time.** It contains disk device num, inode num, number of processes which have the file open, Major/minor num (Major: type of device (disk / network), Minor: specific instance). Indirect Pointers can increase the num of data blocks associated with a file (IPs point to blocks with more IPs or a pointers to a block). **1)** We have an OS with inodes containing: 6 direct pointers, 1 indirect and 1 doubly indirect. Pointer = 8B, block = 1024B, each indirect block fills a whole block on the disk. We can compute max file size: An indirect block holds 1024/8 = 128 pointers to other blocks. The maximum number of blocks = 6 x 128 + 128 × 128 = 16,518 blocks = 16,914,432 bytes. **2)** We can also determine the number of disk reads required for accessing a block. The same setup, we can calculate the disk block reads required to get the 1020th data byte, and then 510,100th data byte, assuming nothing is cached in memory. 1020th byte = 1st block. Hence 1 read required to get the inode, and 1 required to get the data from the block pointer & the offset of 1020 bytes. Requires 2 reads. 510,100th byte 510100 / 1024 = 499 → 499th block. Each block of pointers contains 1024/4 = 256 pointers. Hence this block is pointed to indirectly by the doubly indirect pointer. Thus 1 read to get the inode, then 2 reads to get the data block, then 1 more to read from the data block. Requires 4 reads. **9.2) Free Space Management** – We need quick access to free blocks to alloc (Free List). We can implement the **1)** Free List using a Linked List. Alloc from start, add newly freed to end. Inexpensive, but non-contiguous blocks increasing file access time. **2)** Bitmap: Each bit represents a block at an index. Uses low memory, quickly determine contiguous blocks (for placing file's – min seek time), bit operations are very fast (O(1)). May need to search entire map to find free entry (slow). **9.5) File System Layout**

|Boot Block|Super Block|Inode Bitmap|Free Block Bitmap|Data & Inode Blocks:|…||.

Boot Block: Stores data to be loaded to start OS. Super Block: Contains info like: Num inodes, num data blocks, start of the Inode & Free space bitmaps, location of first data block, block size, max file size. **Directory organization –** Directories map symbolic names to logical disk locations. Prevent naming collisions (names only have to be unique in a dir), and help file organisation. Operations on Directories: Open, close, cd, ls, unlink read, write of file attributes. We can search files with regex. **Mount:** Link to another file system such as remote server or another disk. When mounting: Combine many filespaces into single namespace, allow ref from one root dir, **mount point** – the folder from which the mounted file sys can be accessed from the native one. Support soft links **but not hard links (dependent on filesys implementation)** - done using a mounting table which records location of mount points and devices. When a mount point is encountered the native filesystem consults the mounting table to determine which file system to go to. When a mount point is encountered the native filesystem consults the mounting table to determine which file system to go to.

Hard Links are not allowed for directories as may result in an **island** (hard link points to itself, or two point to each other) – can't ever be deleted. Directory Entry Struct: Inode number, offset into the directory, length of the record, file type, file name. **9.4) Linux ext2fs** The second linux extended file sys is a high performance robust file system. Uses block sizes of 1024, 2048, 4096, 8192. 5% of blocks are reserved for the root. (ensure root processes can run even if malware takes entire disk). ext2 inode is used to represent files & directories. It stores access, permissions, and other metadata. It uses 12 direct pointers, 1 indirect, 1 doubly indirect and 1 triply indirect (fast access to small files, and allows for large files). It uses **Block Groups**, and attempts to store related blocks in the same group to reduce seek time. **10) Security** – Objectives**: Data Confidentiality, Integrity, System Availability** (preventing DOS). **Policy:** Specifies what security is provided, what is protected, who has access, what access is allowed. **Mechanisms: How** we implement security.**People, Hardware, Software Security**. **10.1) People Security: Insiders**: people who abuse elevated permissions to cause dmg. **Social Engineering:** Blagging (personal fishing – customer support fake), Phishing, Shouldering. People are vulnerable bc they might ignore security for convenience, might not know, and the system is as secure as the weakest link. **10.2) Hardware Security: Physical access** – once acquired, a malicious entity can: read and alter content, snoop, or destroy the machine. Vulnerabilities: hardware flaws (meltdown attack, bugs). **Side channel attacks**: Gaining info about implementation of a system. Cache attack (using cache accesses by a victim to gain info), **Timing attacks** (measuring time taken by victim to gain info), **Data Remnants** (reading info after user has deleted it – might be in memory / might not have been fully wiped, just marked as free on the disk). **10.3) Software Security:** Systems can be compromised through software (unauth read, or RW DOS). Common exploits include: Buffer overflow (strcpy), Integer overflow (SSH had this bug), String formatting (Log4J). **10.4) Access Control: Principals**: Users, User groups or processes that may want to perform actions. **Authentication**: Verifying the identity of principals. **Authorisation**: Only allow principals to perform actions when authorised. **10.4.1) Authentication** – can be done on 1) **Personal Characteristics** (Biometrics, fingerprint). **+:** Hard to forge, convenient. **-:** Requires special hardware, false positive/ negatives. 2) **Possessions**: **+:** Convenient, **-:** can be lost/ stolen and devices are expensive 3) **Knowledge:** Passwords. Password turnover should be high otherwise crackers have more time. A long uncommon password can only be guessed by brute force. **+:** Cheap, secure. **-:** Password often reused (and so they're only as strong as the weakest system using it), if they're stored (LastPass or a safe) they're only as secure as the storage, Dictionary attacks. **New systems** use a 1 way cryptographic hash – so PWs can't be found in a leak. **Rainbow Tables** can crack hashes as hashes of common PWs are added to a database, allowing hashes to be converted back to PWs. **Password Salting** stops this – when hashing another value *s* is used. Randomly created on account creation and stored with the hash. Computing a table now becomes unfeasible as every possible salt must be used. Two identical passwords very likely don't have the same hash as a result. **10.4.2) Authorisation** – determines which principals can access what objects, and how they should. **The Principle of Least Privilege**: Users should always be given out the minimal number of permissions to do their tasks. **Access Rights:** a set of objects with the operations permitted on them. Each principle has a domain specifying their access rights. **Access Control Matrices:** Specify rights of each principal on all objects. An actual matrix isn't used as it'd be too big / we'd have to synchronize a massive structure. We implement ACMs as: **Access Control Lists** (Each object has a list of principals and the ops those principals can perform on it) or a **Capability List** (Instead each principal has a list of objects and the ops they can do on each. Kernel holds the info, or in user memory but encrypted). **ACLs vs CLs: 1)** Principle of Least Privilege: **Cap** better, can be finegrained. ACLs use of groups mean all users in a group get access rights. **2)** Revocation: ACL, can revoke all access by changing ACL for 1 object. For Cap we have to search all Caps. **3)** Rights transfer: **Cap** can transfer capabilities from principal to another easily. Can't for ACL. **4) Persistence**: ACL: File Systems are designed to be persistent. Cap – complex. In **Linux**, users are principals. They have a unique UID, the superuser (root 0) can access everything. Files are objects (remember, Linux: everything = file). Each user can belong to multiple groups (principles group together, thus we have to produce less policies). Each file belongs to **one group**. When a file is executed, it runs with the privileges of the user executing, unless SUID bit is set (in which case the file runs with the owner's privileges). Thus users have 3 UIDs: The Real UID, (UID of user started process) Effective UID (of process) and Saved UID (UID the effective UID that can be switched to). **Discretionary Access Control** (DAC): Principals determine who may access their objects. This is the default in UNIX/Linux, file owners determine the access rights. **Mandatory Access Control** (MAC): Precise system rules determine object access. A policy determines access controls for files, devices etc. Usually done by some single entity - the superuser? **Policies**: 1) **La Padula Model:** A MAC policy where information cannot travel down the security hierarchy. Objects and principals are assigned a **security level** (e.g unclassified, confidential, top secret) **Simple security property:** only read info of data of levels below us. **The * property:** only write to data of levels above us. Ensures **confidentiality** but not integrity. A low level object could write over very high levels and destroy it. Something at a high level could read something at a low level and write to itself, even though it might be garbage – hurts integrity. 2) **Biba Model:** A MAC policy guaranteeing data integrity. Inverts La Padula. **Simple Integrity Property:** Can only write to objects of the same or lower security level. **Integrity * Property:** can only read objects of the same or higher security level. In practice we could use both as we want both at different times. We often have some labels with rules on data propagation defined by those labels. We may have to change labels to - a privileged operation.

10.5) Overall Design Principles for Security Design Principles:

Least Privilege: Each process should run with the lowest privilege possible. Default should be no access.

Simple & Uniform Mechanism: ensures design is easy to reason about, and can be applied to many types of principals & objects.

Psychologically Acceptable: If policies are too much of a burden, too complex, or difficult to deal with then people will not implement them.

Public: By making the designs public they can be critiqued for flaws.