1 Kernels

Can be: **monolithic** (single *black-box*, like Latex). Single exec. bin with its own addr space. Efficient calls, easy to write but complex design & no protection within internal components. **Microkernel** (min. functionality). Simple, less error prone, clean server interface, server can crash without bringing kernel down but high IPC overhead. **Hybrid** - more structured design but performance hit for user level servers.

2 Processes

Provide **illusion** of *concurrency*, **isolate** procs, for simpler programming & better resource man. OS **context switches** in response to **events**/**interrupts**, (*or when we need to change the addr space*). Data about procs is kept in **proc control block (PCB)**, in the *proc table*. Each proc has a *virtual CPU*, its own addr space, open file descs, *PC*, page table register, stack ptr, proc man info, file man info. Context switches have a **direct cost** & an **indirect cost** (*cache misses*, *TLB misses* & *pipeline flushes*).

- `void exit(int)` - term with statcode.
- `pid_t fork()` - Duplicates proc. If < 0, err. Returns child PID in parent proc, and 0 in child proc.
- `pid_t waitpid(pid_t, int*, int)` - waits for child pid, stores child stat at pointer, can take optional flags (e.g. `WNOHANG`).
- `void kill(pid_t, int)` - kills pid with `SIGKILL`, `SIGTERM`, `SIGUSR1`, etc.

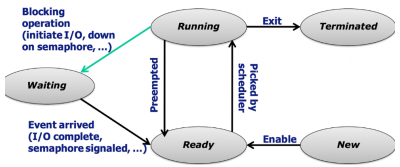
Proc communication can use files, signals, events, pipes, message queues, sockets, shared memory, semaphores or more. **Inter Proc Comms (IPC)** allows **signals** to be sent similar to hardware interrupts. By default, signals terminate a proc, but may be ignored or handled. `SIGKILL` & `SIGSTOP` cannot be ignored. Use when exception occurs, for notifications, `kill()`, etc.

Pipes connect the `stdout` of a proc to the `stdin` of another for *one-way* comms.

3 Threads

Threads are **exec streams** that *share an addr space*. In multithreading, a proc may use ≥ 1 threads, with individual *PC*, *regs* & *stack*. However, thread may write to another's stack in **mem corruption**, & **concurrency bugs** may occur. So, we have ***pthreads***:

- `int pthread_create(pthread_t, const pthread_attr_t, void*, void*)` - starts thread, places id in pthread, uses attrs (`NULL` default), runs specified `void *` function with `void *` args.
- `void pthread_exit(void *)` - terminates thread and makes value available to any successful join. Called implicitly after start routine returns.
- `int pthread_yield(void)` - give up CPU. 0 if success.
- `int pthread_join(pthread_t, void**)` - waits for thread to finish, and stores retval at pointer.



Threads can be implemented as **user level** (kernel is not aware of threads): fast creation, termination, switching & sync. Each program may have its own scheduler, but blocking syscalls stop all threads, & non-blocking IO may be used. **Kernel level** threads are managed by kernel. Thread creation, termination, switching & sync is *slow* requiring *syscalls*. However, blocking syscalls are easily accommodated. Hybrid approaches exist.

4 Scheduling

A scheduler chooses which threads to `preempt` & `dispatch`, ensuring *fairness*, avoiding *starvation*, enforcing *policies*, minimising *overhead* & maximising *resource utilisation*. For **batch** systems, throughput and turnaround time important. Prioritise response time for **interactive** systems. **Non-preemptive** - let proc run until it blocks / releases CPU.

- **First Come First Serve (FCFS)** non-preempt, each thread added to queue. Easy to implement, no indefinite postponement. But, **Head of Line (HOL) blocking** if a long proc is at the front of the queue.
- **Round Robin (RR)** is non-preemptive, every thread is given a time quantum to run. It is fair, has better response time. A larger quantum means smaller overhead but larger response time. A good quantum should be much larger than context switch time, but still provide low response time.
- **Shortest Job First (SJR)** is non-preemptive, shortest job is run based on known run-times or a heuristic.
- **Shortest Time Remaining (STR)** is preeptive versino of SJR. Allows new short jobs to get good service.
- **Fair Share RR** divides the CPU into fractions *between users*, using RR within each.
- **Priority** - jobs are run with highest priority. Can be either *static* or *dynamic*, supplied by the user or calculated.

In reality, many of these are used. We want to favour **short** & **IO bound** jobs for shorter response times & quickly determine the nature of a job & adapt to changes:

- **Multilevel Feedback Queues** - each priority level has queue, which usually use RR scheduling. Job priorities periodically recomputed with *exponentially weighted moving average*, & starvation is prevented with *aging*. However, it is not flexible, reacts slowly to changes, cannot donate priority, & allows cheating.
- **Lottery Scheduling** - jobs receive n tickets for a resource. When scheduling a resource, the job with the most tickets is most likely to receive it. Highly responsive & no starvation, but unpredictable response time.

5 Synchronization

In a **critical section** of code a proc or thread accesses a shared resource. **Mutual exclusion** is required s.t. ***(1)*** no two procs are simultaneously in a CS, ***(2)*** no procs outside the CS may block other procs from entering the CS, ***(3)*** no procs wait forever to access CS, ***(4)*** no assumptions are made about the speeds of the procs. We can achieve this by:

- **Disabling interrupts** - only used by single-core kernel, not recommended.
- **Strict Altercation** - uses global access flag. Violates ***(2)***, only should be used when waiting is short (violate ***(4)***).
- **Peterson's Algorithm** solves this:

```
val turn = 0;
val interest = {0, 0};
def enter_critical(thread: Int) =
  val o = 1 - thread; // Other
  interest(thread) = 1;
  turn = other;
  while (turn == o && interest(o));
def leave_critical(thread: Int) =
  interest(thread) = 0;
```

- **Spin Locks** - locks that need busy wait. They also suffer from **priority inversion**, the low priority thread holds the lock required for a high priority thread.
- **Locks** within a proc, **mutex** are across procs. User locks dont store list of waiters.
- **Read/Write Locks** treat R/W seperately - exclusive access in W, not in R.

Semaphore: *The scheduler must maintain a queue of threads waiting on the semaphore when its value is zero.*
Lock: *The scheduler tracks the owner of the lock and a queue of threads blocked waiting for it.*
Condition Variable: *The scheduler maintains a queue of threads waiting for a signal on the condition.*
Mutex: *The scheduler tracks lock ownership and ensures mutual exclusion among threads.*
Monitor: *The scheduler enforces mutual exclusion and manages condition queues within the monitor.*
Lock granularity allows specifying CS size *(e.g. a record in a DB)*. We choose fine granularity to reduce **lock contention** (*num threads waiting on lock*), but this increases overhead. **Read/write locks** give excl access in write, but simultaneous access in read.
Race condition occurs when > 1 threads R/W shared data, and final result depends on timing of their execution.
A **strong mem model** we assume sequential consistency (atomic ops are exec in-order), so we can use **semaphores**. `down(s)` recieves a signal (wait) on `s`. `up(s)` transmits a signal to `s`. `init(s, n)` inits `s` with val `n`. In mutual exclusion $n_0 = 1$. In ordering $n_0 = 0$ where initial n_0 indicates how many procs can access shared data simultaneously.
A **monitor** has *shared data*, *entry & internal procedures*, *an implicit lock*, & ≥ 1 *condition variables*. It can `wait(c)`, `signal(c)`, `broadcast(c)` on condition `c`. Signals do not accumulate, lost if no waiters.

6 Deadlocks

The 4 conds are: ***(1)*** **Mutual Excl.** - each res available or asgned to 1 proc; ***(2)*** **Hold & Wait** - a proc can request res while holding others; ***(3)*** **No Preemption** - res given to a proc cant be forcibly revoked; ***(4)*** **Circular Wait** - procs in a loop, waiting for res held by next.
We can *ignore*, *recover* (detect *cycles* in graph, & recover with *preemption*, *rollback* or *killing procs*), *dynamically avoid* (consider every request) or *prevent* (break a deadlock condition).
Deadlocks may also happen in communication. **Livelock** means no blocked procs but system not making progress (e.g. *receive livelock* or *starvation*).

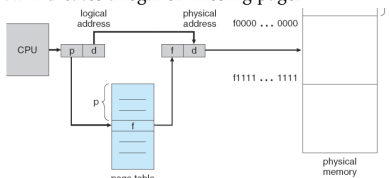
7 Memory Management

Memman needs *allocation* & *protection*. Use **memman unit (MMU)** to bind *logical space* to *physical space*. Mem is split into **kernel** (*low addr mem*) & **user** (*high addr mem*) partitions. In **contiguous alloc**, ***base*** reg contains smallest phys addr of proc & ***limit*** reg contains range of logical addrs. MMU maps logical addrs dynamically. If the logical addr is outside the user segment, we `SEGFAULT`.
Holes (blocks of avail mem) are scattered. When a proc arrives, it is given a large enough hole. OS maintains info on alloced & free partitions. We can use *first*, *best* (smallest hole) or *worst* (largest hole) fit. **Fragmentation** may occur. ***Internal frag*** when alloced mem > requested mem. ***External frag*** when mem allocated not contiguous. **Compaction** shuffles memory to place free memory together in one block. *Slow*. To stop mem limiting proc count, we use **swap** part. *Trans time* is bottleneck.
Virtual mem seperates logical & phys addrs. Only part of the proc needs to be in mem for exec, & logical addr spaces can be very large. Addr spaces can be shared, allowing fast proc creation. Done via *paging* or *segmentation*.

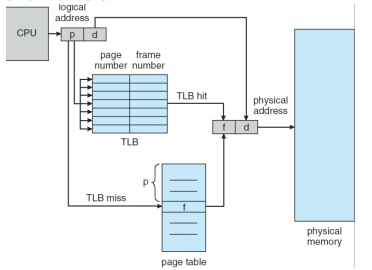
7.1 Paging

Calc p and d!

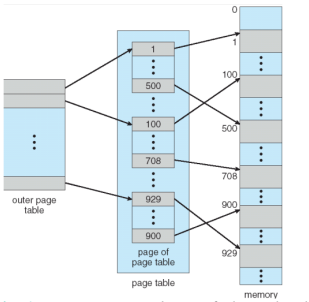
Paging - *logical addr spaces can be noncontiguous.* Procs are alloced phys mem in *fixed-size* **frames**. Pages are the same size as frames. To run a program of n pages, find n free frames & build a page table. Generated addrs can be divided into *page number* (page table index) & *page offset* within each page. For addr space 2^m & page size 2^n, page number has $m - n$ bits & page offset has n bits. Fixed size pages avoid ext. frag. **Memory Protection** is done with **protection bits** per page table entry. A *valid/invalid bit* indicates a legal or missing page.



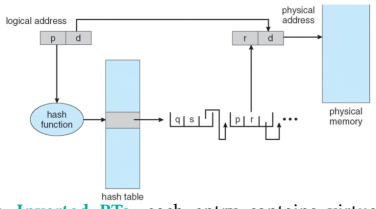
Page tables can be stored with a base reg **(PTBR)** & length reg **(PTLR)**, but this requires 2 mem accesses *(without paging, its 1)*. Instead, a hardware cache is used as associative memory. *1 for all cores.* To translate $\langle p, d \rangle$, if p is in associated reg, get frame number; otherwise get from page table. **Translation Lookaside Buffers (TLB)** are the cache, & need to be expensively flushed after context switch. Enable fast lookup of physical addr during addr trans from virtual addr to physical addr. *Per CPU core.* We can measure TLB performance with **effective access time (EAT)** $= (\epsilon + t)\alpha + (\epsilon + (l + 1)t)(1 - \alpha)$ where ϵ is time for assoc lookup, t is time for mem access, l is num of assoc regs, α is **cache hit ratio**.



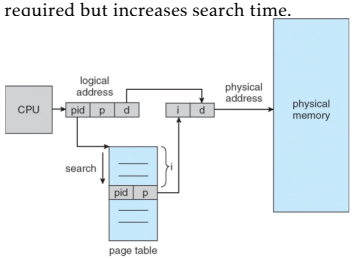
- **Heirarchical PTs**, we have multi-level PTs with many page numbers. This saves PT space.



- **Hashed PTs**, contains chain of elems leading to same loc. Search for match of virtual page num on the chain & extract frame number.



- **Inverted PTs**, each entry contains virtual addr of page stored in frame & info on the proc that owns the frame. Decreases mem required but increases search time.



7.2 Segmentation

Segmentation instead of paging provides a dynamic addr space & supports different protection kinds *(e.g. read-only)*, but allocation is harder & may suffer from ext. frag. Programmers are aware of segments but not pages.

7.3 Demand Paging

Demand Paging means bringing page into mem only when needed, causing lower IO load, less total mem, faster response time & support for more users.
Before bringing into mem, we check if valid mem (using valid bit), otherwise, **page fault**. First reference always leads to page fault. If the reference is valid mem, get an empty page frame, swap required page into frame from disk, update page table & restart faulting instruction; otherwise abort.
Now, EAT $= (1 - p)t + p\tau$. where t is mem access time & τ is page fault overhead + swap page in + swap page out + restart overhead.
Copy on Write (COW) shares pages between parent & child procs. Then, if a write occurs, the page is copied, optimising proc creation. Free pages are already zeroed. **Mem mapped files** map files into virtual addr space for simpler IO.

fork() must be optimised so it doesnt copy entire addr space when making a copy of parent process image. Can modify COW to give child its own page table with read-only pointers to parents pages.

An optimal **page replacement** strategy minimises page faults, prevents over-allocation & uses a dirty-bit to reduce overhead of page transfers. In general, *as the number of frames increases, the number of page faults decreases*.

- **FIFO PR** - replace the oldest page, using a *circular queue*. May suffer from **Belady's Anomaly** - more frames ⇒ more page faults, as FIFO doesn't consider frequency of page access.
- **LRU PR** - replaces least recently used page. Since using system clock is expensive to immplement (requires a search), we use a reference bit, set to 1 on access. Periodically set all frames to 0, & only replace frames with 0. This is not a *proper ordering*. A **second chance algortihm** combiners RR with reference bit, setting a bit to 0 after it is read to have a 1 in page replacement & performing 2 loops, meaning a free frame is more likely to be found.
- **LFU PR** - replaces least frequently used page, but this may replace a new page. Aging, or resetting counters must be used.
- **MFU PR** - replaces most frequently used page, but may remove page used heavily at start of proc.

Excessive paging activity causes low CPU utilisation (**thrashing**). Additionally, we want to keep pages requested close in space & time close physically (**locality of storage**) as programs request similar pages (**locality of reference**).

A **working set model** $W(t,w)$ is a set of pages referenced in time interval $(t-w,t)$. A **working set clock algorithm** adds to the clock algorithm keeping track of the page's current working set. If the ref. bit is 0: if age < working set window, move to next page; otherwise if the page is clean, replace, otherwise trigger a write back & continue. Size of working set may be chosen by observing **page fault recovery**: if the working set is too small, the page fault frequency will be high; if the working set is too large, the page fault frequency will be low, but the working set will be too large to fit in memory.

In **local** replacement, each proc gets a fixed alloc of physmem, OS needs to pick up changes in working set size. In **global** replacement, memory is shared dynamically between procs, initial mem alloced ∝ proc size, & **page fault frequency** is used to tune allocation.

8 Devices

Device independence separates *logical* device properties from its *physical* properties. This includes, **device type**, **device instance**, **device variations** (*data transfer unit*, *supported ops*, *sync/async ops*, *speed*, *shareable/single-user*, *err conditions*). On **character** devices, interrupt handler is called for each byte. On **block devices** it is called when entire block is transferred.

A **device driver** handles 1 device type. It implements read/write, accesses device regs, initials ops & schedules requests, handles errors.

The OS' **device independent layer** maps logical to phys devs, requests validation against dev characteristics, allocates dedicated devs, user access validation, buffering for performance, block size independence & error reporting. Devices can be **dedicated** (Simple policy, alloced for long periods for authed procs, e.g. *printer*), **shared** (e.g. *disks*) & **spooled** (run async by a *daemon*, no direct access allowed e.g. *printer*).

When **buffer** output, data transferred to OS buffer. Proc suspends on full buffer. When buffer input, OS reads ahead & reads are satisfied from buffer. Proc suspends on empty buffer. Buffered IO is used to *smooth peaks* in IO traffic, & caters for differences in data transfer between devs.

IO can be **programmed** (CPU is used to transfer between mem & dev), **interrupt driven** (CPU is interrupted when dev ready), or **Direct Memory Access (DMA)** (dev controller transfers directly to mem without CPU). A **blocking** IO call returns when the op completes. Proc is suspended, & IO appears instantaneous. A **non-blocking** IO call returns as much as available, providing application level polling for IO. In **async** IO, procs exec in parallel with IO ops, with no blocking in interface - the IO subsystem notifies completion with a callback, supporting wait/check. This is very flexible & efficient, but hard to use & can be less secure.

Loadable kernel modules provide dev drivers. Code laoded *on-demand*, dynamically linked. Kernel provides common IO syscall interface to the **virtual file system**. Devs are grouped into **dev classes** of similar function. **Major / minor IDs** identify devs in drivers. Same major ID means same driver. **Dev special files** in `/dev` provide device access. `ioctl` syscall supports special tasks. **Block IO subsystem** modularises block IO ops by placing common code in different layers. It **caches** data, or can bypass with *direct IO*. **Sockets** used to exchange info locally.

9 Disk Management

HDDs has phys addrs composed of *cylinder*, *surface* & *sector* addr. Modern disks use *logical sector addressing*. Before a drive is used, it is **formatted**, containing a **disk sector layout** (split between data, preamable & *error correction code (ECC)*), accouting for **cylinder skew** (time to rotate to next cylinder), & support for **interleave** (time to rotate to the next sector). A high level format also contains the **boot block**, list of **free blocks**, **root dir** & an empty **file system**.

The **seek time** t_s is time to move to cylinder. The **rotation time** t_r is time to rotate to sector. The **transfer time** t_t is time to read/write data. To transfer b bytes with N bytes per track & r revolutions per second, the latency is $t_r = 0.5r^{-1}$; $t_t = br^{-1}N^{-1}$; $T = t_s + t_r + t_t$.

Disk scheduling minimises access time by ordering requests based on *head position*. This can be **FCFS** (no ordering) or **SCAN** aka *elevator* (choose requests for shortest t_s in a *preferred direction*, changing dir when reaching top/bot cylinder; can result in long delays for extreme location).

IO requests placed in **req list**. Block dev drivers define req func, called by kernel.

SSDs consist of dies with blocks with pages. Page size smallest unit to be read / written; block size smallest unit to be ersaed. Faster & more bandwidth than HDD.

RAID (Redundant Array of Inexpensive Disks) uses an array of virtual drives acting as a single drive, storing distributed data over to allow parallel operations (**striping**), & uses redundant capacity to respond to failure. More disks means more *mean time to failure*. A **raid controller** manages it, can be software or hardware. *In general, fast concurrent reads, slow writes.* There are many RAID levels:

0	**Striping** - disks can seek & transfer concurrently, with no redundancy.
1	**Mirroring** - mirror data across disks, causing fast reads & slow writes. Failure recovery is easy, costing storage overhead.
5	**Block Level Distributed XOR** - distributed parity information. Potential for *some* write concurrency, but good efficiency / redundancy tradeoff. Reconstruction of failed disk is slow.

Mainmem can cache disk IO for faster operation. For this we need a **replacement policy**:

- **LRU** - Use a stack. Very efficient, but doesn't track popularity.
- **LFU** - Use a counter table. Blocks may be accessed many times in a short period, causing a misleading counter.
- **Frequency Based** - Divide LRU stack into *new* & *old*. Increment the counter if its not already in *new*. Replace block with lowest count in *old*. However, with this method, blocks age out too quickly.

10 File Systems

Responsible for *long term storage*, *sharing data*, *concurrent access* & *data organisation*. Stores info on file name, type, & attributes, & provides functions. File attributes include *name*, *type*, *organisation* (seq, rand, etc), *creator*, *disk vol*, *start addr*, *size used/alloced*, *owner*, *rights*, *usage info*.

File attrs accessed with `stat` syscall, returning info in a `struct stat`.

File size is variable, so we **dynamically man space** with **blocks**. Small blocksize wastes space for large files with high overhead for managing data, transfer time & seek time. High blocksize wastes space for small files with more memory required for buffer space.

File data may be placed at **contiguous addrs**, but this causes fragmentation when files are deleted, & performance issues when resized. Instead, **block linkage** has each block containing a ptr to the next. Although this speeds up insertion/deletion (with ptr modification), large blocksize causes internal frag, & small blocksize causes slow read time due to many seeks. Also wastes ptr space in each block.

A **block alloc table (BAT)** stores ptrs to blocks. Dir points to first block, & next is determined by chaining. A **file alloc table (FAT)** links file names to their phys addr. It is stored on disk, cached in mem. Fragmented disks can increaase the FAT size.

Each file contains ≥ 1 **index blocks**, which store a list of ptrs to data blocks. Index blocks may be chained nby reserving last feq entries for ptrs to more index blocks. This reduces seek time, & can be further optimised by caching in memory.

inodes are linux index blocks. On file open, OS opens **inode table**, stores on-disk inode structure with dev number, inode num, refcount, maj/min dev number, etc.

To keep track of free blocks, we can use a **free list** (linked list, low overhead, likely to create noncontiguous blocks) or **bitmap** (1 bit per block, quick to look for block in certain locations, slow overhead when searching entire bitmap).

A **fixed disk layout** contains a **boot block** (bootstrap code), **super block** (number of blocks/inodes, block size, max file size, etc), **inode/block bitmaps**, **data**. Most FS are **heirarchical**, with a *root dir* & *pathnames*. We can store **links** (hard - logical addr, soft - pathname), but these can cause problems removing files (hard links need a *link counter*) or when traversing (infinite loops). A **mount** operation combines multiple FS into a single namespace. Hard links may not be mounted. A **mount point** is a dir in the native FS that is assigned to the root of the mounted FS. **Mount tables** manage mounted dir, storing the loc of mount points & devs.

EXT2FS has 4KiB blocksize with 5% reserved for root. **ext inode** represents files and dirs, first 12 ptrs direct ptrs to datablocks. 13th ptr is an indirect ptr to a block of ptrs. 14th ptr is a double indirect ptr to a block of single indirect ptrs. 15th ptr is a triple indirect ptr to a block of double indirect ptrs.
Block groups: clusters of contiguous blocks. FS attempts to store related data in same block group, reducing seek time.

In general, *OS Layer Responsible* for dynamic prefetching of file blocks, as it is best positioned to calculate good heuristics.

11 Security

A **security policy** specifies *for what*, *for who* & *what kind* of access is granted. A **security mechanism** specifies how to implement it. Security can be **People security** e.g. social engineering; **Hardware security** needs *physical access* to hack; **Software security** about finding bugs.

Authentication verifies users, based on **possession** (keys) or **knowledge** (passwords). Passwords suffer from **turnover** (guessing, or reuse), and **dictionary attacks**. A hash of the pass is stored instead of the pass itself. A **rainbow table** is a precomputed table for reversing hash funcs. It can be used to crack pass. A **salt** is a random value that is added to the pass before hashing. This makes it harder to crack passes using rainbow tables.

Access control uses **principle of least privelage (PoLP)** where users are given min rights. **Protection domains** are a set of **access rights** defined as **objects** & **operations permitted**, represented in **access control lists (ACL)**, each elem is a list of rules per domain. Access is controlled by *object owner* **(discretionary)** or *system* **(mandatory)**.

Users are principals with unique ID. `root` (id=0) has full access. Each file/dir has **access rights** (R/W/X). Each proc has **real UID** (user who started proc), **effective UID** (access control checks), and **saved UID** (option for effective UID).

The **bell-lapaluda model** is **no read up** (read at its level or lower) & **no write down** (write at its level or higher), ensuring confidentiality not integrity. The **biba** model is **no write up** (write at its level or lower) & **no read down** (read at its level or higher), ensuring integrity not confidentiality.

12 Virtualisation

In **virtualisation**, a proc runs a full OS, providing security in isolation & legacy software support. A **virtual machine monitor (VMM)** partitions hardware resources to provide each proc with a VM, intercepting all instrs & emulating their exec (introducing overhead). We only need to trap & emulate sensitive instrs. A CPU is **virtualisable** if all sensitive instrs trap. A **hypervisor** is a hardware VMM, can be type 1 (*bare metal*) or 2 (*virtualization*). 1 is faster but needs hardware support, & 2 is more versatile.

Instead of emulation, **binary translation** can run the code faster, scanning each block & replacing sensitive instrs with hypercalls. We can handle unvirtualisable archs with **paravirtualisation** by binary translation. The **virtual machine interface (VMI)** is a set of standardised hypercalls that can be used by any OS.

A **phys map (PMAP)** maps phys addrs to machine addrs for a VM, stored in the hypervisor. **Shadow page tables** map virt addrs to machine addrs, stored in the guest OS, used by hardware MMU. On a miss, MMU searches in shadow PT, then PF happens in the VMM, where it looks for mapping in the guest OS' PT. If not found, a **true PF** occurs, forwarded to the guest OS. Otherwise, a **hidden PF** occurs, & it is added to shadow PT. VMM must sync guest & shadow PT. On a TLB miss, MMU searches in guest PT. If not found, **true PF**. Otherwise, MMU searches in PMAP, TLB is updated & instr reexeced. Otherwise, **hidden PF**.

For the hypervisor to reclaim mem from a VM, it allocates a **balloon driver** in the guest OS, which it can inflate to use up memory in the guest. The guest then swaps out pages under pressure. Hypervisor reclaims memory allocated by the ballon driver, & then deflates it.

Memory can be shared by mapping 2 phys addrs to a machine addr, by computing a hash of pages to find similar pages.