# 19. Complete VM Systems

## HOW TO BUILD A COMPLETE VM SYSTEM

What features are needed to realize a complete virtual memory system? How do they improve performance, increase security, or otherwise improve the system?

## VAX/VMS

- **the OS had to have mechanisms & policies that worked across huge range of systems** : run on a broad range of machines, including very inexpensive one to extremly high-end & powerful machines

- excellent software innovation that hides some of inherent hardware flaws of the architecture

### Memory Management Hardware

the system was a hybrid of paging and segmentation

## Address space

Lower-half : process space

- unique to each process
- first half (P0) : the user program is found, heap which grows downward
- second half (P1) : stack, grows upwards

Upper-half : system space (S)

- only half of it is used.
- Protected OS code and data reside here
- the OS is in this way shared across processes.

**Problem : incredibly small size of pages in the VAX hardware (512bytes)**

- Chosen for historical reasons
- make simple linear page tables excessively large

=> GOAL : VMS Should not overwhelm memory with page tables

## REDUCE the pressure page tables place on memory

**1. Segmenting the user address space into two**

- provides a page table for each of P0 and P1 per process
- no page-table space is needed for unsued protion of the address space between the stack & heap
- the base & bounds registers are used
  - base register : holds the address of the page table for that segment
  - bounds : holds its size (= the number of page-table entries)

**2. Placing user page tables (for P0 & P1 / 2 per process) in kernel virtual memory = S**

- when allocating / growing a page table, the kernel allocates space out of "its own virtual memory"(= segment S).

- virtual : if memory comes under severe pressure, the kernel can swap pages of these page tables out to disk => making physical

memory available for other uses.

- kernel : Placing page tables in kernel virtual memory (not user virtual memory) saves user virtual address space.

- address translation is even further complicated / but made faster by the hardware-managed TLBs
  - what PTBR saves
    - in normal case : **physical address** of the start of the page table.
    - in VAX : **a kernel virtual address** pointing to where the page table for P0/P1 is located.

    1. The hardware wants to look up the page-table entry (PTE) for the user virtual address in the user page table (for P0 or P1).
    2. The user page table **itself resides in kernel virtual memory**, so to access that PTE, the hardware must translate the kernel virtual address of the user page table entry.
    3. To do that translation, the hardware consults the **system page table**, which resides in physical memory, to map the kernel virtual address of the PTE to a physical address.
    4. With the physical address of the page table entry obtained, the hardware reads the PTE.
    5. Finally, using the PTE, the hardware computes the physical address of the original memory reference.

## The curse of generality

the OS is not likely to support any one installation very well
But it is no less real today, where Linux is expected to run well on every machine.

## A Real Address Space

**the code segment never begins at page 0**

- page 0 : is marked inaccessible, to provide support for detecting null-pointer access => invalid access / segfault

**the kernel virtual address space (its data structures & code) : part of each user address space**

- On a context switch, the OS
  - changes the P0 and P1 registers to point to the appropriate page tables of the soon-to-be-run process.
  - doesn't chage the S base and bound registers, and as a result the "same" kernel structures are mapped into each user address space. => "Shared" / kernel appears almost as a library to applications but a protected one.

the kernel is mapped into each address space : easier for the kernel

**protection bit in the page table**

system data and code are set to a higher level of protection than user data & code

## Page Replacement

**no reference bit** => VMS replacement algorithm must make do without hardware support for determining which pages are active

**you don't need a hardware reference bit to see which pages are in use in a system. => use protection bits**

1. mark all of the pages in the page table as inaccessible (but keep around the info as to which pages are really accessible by the process)
2. when a process accesses a page, check if the page really should be accessible => if so revert the page to its normal protections
3. when replacement, the OS can check pages remain marked

inaccessible => have not been recently used

- not be too aggressive in marking pages inaccessible => overhead would be too high
- not be too passive=> all pages will end up referenced => no good idea which page to evict

**memory hogs**
programs that use a lot of memory
=> make it hard for other programs to run

- most policies are susceptible to hogging
  /ex. LRU doesn't share memory fairly among process

To solve no reference bit & memory hogs
## => segmented FIFO replacement policy

- each process has a maximum number of pages it can keep in memory = resident set size (RSS)
- each of these pages is kept on a FIFO list
- when a process exceeds its RSS, the first-in page is evicted
- FIFO clearly does not need any support from the hardware => easy to implement

**pure FIFO doesn't work well => 2 second-chance lists**
: pages are placed before getting evicted from memory

- global clean-page free list / dirty-page list
- when a process P exceeds its RSS, a page is removed from its per-process FIFO
- if clean (not modified) => placed on the end of the clean-page list
- if dirty (modified) => placed on the end of the dirty-page list

- if another process Q needs a free page, it takes the first free page off of the global clean list.
- However, if the original process P faults on that page before it is reclaimed, P reclaims it from the free (or dirty) list => avoiding a

costly disk access.

**The bigger these global second-chance lists**
**=> the closer the segmented FIFO algorithm performs to LRU**

**with small pages, disk i/o during swapping could be highly inefficient**
disk do better with large transfers
=> to make swapping I/O more efficient => **clustering**
VMS groups large batches of pages together from the global dirty list
=> writes them to disk in one fell swoop => making them clean

Clustering is used in most modern systems, as the freedom to place pages anywhere within swap space lets the OS group pages
* you can place pages closer to each other in swap spaces
=> perform fewer & bigger writes => improve performance

## Other Neat Tricks : lazy optimization

### demand zeroing

- zeroing page before mapping into your address space : for security => costly. especially if the page doesn't get used by the process
- demand zeroing : the OS does very little work when the page is added to your address space

1. puts an entry in the page table that marks the page inaccessible
2. if the process then reads or writes the page, a trap into the OS takes place
3. when handling the trap, the OS notices that this is actually a demend-zero page (usually through some bits marked in the "reserved for OS" portion of the PTE)
4. the OS does the needed work of finding a physical page, zeroing it => mapping it into the process's address space

IF the process NEVER accesses the page, all such work is avoided

**copy-on-write**
when the OS needs to copy a page from one address space to another,
instead of copying it, it can map it into the target address space & mark it read-only in both address spaces.
=> a fast copy wihout acutually moving any data
*fork(), fork()->exec()

# Linux

- VM system must be flexible enough: runs effectively on systems as small & underpowered as phones to the most scalable multicore systems found in modern datacenters.

## The Linux Address Space

: consists of a user portion / kernel portion

upon a context switch
- user portion of the currently-running address space changes
- kernel portion : same across processes

a program running in user mode can't access kernel virtual pages
: only by trapping into the kernel and transitioning to privileged mode can such memory be accessed

## Kernel virtual addresses : logical / virtual

## Kernel logical address

- normal virtual address space of the kernel
- to get more memory of this type, kernel code merely needs to call kmalloc
- most kernel data structures live here : page tables, per-process kernel stacks, etc
- kernel logical memory cannot be swapped to disk

- direct paging between kernel logical addresses <-> the first portion of physical memory.
  - simple to translate => treated as if they are indeed physical
  - if a chunk of memory is contigous in kernel logical address space => so do in physical memory
  - suitable for operations which need contiguous physical memory to work correctly => I/O transfers via direct memory access

## Kernel virtual address

- to get memory of this type, kernel code calls a different allocator vmalloc, which returns a pointer to a virtually contiguous region of the desired size.
- usually not contiguous : map to non-contiguous physical memory => Not suitable for DMA
- easier to allocate => used for large buffers where finding a contiguous large chunk of physical memory would be challenging.
- enable the kernel to address more than (roughly) 1GB memory : disconnection from a strict one-to-one mapping to physical memory make this possible * but no problem with 64-bit linux

## Large Page Support

### Pros

- lower TLB miss rate (main),
- fewer page-table entires
- shorter TLB miss path : comes down to **fewer page table levels** and **fewer TLB entries needed**.
- allocation can be quite fast

**transparent huge page support**

the operating system automatically looks for opportunities to allocate huge pages wihtout requireing application modification.

**Cost**

- internal fragmentation
- swapping does not work well, sometimes greatly amplifying the amount of I/O a system does
- Overhead of allocation can also be bad (contiguous, large physical memory needed)

growing memory sizes demand that we consider large pages

## The page Cache

the linux page cache is unified, keeping pages in memory from 3 primary sources

- memory-mapped files
- file data & metadata from devices (usually accessed by read(), write())
- heap & stack pages that comprise each process = anonymous memory(no named file underneath of it, but rather swap space)

these entities are kept in a page cache hash table, allowing for quick lookup when said data is needed

**the page cache tracks if entries are clean / modified**

dirty data is periodically written to the backing store by background threads => ensuring that modified data eventually is written back to persistent storage

- periodically or if there are too many dirty data

**page replacement**

standard LRU replacement is effective, but can be subverted by certain common access patterns

=> keep 2 lists & divide memory between them

- when accessed for the firt time , a page is placed on one queue (inactive list)
- when it is re-referenced, the page is promoted to the other queue (active list)

- when replacement needs to take place, one is taken from the inactive list
- periodically moves pages from the bottom of the active list to the inactive list, keeping the active list to about 2/3 of the total page cache size
- manage these lists in an approximation of LRU order
- generally behabes quite a bit like LRU
- but notably handles the case where a cyclic large-file access occurs by confining the pages of that cyclic access to the inactive list. (pages are never re-referenced before)

### Memory Mapping

by calling mmap() on an already opened file descriptor, a process is returned a pointer to the beginning of a region of virtual memory where the contents of the file seem to be located.
By then using that pointer, a process can access any part of the file with a simple pointer dereference

accesses to parts of a memory-mapped file that have not yet been brought into memory trigger page faults, at which point the OS will page in the relevant data and make it accessible by updating the page table of the process accordingly (=demand paging)

## Security & Buffer Overflows

## buffer overflow attacks

find a bug in the target system which lets the attacker inject arbitrary data into the target's address space

**vulnerabilities arise because**

the developer assumes erroneously that an input will not be overly long
=> trustingly copies the input into a buffer
because input is in fact too long => overflows the buffer
=> overwriting memory of the target

```
int some_function(char *input) {
  char dest_buffer[100];
  strcpy(dest_buffer, input); // oops, unbounded copy!
}
```

many cases, it's not catastrophic.

However, malicious programmers can carefully craft the input that overflows the buffer so as to inject their own code into the targeted system, essentially allowing them to take it over and do their own bidding

**How to defence?**
prevent execution of any code found within certain regions of an address space (ex. NX bit)

> prevents execution from any page which has NXbit set in its corresponding page table entry

## Return-oriented programming(ROP)

there are lots of bits of code within any program's address space
=> attacker can overwrite the stack such that the return address in the currently executing function points to a desired malicious instruction, followed by a return instruction.

**How to defend?**
**in linux, address space layout randomization(ASLR)**
instead of placing code, stack, and the heap at **fixed** locations within the virtual address space,

the OS randomizes their placement, thus making it quite challenging to craft the intricate code sequence required to implement this class of attacks.
=> most attacks on vulnerable user programs will cause crashes but not be able to gain control of the running program

ASLR : for user-level programs / KASLR : for kernel level

# Meltdown & Spectre

because CPUs perform all sorts of crazy behind-the-scenes tricks to improve performance ex. speculative execution

**Speculative execution**

CPU guesses which instructions will soon be executed in the future
=> starts executing them ahead of time

- if the guesses are correct : the program runs faster
- if not : the CPU undoes their effects on architectural state (ex.registers) & tries again / this time going down the right path

problem : tends to leave traces of its execution in various parts of the system, such as processor caches, branch prdictors, etc
=> such state can make vulnerable the contents of memory, even memory that we thought was protected by the MMU

**To increase kernel protection : KPTI**

- remove as much of the kernel address space from each user process
- instead have a separate kernel page table for most kernel data (called kernel pagetable isolation, or KPTI)
- instead of mapping the kernel's code & data structures into each process, only the barest minumum is kept therein
- when switching into the kernel, a switch to the kernel page table is now needed
- improves security / but performance goes down (switching page tables is costly)
- KPTI doesn't solve all of teh security problems

what about just turn off the speculation?
=> system would run thousands of times slower