Virtual Memory: Demand Paging and Replacement

Virtual Memory Illustrated

- executable file
  - header
  - text
  - idata
  - wdata
  - symbol table, etc.
- virtual memory
  - text
  - data
  - BSS
  - user stack
  - kernel
- process segments
- virtual-to-physical translations
- physical memory
  - physical page frames
- backing storage
- pageout/eviction
- page fetch
Virtual Address Translation

**Example:** typical 32-bit architecture with 8KB pages.

Virtual address translation maps a *virtual page number* (VPN) to a physical *page frame number* (PFN): the rest is easy.

Deliver exception to OS if translation is not valid and accessible in requested mode.

Physical address:

\[
\begin{align*}
\text{PFN} + \text{offset} & \quad \text{physical address} \\
\text{VPN} \quad & \quad \text{virtual address}
\end{align*}
\]

Role of MMU Hardware and OS

VM address translation must be very cheap (on average).

- Every instruction includes one or two memory references.
  (including the reference to the instruction itself)

VM translation is supported in hardware by a *Memory Management Unit* or *MMU*.

- The addressing model is defined by the CPU architecture.
- The MMU itself is an integral part of the CPU.

The role of the OS is to install the virtual-physical mapping and intervene if the MMU reports that it cannot complete the translation.
The Translation Lookaside Buffer (TLB)

An on-chip hardware translation buffer (TB or TLB) caches recently used virtual-physical translations (ptes).

Alpha 21164: 48-entry fully associative TLB.

A CPU pipeline stage probes the TLB to complete over 99% of address translations in a single cycle.

Like other memory system caches, replacement of TLB entries is simple and controlled by hardware.

  e.g., Not Last Used

If a translation misses in the TLB, the entry must be fetched by accessing the page table(s) in memory.

  cost: 10-200 cycles

A View of the MMU and the TLB
Completing a VM Reference

The OS Directs the MMU

The OS controls the operation of the MMU to select:

1. the subset of possible virtual addresses that are valid for each process (the process virtual address space);
2. the physical translations for those virtual addresses;
3. the modes of permissible access to those virtual addresses; read/write/execute
4. the specific set of translations in effect at any instant. need rapid context switch from one address space to another

MMU completes a reference only if the OS “says it’s OK”. MMU raises an exception if the reference is “not OK”.

Alpha Page Tables (Forward Mapped)

offset at each level is determined by specific bits in VA

three-level page table (forward-mapped)

sparse 64-bit address space (43 bits in 21064 and 21164)

A Page Table Entry (PTE)

This is (roughly) what a MIPS/Nachos page table entry (pte) looks like.

valid bit: OS uses this bit to tell the MMU if the translation is valid.

write-enable: OS touches this to enable or disable write access for this mapping.

dirty bit: MMU sets this when a store is completed to the page (page is modified).

reference bit: MMU sets this when a reference is made through the mapping.
Paged Virtual Memory

Like the file system, the paging system manages physical memory as a page cache over a larger virtual store.

- Pages not resident in memory can be zero-filled or found somewhere on secondary storage.
- MMU and TLB handle references to resident pages.
- A reference to a non-resident page causes the MMU to raise a page fault exception to the OS kernel.
  - Page fault handler validates access and services the fault.
  - Returns by restarting the faulting instruction.
- Page faults are (mostly) transparent to the interrupted code.

Care and Feeding of TLBs

The OS kernel carries out its memory management functions by issuing privileged operations on the MMU.

**Choice 1:** OS maintains page tables examined by the MMU.

- MMU loads TLB autonomously on each TLB miss
- page table format is defined by the architecture
- OS loads page table bases and lengths into privileged memory management registers on each context switch.

**Choice 2:** OS controls the TLB directly.

- MMU raises exception if the needed pte is not in the TLB.
- Exception handler loads the missing pte by reading data structures in memory (software-loaded TLB).
Where Pages Come From

- Text
- Data
- BSS
- User stack
- Args/env
- Kernel

Initial references to user stack and BSS are satisfied by zero-fill on demand.

Modified (dirty) pages are pushed to backing store (swap) on eviction.

Paged-out pages are fetched from backing store when needed.

Fetches for clean text or data are typically fill-from-file.

Questions for Paged Virtual Memory

1. How do we prevent users from accessing protected data?
2. If a page is in memory, how do we find it?
   
   Address translation must be fast.

3. If a page is not in memory, how do we find it?
4. When is a page brought into memory?
5. If a page is brought into memory, where do we put it?
6. If a page is evicted from memory, where do we put it?
7. How do we decide which pages to evict from memory?
   
   Page replacement policy should minimize overall I/O.
Demand Paging and Page Faults

OS may leave some virtual-physical translations unspecified. mark the pte for a virtual page as invalid

If an unmapped page is referenced, the machine passes control to the kernel exception handler (page fault).

passes faulting virtual address and attempted access mode

Handler initializes a page frame, updates pte, and restarts.

If a disk access is required, the OS may switch to another process after initiating the I/O.

Page faults are delivered at IPL 0, just like a system call trap.

Fault handler executes in context of faulted process, blocks on a semaphore or condition variable awaiting I/O completion.

Issues for Paged Memory Management

The OS tries to minimize page fault costs incurred by all processes, balancing fairness, system throughput, etc.

(1) fetch policy: When are pages brought into memory?

prepaging: reduce page faults by bring pages in before needed
clustering: reduce seeks on backing storage

(2) replacement policy: How and when does the system select victim pages to be evicted/discard from memory?

(3) backing storage policy:

Where does the system store evicted pages?

When is the backing storage allocated?

When does the system write modified pages to backing store?
Four Kinds of Page Faults

The OS can respond to a page fault in any of four ways.

1. **invalid reference**: notify and/or kill the process. 
   - segmentation violation or protection violation
2. **zero fill**: map in a page initialized to an array of zeroes. 
   - e.g., first reference to a page of uninitialized data or stack
3. **fill from file**: fetch a page from a file 
   - e.g., executable text or initialized static data 
   - the page table or other data structure must give (file, offset)
4. **page in from backing store**
   - e.g., a private page previously evicted from the page cache

Mapped Files

With appropriate support, virtual memory is a useful basis for accessing file storage (vnodes).

- **bind file to a region of virtual memory with mmap syscall.** 
  - e.g., start address \( x \)
  - virtual address \( x + n \) maps to offset \( n \) of the file
- **several advantages over stream file access**
  - uniform access for files and memory (just use pointers)
  - performance: zero-copy reads and writes for low-overhead I/O
  - *but*: program has less control over data movement
  - style does not generalize to pipes, sockets, terminal I/O, etc.
Using File Mapping to Build a VAS

Memory-mapped files are used internally for demand-paged text and initialized static data.

BSS and user stack are "anonymous" segments.
1. no name outside the process
2. not sharable
3. destroyed on process exit

Mach-Derived VM Structures

One pmap (physical map) per virtual address space.

page table

system-wide
phys-virtual map

address space (task)

memory objects

pmap

pmap_enter()

pmap_remove()

putpage

getpage

start, len, prot

start, len, prot

start, len, prot

vm_map

lookup

enter

pmap_page_protect

pmap_clear_modify

pmap_is_modified

pmap_is_referenced

pmap_clear_reference

page cells (vm_page_t)
array indexed by PFN
Memory Objects

Memory objects “virtualize” VM backing storage policy.

- source and sink for pages triggered by faults …or OS eviction policy
- manage their own storage
- external pager has some control:
  - prefetch
  - prewrite
  - protect/enable
- can be shared via \texttt{vm\_map()}
  (Mach extended \texttt{mmap})

\begin{itemize}
  \item \texttt{object->putpage(page)}
  \item \texttt{object->getpage(offset, page, mode)}
\end{itemize}

\begin{center}
\begin{tikzpicture}
  \node (vm_object) {
    \texttt{vm\_object}
  };
  \node (vnode) at (vm_object -| 1.2, 0) {
    \texttt{vnode}
  };
  \node (swap) at (2, 1.5) {
    \texttt{swap pager}
  };
  \node (vnode) at (2, 0) {
    \texttt{vnode pager}
  };
  \node (extern) at (2, -1.5) {
    \texttt{extern pager}
  };
  \node (anonymous) at (vm_object -| 3, 0) {
    \texttt{anonymous VM}
  };
  \node (mapped) at (vm_object -| 3, 1.5) {
    \texttt{mapped files}
  };
  \node (DSM) at (vm_object -| 3, -1.5) {
    \texttt{DSM databases reliable VM etc.}
  };
  \draw[->] (vm_object) -- (swap);
  \draw[->] (vm_object) -- (vnode);
  \draw[->] (vm_object) -- (extern);
  \draw[->] (vm_object) -- (anonymous);
  \draw[->] (vm_object) -- (mapped);
  \draw[->] (vm_object) -- (DSM);
\end{tikzpicture}
\end{center}

\textbf{The Block/Page I/O Subsystem}

The VFS/memory object/pmap framework reduces VM and file access to the central issue:

- executable files
- data files
- anonymous paging files (swap files)
- reads on demand from file syscalls
- reads on demand from VM page faults
- writes on demand

To deliver good performance, we must manage system memory as an I/O cache of pages and blocks.
The Page Caching Problem

Each thread/process/job utters a stream of page references.

- *reference string*: e.g., “abcabcdabe..”

The OS tries to minimize the number of faults incurred.

- The set of pages (the *working set*) actively used by each job changes relatively slowly.
- Try to arrange for the *resident set* of pages for each active job to closely approximate its working set.

Replacement policy is the key.

- On each page fault, select a victim page to evict from memory; read the new page into the victim’s frame.
- Most systems try to approximate an LRU policy.

VM Page Cache

**HASH**(memory object/segment, logical block)

1. Pages in active use are mapped through the page table of one or more processes.

2. On a fault, the global object/offset hash table in kernel finds pages brought into memory by other processes.

3. Several page queues wind through the set of active frames, keeping track of usage.

4. Pages selected for eviction are removed from all page tables first.
Managing the VM Page Cache

Managing a VM page cache is similar to a file block cache, but with some new twists.

1. Pages are typically referenced by page table (`pmap`) entries.
   Must `pmap_page_protect` to invalidate before reusing the frame.

2. Reads and writes are implicit; the TLB hides them from the OS.
   How can we tell if a page is dirty?
   How can we tell if a page is referenced?

3. Cache manager must run policies periodically, sampling page state.
   Continuously push dirty pages to disk to “launder” them.
   Continuously check references to judge how “hot” each page is.
   Balance accuracy with sampling overhead.

The Paging Daemon

Most OS have one or more system processes responsible for implementing the VM page cache replacement policy.

- A daemon is an autonomous system process that periodically performs some housekeeping task.

The paging daemon prepares for page eviction before the need arises.

- Wake up when free memory becomes low.
- Clean dirty pages by pushing to backing store.
  `prewrite` or `pageout`
- Maintain ordered lists of eviction candidates.
- Decide how much memory to allocate to UBC, VM, etc.
LRU Approximations for Paging

Pure LRU and LFU are prohibitively expensive to implement.
- most references are hidden by the TLB
- OS typically sees less than 10% of all references
- can’t tweak your ordered page list on every reference

Most systems rely on an approximation to LRU for paging.
- periodically sample the reference bit on each page visit page and set reference bit to zero run the process for a while (the reference window) come back and check the bit again
- reorder the list of eviction candidates based on sampling

FIFO with Second Chance (Mach)

Idea: do simple FIFO replacement, but give pages a “second chance” to prove their value before they are replaced.
- Every frame is on one of three FIFO lists: active, inactive and free
- Page fault handler installs new pages on tail of active list.
- “Old” pages are moved to the tail of the inactive list. Paging daemon moves pages from head of active list to tail of inactive list when demands for free frames is high.
  Clear the refbit and protect the inactive page to “monitor” it.
- Pages on the inactive list get a “second chance”.
  If referenced while inactive, reactivate to the tail of active list.
Illustrating FIFO-2C

Active list

Inactive list

Free list

Restock inactive list by pulling pages from the head of the active list: knock off the reference bit and inactivate.

Inactive list scan:
1. Page on inactive list has been referenced? Return to tail of active list (reactivation).
2. Page at head of inactive list has not been referenced? `pmap_page_protect` and place on tail of free list.
3. Dirty page on inactive list? Push and return to inactive list tail.

Consume frames from the head of the free list.

If free shrinks below threshold, kick the paging daemon to start a scan.

Paging daemon scans a few times per second, even if not needed to restock free list.

Viewing Memory as a Unified I/O Cache

A key role of the I/O system is to manage the page/block cache for performance and reliability.

- tracking cache contents and managing page/block sharing
- choreographing movement to/from external storage
- balancing competing uses of memory

Modern systems attempt to balance memory usage between the VM system and the file cache.

- Grow the file cache for file-intensive workloads.
- Grow the VM page cache for memory-intensive workloads.
- Support a consistent view of files across different style of access.

`unified buffer cache`
**Pros and Cons of Paged Virtual Memory**

Demand paging gives the OS flexibility to manage memory...

- programs may run with pages missing
  - unused or “cold” pages do not consume real memory
  - improves degree of multiprogramming
- program size is not limited by physical memory
  - program size may grow (e.g., stack and heap)

…but VM takes control away from the application.

- With traditional interfaces, the application cannot tell how much memory it has or how much a given reference costs.
- Fetching pages on demand may force the application to incur I/O stalls for many of its references.

**More Issues for VM Paging**

1. synchronizing shared pages
2. clustered reads/writes from backing store, and prefetching
3. adapting replacement strategies (e.g., switch to MRU)
4. trading off memory between the file (UBC) and VM caches
5. trading off memory usage among processes
6. parameterizing the paging daemon:
   - Keep the paging devices fully utilized if pages are to be pushed, but don’t swamp the paging device.
   - Balance LRU accuracy with reactivation overhead.
Shadow Objects and Copy-on-Write

Operating systems spend a lot of their time copying data.
- particularly Unix operating systems, e.g., `fork()`
- cross-address space copies are common and expensive

Idea: defer big copy operations as long as possible, and hope they can be avoided completely.
- create a new shadow object backed by an existing object
- shared pages are mapped readonly in participating spaces
  - read faults are satisfied from the original object (typically)
  - write faults trap to the kernel
    - make a (real) copy of the faulted page
    - install it in the shadow object with writes enabled

A Copy-on-Write Mapping

Warning: this is a fictional diagram intended to be representative only; any similarity to any specific system is purely coincidental.