Chapter VIII
Virtual Memory

Jehan-François Pâris
jfparis@uh.edu
Chapter overview

- Basics
  - Address translation
  - On-demand fetch
- Page table organization
- Page replacement policies
- Virtual memory tuning
Basics
Virtual memory

- Combines two big ideas
  - **Non-contiguous memory allocation:** processes are allocated page frames scattered all over the main memory
  - **On-demand fetch:** Process pages are brought in main memory when they are accessed for the first time

- **MMU takes care of almost everything**
Main memory

- Divided into fixed-size page frames
  - Allocation units
  - Sizes are powers of 2 (512B, 1KB, 2KB, 4KB)
  - Properly aligned
  - Numbered 0, 1, 2, ...
Process address space

- Divided into fixed-size *pages*
  - Same sizes as page frames
  - Properly aligned
  - Also numbered 0, 1, 2, …
The mapping

- Will allocate non-contiguous page frames to the pages of a process
The mapping

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
The mapping

- Assuming 1KB pages and page frames

<table>
<thead>
<tr>
<th>Virtual Addresses</th>
<th>Physical Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 1,023</td>
<td>0 to 1,023</td>
</tr>
<tr>
<td>1,024 to 2,047</td>
<td>4,096 to 5,119</td>
</tr>
<tr>
<td>2,048 to 3,071</td>
<td>2,048 to 3,071</td>
</tr>
</tbody>
</table>
The mapping

- Observing that $2^{10} = 1000000000$ in binary
- We will write 0-0 for ten zeroes and 1-1 for ten ones

<table>
<thead>
<tr>
<th>Virtual Addresses</th>
<th>Physical Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0-0 to 000 1-1</td>
<td>000 0-0 to 0001-1</td>
</tr>
<tr>
<td>001 0-0 to 001 1-1</td>
<td>100 0-0 to 100 1-1</td>
</tr>
<tr>
<td>010 0-0 to 010 1-1</td>
<td>010 0-0 to 010 1-1</td>
</tr>
</tbody>
</table>
The mapping

- The ten least significant bits of the address do not change

<table>
<thead>
<tr>
<th>Virtual Addresses</th>
<th>Physical Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0-0 to 000 1-1</td>
<td>000 0-0 to 000 1-1</td>
</tr>
<tr>
<td>001 0-0 to 001 1-1</td>
<td>100 0-0 to 100 1-1</td>
</tr>
<tr>
<td>010 0-0 to 010 1-1</td>
<td>010 0-0 to 010 1-1</td>
</tr>
</tbody>
</table>
The mapping

- Must only map page numbers into page frame numbers

<table>
<thead>
<tr>
<th>Page number</th>
<th>Page frame number</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>100</td>
</tr>
<tr>
<td>010</td>
<td>010</td>
</tr>
</tbody>
</table>
The mapping

- Same mapping in decimal

<table>
<thead>
<tr>
<th>Page number</th>
<th>Page frame number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
The mapping

- Since page numbers are always in sequence, they are redundant

<table>
<thead>
<tr>
<th>Page number</th>
<th>Page frame number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
The algorithm

- Assume page size $= 2^p$
- Chop $p$ least significant bits from virtual address to obtain the page number
- Use page number to find corresponding page frame number in page table
- Append $p$ least significant bits from virtual address to page frame number to get physical address
Realization

Virtual Address

Physical Address

Page No

Offset

5

897

Frame No

Offset

(unchanged)

1

3

5

7

PAGE TABLE

PAGE TABLE
The offset

- Offset contains all bits that remain unchanged through the address translation process
- Function of page size

<table>
<thead>
<tr>
<th>Page size</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB</td>
<td>10 bits</td>
</tr>
<tr>
<td>2 KB</td>
<td>11 bits</td>
</tr>
<tr>
<td>4 KB</td>
<td>12 bits</td>
</tr>
</tbody>
</table>
The page number

- Contains other bits of virtual address
- With old **32-bit addresses**

<table>
<thead>
<tr>
<th>Page size</th>
<th>Offset</th>
<th>Page number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB</td>
<td>10 bits</td>
<td>22 bits</td>
</tr>
<tr>
<td>2KB</td>
<td>11 bits</td>
<td>21 bits</td>
</tr>
<tr>
<td>4KB</td>
<td>12 bits</td>
<td>20 bits</td>
</tr>
</tbody>
</table>
With the newer 64 bit addresses

- Current processor limitations allow for 48 address lines
  - Can address $2^{48}$ bytes = 256 Terabytes

<table>
<thead>
<tr>
<th>Page size</th>
<th>Offset</th>
<th>Page number</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>12 bits</td>
<td>36 bits</td>
</tr>
</tbody>
</table>
Windows x64 virtual addresses

- Restricted to 256 TB (48-bit addresses)
  - Lower 128 TB are available as private address space for user processes
  - Upper 128 TB are system space

Maximum process address space is $2^{47}$ bytes, that is, 0.00076 percent of the theoretical limit of $2^{64}$ bytes.
Windows x86 virtual addresses

- 32 bit addresses allow us to access 4GB
- By default
  - Lower 2 GB are available as private address space for user processes
  - Upper 2 GB are system space
- But
  - Can give up to 3GB to user processes
  - Complex extension mechanism allowing x86 systems to use more than 4 GB of RAM
Internal fragmentation

- Each process now occupies an integer number of pages
- Actual process space is not a round number
  - Last page of a process is rarely full
- On the average, half a page is wasted
  - Not a big issue
  - Internal fragmentation
On-demand fetch (I)

- Most processes terminate without having accessed their whole address space
  - "Code handling rare error conditions, . . ."
- Other processes go to multiple phases during which they access different parts of their address space
  - "Compilers"
On-demand fetch (II)

- VM systems do not fetch whole address space of a process when it is brought into memory.
- They fetch individual pages on demand when they get accessed the first time.
  - Page miss or page fault
- When memory is full, they expel from memory pages that are not currently in use.
On-demand fetch (III)

- The pages of a process that are not in main memory reside on disk
  - In the *executable file* for the program being run for the pages in the code segment
  - In a special *swap area* for the data pages that were expelled from main memory
On-demand fetch (IV)
On-demand fetch (V)

- When a process tries to access data that are not present in main memory
  - MMU hardware detects that the page is *missing* and causes an *interrupt*
  - Interrupt wakes up *page fault handler*
  - Page fault handler puts process in *blocked state* and brings missing page in main memory
Advantages

- VM systems use main memory more efficiently than other memory management schemes
  - Give to each process *more or less what it needs*
- Process sizes are not limited by the size of main memory
  - Greatly simplifies program organization
Sole disadvantage

- Bringing pages from disk is a relatively slow operation
  - Takes milliseconds while memory accesses take nanoseconds
    - Ten thousand times to hundred thousand times slower
The cost of a page fault

Let
- $T_m$ be the main memory access time
- $T_d$ the disk access time
- $f$ the page fault rate
- $T_a$ the average access time of the VM

We have
- $T_a = (1 - f)T_m + f(T_m + T_d) = T_m + fT_d$
Example

- Assume $T_m = 70 \text{ ns}$ and $T_d = 7 \text{ ms}$

<table>
<thead>
<tr>
<th>$f$</th>
<th>$T_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-3}$</td>
<td>$= 70\text{ns} + 7\text{ms}/10^3 = 7,070\text{ns}$</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>$= 70\text{ns} + 7\text{ms}/10^4 = 770\text{ns}$</td>
</tr>
<tr>
<td>$10^{-5}$</td>
<td>$= 70\text{ns} + 7\text{ms}/10^5 = 140\text{ns}$</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>$= 70\text{ns} + 7\text{ms}/10^6 = 77\text{ns}$</td>
</tr>
</tbody>
</table>
Replacing the disk by an SSD

Assume $T_m = 70\ ns$ and $T_{SSD} = 70\ \mu s$

<table>
<thead>
<tr>
<th>$f$</th>
<th>$T_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-3}$</td>
<td>$= 70\text{ns} + 70\mu\text{s}/10^3 = 140\text{ns}$</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>$= 70\text{ns} + 70\mu\text{s}/10^4 = 77\text{ns}$</td>
</tr>
<tr>
<td>$10^{-5}$</td>
<td>$= 70\text{ns} + 70\mu\text{s}/10^5 = 70.7\text{ns}$</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>$= 70\text{ns} + 70\mu\text{s}/10^6 = 70.07\text{ns}$</td>
</tr>
</tbody>
</table>
Conclusion

- Virtual memory works best when page fault rate is less than a page fault per 100,000 instructions
  - Because page faults are very costly
Locality principle (I)

- A process that would access its pages in a totally unpredictable fashion would perform very poorly in a VM system unless all its pages are in main memory
Locality principle (II)

- Process $P$ accesses randomly a very large array
  - $n$ pages
- If $m$ of these $n$ pages are in main memory, the page fault frequency of the process will be $(n - m)/n$

- Must switch to another algorithm
Locality principle (III)

- Fortunately for us most programs obey the locality principle
  - They access at any time a small fraction of their address space
    - \textit{Spatial locality}
  - They tend to reference again the pages they have recently referenced
    - \textit{Temporal locality}
Tuning considerations

- In order to achieve an acceptable performance, a VM system must ensure that each process has in main memory all the pages it is currently referencing.

- When this is not the case, the system performance will quickly collapse.
Page Table Representations
Page table entries

- A page table entry (PTE) contains
  - A *page frame number*
  - Several *special bits*
- Assuming 64-bit addresses, all fit into eight bytes
The special bits (I)

- **Present bit/Valid bit:**
  - 1 if page is in main memory,
  - 0 otherwise

- **Missing bit:**
  - 1 if page is in *not* main memory,
  - 0 otherwise
The special bits (II)

- **Dirty bit:**
  - 1 if page has been modified since it was brought into main memory,
  - 0 otherwise
- A *dirty* page must be saved in the process swap area on disk before being expelled from main memory
- A *clean* page can be immediately expelled
The special bits (III)

- **Page-referenced bit:**
  - 1 if page has been recently *accessed*,
  - 0 otherwise

- Not present on many computers
  - Can be *simulated* in software
Where to store page tables

- Use a three-level approach
- Store parts of page table
  - In *high speed registers* located in the MMU: the *translation lookaside buffer* (TLB) (good solution)
  - In *main memory* (bad solution)
  - On *disk* (ugly solution)
The translation look aside buffer

- Small high-speed memory
  - Contains fixed number of PTEs
  - Content-addressable memory
    - Entries include page frame number and page number

| Page number | Page frame number | Bits |
TLB misses

- When a PTE cannot be found in the TLB, a **TLB miss** is said to occur.
- TLB misses can be handled:
  - By the computer firmware:
    - Cost of miss is one extra memory access.
  - By the OS kernel:
    - Cost of miss is two context switches.
Performance implications

- When TLB misses are handled by the firmware, they are very cheap
  - A TLB hit rate of 99% is very good:
    - Average access cost will be
      \[ T_a = 0.99 \times T_m + 0.01 \times 2 \times T_m = 1.01 \times T_m \]

- *Not true* if TLB misses are handled by the kernel
TLB coverage issues (I)

- TLBs have remained fairly small:
  - Sometimes just a few hundred entries
  - To remain fast

- Intel Skylake have two-level TLBs
  - \( L_1 \) can hold 64 PTEs
  - \( L_2 \) can hold 1536 (128×12) PTEs
TLB coverage issues (II)

- Together they can hold 1600 PTEs
  - Will cover a bit less than 1.6K×4KB, between 6 and 7MB of main memory
- Processes with very large working sets can incur too many TLB misses
  - Will affect system performance
Linear page tables (I)

- PTs are too large to be stored in main memory
  - Store PT in virtual memory (VMS solution)
    - Worked well for 32-bit architectures
  - Very large page tables need more than 2 levels
    - 3 levels on MIPS R3000
Linear page tables (II)
Linear page tables (III)

- Assuming a page size of 4KB,
  - Each page of virtual memory requires 4 bytes of physical memory
  - Each PT maps 4GB of virtual addresses
  - A PT will occupy 4MB
  - Storing these 4MB in virtual memory will require 4KB of physical memory
Multi-level page tables (I)

- PT is divided into
  - A primary index that always remains in main memory
  - Secondary indexes or subindexes that can be expelled from main memory
Multi-level page tables (II)
Multi-level page tables (III)

- Especially suited for a page size of 4 KB and 32-bit virtual addresses
- Will allocate
  - 10 bits of the address for the first level (primary index),
  - 10 bits for the second level (the secondary indexes, and
  - 12 bits for the offset.
- Primary index and all secondary indexes will all have $2^{10}$ entries and will all occupy 4KB
ARM virtual address translation

VIRTUAL ADDRESS

Page Directory

Page Table(s)

Frame

Frame No

Offset

TTBR

Addr

10 bits 10 bits 12 bits

(unchanged)
Multi-level page tables (IV)

- What if we want larger address space?
- Linux uses three-level page tables
  - One *Page Global Directory* (PGD):
    - Occupies one page frame
  - Multiple *Page Middle Directories* (PMD)
  - Multiple *Page Tables*
- Actual sizes are implementation dependant
Multi-level page tables (V)

64-bit address

<table>
<thead>
<tr>
<th>Not used</th>
<th>PGD</th>
<th>PMD</th>
<th>PT</th>
<th>offset</th>
</tr>
</thead>
</table>

- Page Directory Pointer
- Page Directory index
- Page Table index
- Byte offset
x86 virtual address translation

32-bit address

PDPT index | Page Directory index | Page Table index | Byte offset

PDPT is Page Directory Pointer Table
specifies one of four possible page directories
The bad news

- More difficult to have 4KB pages and 4KB directories
  - With 64-bit addresses, can only put 512 PTEs per page
  - Could only address
    \[2^9 \times 2^9 \times 2^9 \times 2^{12} \text{B} = 2^{39} \text{B} = 512 \text{ GB}\]
X64 virtual address translation

64-bit address

<table>
<thead>
<tr>
<th>“Reserved”</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Map Index (level 4)</td>
<td>Page Directory Pointers Index (level 3)</td>
<td>Page Directory Index (level 2)</td>
<td>Page Table Index (level 1)</td>
<td>Byte offset</td>
<td></td>
</tr>
</tbody>
</table>
X64 virtual address translation

64-bit address

<table>
<thead>
<tr>
<th>“Reserved”</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Map Index (level 4)</td>
<td>Page Directory Pointers Index (level 3)</td>
<td>Page Directory Index (level 2)</td>
<td>Page Table Index (level 1)</td>
<td>Byte offset</td>
<td></td>
</tr>
</tbody>
</table>
Hashed page tables (I)

- Only contain pages that are in main memory
  - PTs are much smaller
- Also known as *inverted page tables*
Hashed page table (II)

PN = page number
PFN = page frame number
Discussion

- We have much fewer PTEs than with regular page tables
  - *Whole PT can reside in main memory*

- Hashed/inverted PTEs occupy **three times** more space than regular PTEs
  - Must store page number, page frame number and a pointer to next entry
Selecting the right page size

- Increasing the page size
  - Increases the length of the offset
  - Decreases the length of the page number
  - Reduces the size of page tables
    - Fewer entries
  - Increases internal fragmentation

- 4KB seems to be a good choice
Page replacement policies
Their function

- Selecting which page to expel from main memory when
  - Memory is full
  - Must bring in a new page
Objectives

- A good page replacement policy should
  - Select the right page to expel (victim)
  - Have a reasonable run-time overhead

- First objective was more important when memory was extremely expensive
- Second objective has been more important since the mid-eighties
Classification

- Four classes of page replacement policies
  - Fixed-size local policies
  - Global policies
  - Variable-size local policies
  - Hybrid policies (part global and part local)
Fixed-size local policies

- Assign to each process a *fixed number* of page frames
- Whenever a process has used all its page frames, it will have to expel one of its own pages from main memory before bringing in a new page
- Two policies:
  - Local FIFO
  - Local LRU
Local FIFO

- Expels the page that has been in main memory for the longest period of time

- **Very easy to implement:**
  - Can organize the pages frames into a queue

- **Very poor policy:**
  - Does not take into account how the page was used
Local LRU

- Expels the page that has not referenced for the longest period of time
  - LRU stands for *Least Recently Used*

- *Best fixed-size replacement policy*

- *Has an extremely high overhead:*
  - Must keep track of all page accesses
  - Never used for VM
Global policies

- Treat whole memory as a *single pool* of page frames
- Whenever a page fault happens and memory is full, expel a page from any process
  - *Processes “steal” page frames from each other*
- Many policies
Global FIFO and global LRU

- Global variants of local FIFO and local LRU
  - Same advantages and disadvantages
MULTICS Clock policy (I)

- Organizes page frames in a circular list
- When a page fault occurs, policy looks at next frame in list
  - if PR bit = 0, the page is expelled and the page frame receives the incoming page
  - if PR bit = 1, the PR bit is reset and policy looks at next page in list
MULTICS Clock policy

step 1: reset PR bit

step 2: reset PR bit

step 3: expel this page
Algorithm

Frame *clock(Frame *lastVictim) {
    Frame *hand;
    int notFound = 1;
    hand = lastVictim->next;
    do {
        if (hand->PR_Bit == 1) {
            hand->PR_Bit = 0; hand = hand->next;
        } else
            notFound = 0; // found!
    } while notFound;
    return hand;
} // clock

You should not memorize this algorithm, but should try to understand it.
BSD Implementation (I)

- Designed for architectures lacking a PR bit
- Uses the valid bit to simulate the PR bit
  - Resets valid bit to zero instead of resetting PR bit to zero
  - When page is referenced again an interrupt occurs and the kernel sets the valid bit back to one
- Requires two context switches
BSD Implementation (II)

step 1: mark page invalid

step 2: mark page invalid

step 3: expel this page
A first problem

- When memory is overused, hand of clock moves too fast to find pages to be expelled
  - Too many resets
  - Too many context switches
- Berkeley UNIX limited CPU overhead of policy to 10% of CPU time
  - No more than 300 page scans/second
Evolution of the policy

- Policy now runs with much more physical memory
- Hand now moves too slowly
- By the late 80’s a **two-hand policy** was introduced:
  - First hand resets simulated PR bit
  - Second hand follows first at constant angle and expels all pages whose PR bit = 0
The two-hand policy

- Expels
- Resets simulated PR bit
FIFO with second chance (I)

- Used in the Mach 2.5 kernel
- Stores pages from all processes in a single FIFO pool
  - The active queue
- Expelled pages go to the end of a single inactive queue where they wait before being actually expelled from main memory
  - Can be rescued if they were expelled but still active
- FIFO can make bad decisions
FIFO with second chance (II)

Global pool of page frames
FIFO
(Active Queue)

Expelled pages

Reclaimed pages

Inactive Queue

Disk
FIFO with second chance (IV)

- Implementation dependent
  - Presence/absence of a page referenced bit

- **Without a PR bit**
  - Pages in the inactive queue are not mapped into any address space
  - First access requires *two context switches* and returns the page to the active queue
Without a PR bit

Global pool of page frames
FIFO
(Active Queue)

Inactive Queue

Pages are reclaimed at first access

Expelled pages are marked invalid

Disk
FIFO with second chance (V)

- **With a PR bit,**
  - Pages sent to the inactive queue
    - Remain valid
    - Have their PR bit reset to zero
  - First access turns bit on
  - Page will return to the active queue when it would otherwise be expelled
    - *No additional context switch overhead*
With a PR bit

Global pool of page frames  
FIFO  
(Active Queue)

Inactive Queue

Expelled pages have PR bit reset to 0

Disk

Reclaim all pages with PR bit = 1
Variable-space local policies

- **Working set policy** let each process keep into main memory all pages it had accessed during its last $T$ references.
- Provided excellent performance.
- Was never implemented due to its very high cost.
- Influenced research efforts to design better page replacement policies.
  - *No need to discuss them*
Hybrid policies

- Window page replacement policy combines aspects of local and global policies

- Solution adopted by
  - VMS in the late 70s
  - Windows ten years later
    - Started with Windows NT
    - Mainstream since Windows XP
Windows policy (I)

- Allocates to each process a *private partition* that it manages using a FIFO policy.
- Pages expelled by the FIFO policy are put at the end of a large global LRU queue from which they can be reclaimed
  - Predates by several years use of same solution by Mach
Windows policy (II)

- Process P0 resident set of pages
- Process P1 resident set of pages
- Process P2 resident set of pages

Expelled pages → Global LRU queue → Disk

Reclaimed pages
Major advantage

- Supports real-time applications
  - Most VM systems are poorly suited to real-time applications
    - Unpredictable paging delays
  - Policy allows VM to allocate to a process enough page frames to hold all its pages
    - Process will never experience a page fault
Major disadvantage

- Hard to decide how many frames to allocate to each process
  - Allocating too many frames leaves not enough space for the global LRU queue
    - Page fault rate will become closer to that of a global FIFO policy
  - Not allocating enough frames would cause too many reclaims and too many context switches
Windows solution (I)

- Each process is allocated a **minimum** and **maximum working set size**
- Processes start with their minimum allocation of frames
- If the main memory is **not full**, the VM manager allows processes to grow up to their maximum allocation
Windows solution (II)

- As the main memory become full, the VM manager starts trimming the working sets of processes.
- Processes that exhibit a lot of paging can regain some of their lost frames if enough frames remain available.
Virtual Memory Tuning
The problem

- With virtual memory
  - Most processes run without having all their pages in main memory
  - Can have more processes in main memory
    - Reduces CPU idle times
    - Increases the system throughput
- How far can we go?
Effect on throughput

System Throughput

Zone I

Zone II

Zone III

Number of Processes in Memory
(Multiprogramming Level)
Zone I

- **Optimal Behavior:**
  - Throughput increases with multiprogramming level
  - Little or no impact of page faults on system performance
Zone II

- **Unstable Behavior:**
  - Page fault impact on throughput increases
  - Any surge of demand may move the system performance to zone III

Think of a freeway just below its saturation point: Cars still move fast but any incident can cause a slowdown
Zone III

- **Thrashing:**
  - Active pages are constantly expelled from main memory to be brought back again and again
  - Paging device becomes the bottleneck

Think of a freeway above its saturation point: Cars barely move
Preventing thrashing

- Have enough main memory
- Start suspending processes when paging rate starts increasing

**Old empirical rule:**
- Keep utilization of paging disk below 60 percent