Virtual Memory

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>
previously: SRAM ⇔ DRAM
next: DRAM ⇔ HDD, SSD, etc.
i.e., memory as a “cache” for disk
main goals:

1. maximize memory throughput
2. maximize memory utilization
3. provide address space consistency & memory protection to processes
throughput = # bytes per second
- depends on access latencies (DRAM, HDD) and “hit rate”
- improve by minimizing disk accesses
utilization = fraction of allocated memory that contains “user” data (aka payload)
- reduced by storing metadata in memory
- affected by alignment, block size, etc.
address space consistency → provide a uniform “view” of memory to each process
address space consistency $\rightarrow$ provide a uniform “view” of memory to each process
memory protection → prevent processes from directly accessing each other’s address space
memory protection → prevent processes from directly accessing each other’s address space
i.e., every process should be provided with a managed, \textit{virtualized} address space
“memory addresses”: what are they, really?
“physical” address: (byte) index into DRAM
int glob = 0xDEADBEEE;

main() {
    fork();
    glob += 1;
}

(gdb) set detach-on-fork off
(gdb) break main
Breakpoint 1 at 0x400508: file memtest.c, line 7.
(gdb) run
Breakpoint 1, main () at memtest.c:7
7    fork();
(gdb) next
[New process 7450]
8    glob += 1;
(gdb) print &glob
$1 = (int *) 0x6008d4
(gdb) next
9 }
(gdb) print /x glob
$2 = 0xdeadbeef
(gdb) inferior 2
[Switching to inferior 2 [process 7450]
#0 0x0000000310acac49d in __libc_fork ()
131     pid = ARCH_FORK ();
(gdb) finish
Run till exit from #0 in __libc_fork ()
8    glob += 1;
(gdb) print /x glob
$4 = 0xdeadbeee
(gdb) print &glob
$5 = (int *) 0x6008d4
instructions executed by the CPU do not refer directly to physical addresses!
processes reference *virtual* addresses, the CPU relays virtual address requests to the *memory management unit* (MMU), which are *translated* to physical addresses
CPU → virtual address

MMU:
- address translation unit
  - physical address

Main Memory

“swap” space

(note: cache not shown)
the size of virtual memory is determined by the virtual address width

the physical address width is determined by the amount of installed physical memory
e.g., given 48-bit virtual address, 8GB installed DRAM
- 48-bits → 256TB virtual space
- 8GB → 33-bit physical address
virtual address space

$P_0$

$n$

virtual address

$2^n - 1$

$P_1$

$v_n$

virtual address

$2^n - 1$

$P_2$

$n$

virtual address

$2^n - 1$

(global) physical address space

$2^m - 1$

physical address

$m$
essential problem: map request for a virtual address $\rightarrow$ physical address

… and this must be **FAST!** (happens on every memory access)
both hardware/software are involved:

- MMU (hw) handles simple and fast operations (e.g., table lookups)
- Kernel (sw) handles complex tasks (e.g., eviction policy)
§Virtual Memory Implementations
keep in mind goals:

1. maximize memory *throughput*
2. maximize memory *utilization*
3. provide *address space consistency* & *memory protection* to processes
1. simple relocation

- per-process relocation address is loaded by kernel on every context switch
1. simple relocation

- incorporate a *limit* register to provide memory protection

```
CPU
\[ \text{VA}: N \]
\[ \text{PA}: N+B \]

MMU
limit reg. \[ L \]
relocation reg. \[ B \]

Main Memory
\[ N \]
\[ B+L \]
```

assert \((0 \leq N \leq L)\)
pros:

- simple & fast!
- provides protection
but: available memory for mapping depends on value of base address
i.e., address spaces are not consistent!
also: all of a process *below the address limit* must be loaded in memory

i.e., memory may be *vastly under-utilized*
2. segmentation

- partition virtual address space into *multiple logical segments*

- individually map them onto physical memory with relocation registers
virtual address has form `seg#:offset`
MMU Segment Table

<table>
<thead>
<tr>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
<td>$L_0$</td>
</tr>
<tr>
<td>$B_1$</td>
<td>$L_1$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>$L_2$</td>
</tr>
<tr>
<td>$B_3$</td>
<td>$L_3$</td>
</tr>
</tbody>
</table>

VA: seg#:offset

PA: offset + $B_2$

assert (offset ≤ $L_2$)
- implemented as MMU registers
- part of kernel-maintained, per-process metadata (aka “process control block”)
- re-populated on each context switch
pros:

- still very *fast*
  - translation = register access & addition
- memory *protection* via limits
- segmented addresses improve *consistency*
what about utilization?
simple relocation:

![Diagram of simple relocation]

segmentation:

![Diagram of segmentation]
- variable segment sizes → memory fragmentation
- fragmentation potentially lowers utilization
- can fix through compaction, but expensive!
3. paging

- partition virtual and physical address spaces into *uniformly sized* pages

- only map pages onto physical memory that contain required data
- pages boundaries are not aligned to segments!
- instead, aligned to multiples of page size
- minimum mapping granularity = page
- not all of a given segment need be mapped
new mapping problem:
- given virtual address, decompose into virtual page number & virtual page offset
- map VPN $\rightarrow$ physical page number
Given page size $= 2^p$ bytes

VA: $p$

| virtual page number | virtual page offset |

PA: $p$

| physical page number | physical page offset |
VA: virtual page number \hspace{1cm} virtual page offset

address translation

PA: physical page number \hspace{1cm} physical page offset
VA: virtual page number \hspace{0.5cm} virtual page offset

translation structure: page table

\begin{array}{|c|c|}
\hline
\text{valid} & \text{PPN} \\
\hline
\end{array}

PA: physical page number \hspace{0.5cm} physical page offset

if invalid, page is not mapped
page table entries (PTEs) typically contain additional metadata, e.g.:

- dirty (modified) bit
- access bits (shared or kernel-owned pages may be read-only or inaccessible)
e.g., 32-bit virtual address, 
4KB \(2^{12}\) pages, 
4-byte PTEs;

- size of page table?
e.g., 32-bit virtual address,
4KB \( (2^{12}) \) pages,
4-byte PTEs;

- \# pages = \( 2^{(32-12)} = 2^{20} = 1M \)
- page table size = \( 1M \times 4 \text{ bytes} = 4\text{MB} \)
4MB is much too large to fit in the MMU — insufficient registers and SRAM!

Page table resides in **main memory**
The translation process (aka page table walk) is performed by hardware (MMU).

To facilitate this, the kernel populates a page table base register on context switches; the kernel also manages a process’s page table over its lifetime.
translation: *hit*

1. VA: N
2. **page table walk**
3. PA: N'
4. *data*
translation: *miss*

1. VA: N
2. page table walk
3. page fault
4. transfer control to kernel
5. data transfer
6. PTE update
7. VA: N (retry)
8. data
9. PA: N'
10. data

CPU

Address Translator (part of MMU)

Main Memory

Page Table

kernel

Disk (swap space)
kernel decides where to place page, and what to evict (if memory is full)

- e.g., using LRU replacement policy
this system enables on-demand paging
i.e., an active process need only be partly in memory (load rest from swap dynamically)
but if working set (of active processes) exceeds available memory, we may have **swap thrashing**
integration with caches?
Q: do caches use physical or virtual addresses for lookups?
Virtual address based Cache

Process A

Virtual Address Space

L

\[ 0 \]

[Cache]

Address | Data
---|---
L | X
M | Y
N | Z

Process B

Virtual Address Space

\[ Z \]

\[ N \]

\[ M \]

\[ 0 \]

“synonym” problem

ambiguous!
Physical address based Cache

Process A
Virtual Address Space

CPU

Cache

Address | Data
---|---
S | X
Q | Y
R | Z

Physical Memory

X
Z
Y

Process B
Virtual Address Space

M
L
0

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
Q: do caches use physical or virtual addresses for lookups?
A: caches typically use *physical* addresses
saved by hardware: 

the *Translation Lookaside Buffer* (TLB) — a cache used solely for VPN→PPN lookups
TLB + Page table
(exercise for reader: revise earlier translation diagrams!)
virtual address

n-1  p  p-1  0

virtual page number (VPN)  page offset

valid  tag  physical page number (PPN)

physical address

valid  tag  data

byte offset

TLB

Cache

TLB Hit

Cache Hit
but!

TLB mappings are *process specific* — requires flush & reload on context switch

- some architectures store PID (aka “virtual space” ID) in TLB
and another problem:

- TLB caches a few thousand mappings
- vs. *millions* of virtual pages per process!
we can improve TLB hit rate by reducing the number of pages ... by increasing the size of each page
compute # pages for 32-bit memory for:

- 1KB, 512KB, 4MB pages
  - \(2^{32} \div 2^{10} = 2^{22} = 4M\) pages
  - \(2^{32} \div 2^{19} = 2^{13} = 8K\) pages
  - \(2^{32} \div 2^{22} = 2^{10} = 1K\) pages
  (not bad!)
Process A

Virtual Memory

Physical Memory

Virtual Memory

Process B

lots of wasted space!
increasing page size results in increased internal fragmentation and lower utilization
i.e., TLB effectiveness needs to be balanced against memory utilization!
so what about 64-bit systems?

$2^{64} = 16 \text{ ExbiByte address space} \approx 4 \text{ billion x 4GB}$
most modern implementations support a max of $2^{48}$ (256TB) addressable space
page table size?

- # pages   = $2^{48} \div 2^{12} = 2^{36}$
- PTE size = 8 bytes (64 bits)
- PT size   = $2^{36} \times 8 = 2^{39}$ bytes
  = 512GB
512GB
(just for the virtual memory *mapping* structure)
(and we need *one per process*)
(these things aren’t going to fit in memory)
instead, use *multi-level* page tables:

- split an address translation into two (or more) separate table lookups

- unused parts of the table don’t need to be in memory!
“toy” memory system
- 8 bit addresses
- 32-byte pages (5-bit page offset)

Page Table

all 8 PTEs must be in memory at all times
“toy” memory system
- 8 bit addresses
- 32-byte pages (5-bit page offset)

```
1 1 0 1 1 0 1 0
```

page offset

page “directory”
all unmapped; don’t need in memory!
“toy” memory system
- 8 bit addresses
- 32-byte pages (5-bit page offset)
Intel Architecture Memory Management

http://www.intel.com/products/processor/manuals/

(Software Developer’s Manual Volume 3A)
PROTECTED-MODE MEMORY MANAGEMENT

- The segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment).
- The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a linear address in the processor's linear address space.

If paging is not used, the linear address space of the processor is mapped directly into the physical address space of the processor.

The physical address space is defined as the range of addresses that the processor can generate on its address bus.

Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of "virtualizing" the linear address space is needed. This virtualization of the linear address space is handled through the processor's paging mechanism.

Paging supports a "virtual memory" environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk.
Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.

Segment descriptors

Stack
Code
Data
Data
Data
Data
Data

Segment Registers

CS
SS
DS
ES
FS
GS

Segment Descriptors

Linear Address Space (or Physical Memory)

Access | Limit | Base Address

Access | Limit | Base Address

Access | Limit | Base Address

Access | Limit | Base Address

Access | Limit | Base Address

Access | Limit | Base Address

Access | Limit | Base Address

Access | Limit | Base Address
PROTECTED-MODE MEMORY MANAGEMENT

3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3). A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

Figure 3-2. Flat Model

Figure 3-3. Protected Flat Model

“Flat” model
Table 4-1 illustrates the key differences between the three paging modes. Because they are used only if IA32_EFER.LME = 0, 32-bit paging and PAE paging is used only in legacy protected mode. Because legacy protected mode cannot produce linear addresses larger than 32 bits, 32-bit paging and PAE paging translate 32-bit linear addresses.

Because it is used only if IA32_EFER.LME = 1, IA-32e paging is used only in IA-32e mode. (In fact, it is the use of IA-32e paging that defines IA-32e mode.) IA-32e mode has two sub-modes:

- **Compatibility mode.** This mode uses only 32-bit linear addresses. IA-32e paging treats bits 47:32 of such an address as all 0.
- **64-bit mode.** While this mode produces 64-bit linear addresses, the processor ensures that bits 63:47 of such an address are identical. IA-32e paging does not use bits 63:48 of such addresses.

### Table 4-1. Properties of Different Paging Modes

<table>
<thead>
<tr>
<th>Paging Mode</th>
<th>CR0.PG</th>
<th>CR4.PAE</th>
<th>LME in IA32_EFER</th>
<th>Linear-Address Width</th>
<th>Physical-Address Width&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Page Size(s)</th>
<th>Supports Execute-Disable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>32</td>
<td>32</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>32-bit</td>
<td>1</td>
<td>0</td>
<td>0&lt;sup&gt;2&lt;/sup&gt;</td>
<td>32</td>
<td>Up to 40&lt;sup&gt;3&lt;/sup&gt;</td>
<td>4-KByte</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4-MByte</td>
<td></td>
</tr>
<tr>
<td>PAE</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>32</td>
<td>Up to 52</td>
<td>4-KByte</td>
<td>Yes&lt;sup&gt;5&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2-MByte</td>
<td></td>
</tr>
<tr>
<td>IA-32e</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>48</td>
<td>Up to 52</td>
<td>4-KByte</td>
<td>Yes&lt;sup&gt;5&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2-MByte</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1-GByte</td>
<td></td>
</tr>
</tbody>
</table>

### Notes:

1. The physical-address width is always bounded by MAXPHYADDR; see Section 4.1.4.
2. The processor ensures that IA32_EFER.LME must be 0 if CR0.PG = 1 and CR4.PAE = 0.
3. 32-bit paging supports physical-address widths of more than 32 bits only for 4-MByte pages and only if the PSE-36 mechanism is supported; see Section 4.1.4 and Section 4.3.
4. 4-MByte pages are used with 32-bit paging only if CR4.PSE = 1; see Section 4.3.
5. Execute-disable access rights are applied only if IA32_EFER.NXE = 1; see Section 4.6.
6. Not all processors that support IA-32e paging support 1-GByte pages; see Section 4.1.4.

**Paging modes**
IA-32 paging (4KB pages)
IA-32 paging (4MB pages)
IA-32e paging (4KB pages)
The following items describe the IA-32e paging process in more detail as well as how the page size is determined.

- A 4-KByte naturally aligned PML4 table is located at the physical address specified in bits 51:12 of CR3 (see Table 4-12). A PML4 table comprises 512 64-bit entries (PML4Es). A PML4E is selected using the physical address defined as follows:
  - Bits 51:12 are from CR3.
  - Bits 11:3 are bits 47:39 of the linear address.
  - Bits 2:0 are all 0.
  Because a PML4E is identified using bits 47:39 of the linear address, it controls access to a 512-GByte region of the linear-address space.

- A 4-KByte naturally aligned page-directory-pointer table is located at the physical address specified in bits 51:12 of the PML4E (see Table 4-14). A page-directory-pointer table comprises 512 64-bit entries (PDPTEs). A PDPTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PML4E.

IA-32e paging (1GB pages)