Operating Systems: Memory Management

Shankar

April 14, 2015
Overview
Segmentation
Non-demand Paging
Demand Paging
Paged-Segmentation
Sparse Virtual Space
IA-32: x86 – 32-bit
Process Space → System Space

- **Process space** (aka *virtual* space)
  - address space of its (machine code) program
  - structure: code, stack, data, heap, ...
  - attributes: read, read/write, execute, ...

- **System space**
  - hierarchy of caches, main memory, disk, ...
  - small, fast → large, slow

- Virtual space mapped to physical space **at run time**

<table>
<thead>
<tr>
<th>Instruction execution</th>
<th>Virtual address</th>
<th>hw/sw mapper</th>
<th>location in cache/memory/disk</th>
</tr>
</thead>
</table>

- if location in disk
 (hw traps → sw moves content to memory → resume instruction)
Benefits of Run-time Mapping

- Program’s start address is 0 (in each segment)
- Kernel enforces attributes of virtual address
  - memory protection
  - efficient debugging
- Virtual space can be larger than physical memory
  - memory-mapped files, ...
- Only the active part of virtual space need be in physical memory
  - faster starting up
  - higher degree of multi-programming
  - better utilization of memory, IO devices, ...
- Efficient sharing of memory between processes/kernel
  - avoid IO transfers between user and kernel space
- Efficient system virtualization
  - guest operating systems
Some Basics

- Processor can only access locations in memory
  - faster for caches, slower for main memory
- Move more (less) frequently accessed stuff to fast (slow) mem
  - need to maintain info on usage, dirty/clean, ...
- Suspend some programs if movement overhead too high
  - thrashing, low hit rate
- Efficient management of physical space
  - partition space into blocks (physical pages, disk sectors, ...)
  - reduce fragmentation of space
- Need caching in processor to achieve low overhead of mapping
  - TLB (translation-lookaside buffer): physical or virtual
- ...

Overview
Way to run a program that doesn’t fit in physical memory

Program image structured in parts:
- common
- phase 1
- phase 2
- ...

Initially, common and phase 1 parts in physical memory

The last bit of phase $i$, for $i = 1, 2, \cdots$
- loads in phase $i + 1$, overwriting phase $i$
- jumps to start of phase $i + 1$

No OS intervention

No virtual address space: instructions generate physical addresses
Outline

Overview
Segmentation
Non-demand Paging
Demand Paging
Paged-Segmentation
Sparse Virtual Space
IA-32: x86 – 32-bit
## Segmentation: Overview

- **Virtual address:**
  - Segment: `seg#`
  - Offset: `offset`
  - `m-bit` x `n-bit`
- **seg#’s:** 0, ··· , $2^m - 1$
- **seg max size:** $2^n$ bytes

- **Virtual address is generated by an instruction**
  - **Example:** `load 2:0x030, reg1`  
    // put data at 2:0x030 into `reg1`

- **Physical address:**
  - `addr`

- **Program space consists of segments**
  - **Example:**
    - `seg0`: code; `seg2`: data; `seg5`: stack; ···
  - **Subset of:** `{0, ··· , $2^m - 1}`
  - **Need not be consecutive**
  - **Each segment has a size:** (in 0, ··· , $2^n$)
  - **Each segment has access attributes:** (r, r/w, ···)
A segment is mapped completely in one physical memory chunk.

Assume all segments of a (non-suspended) process are mapped,
- hence, total virtual space of processes $\leq$ physical memory size.

Segments can grow if there is adjacent free physical memory,
- allocate physical memory in out-of-bounds exception handler.

External fragmentation: wasted space between segments.
- arises as processes enter/leave.
- chunks of memory too small to use.
- can fix by compaction, but expensive.
Each process has a **segment table** in memory  // part of PCB

<table>
<thead>
<tr>
<th>seg#</th>
<th>valid</th>
<th>base</th>
<th>size</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>

- Created when process is loaded
- Row for every $j$ in $\{0, \cdots, 2^m - 1\}$  // descriptor of $j$
  - valid (1 bit): 1 iff process has segment $j$
  - base: physical address of (start of) segment $j$
  - size of segment $j$
  - access: allowed access to segment $j$, eg, r, r/w, ...
- Layout fixed by hardware
- For partial mapping of segment set, may want status field also.
Processor has a STBR

When a process is dispatched:
  \[ \text{STBR} \leftarrow \text{phys addr of the process’s segment table} \]

In user mode, processor uses segment table pointed to by STBR
  - refer to this table as “STBR.table”

  \[
  \text{virtual address} \quad \rightarrow \quad \text{physical address} \quad \text{STBR.table}[sn].base + \text{offset}
  \]

In kernel mode, STBR not used, ie, physical addr = virtual addr
  - eg, add/delte zero msb’s as needed
Address Translation w/o Cacheing

- Instruction generates virtual addr $segno$ $offset$ in user mode

- Processor does following
  - $sd \leftarrow$ STBR.table[$segno$]
  - if $sd.valid = 0$ $\rightarrow$ invalid-segment exception
  - if $sd.size < offset$ $\rightarrow$ out-of-bounds exception
  - if $sd.access$ violated $\rightarrow$ invalid-access exception
  - $phyaddr \leftarrow$ $sd.base + offset$
  - access physical memory location $phyaddr$

- Upon exception
  - processor enters kernel mode
  - executes exception handler

- Requires 2 memory accesses for every virtual address
  - Overcome with caches in the processor
TLB Cache: Translation Lookaside Buffer

- TLB cache: holds a portion of the current segment table

<table>
<thead>
<tr>
<th>valid (1 bit)</th>
<th>sn (segment #)</th>
<th>base</th>
<th>size</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- for an entry \( x \) with valid bit set:
  
  \[ \text{STBR.table}[x.sn].valid = 1 \quad \text{and} \quad x.[base, size, access] = \text{STBR.table}[x.sn] \]

- Hw (set) associatively searches on valid and sn fields

- OS clears all valid bits when a process is dispatched
  
  - unless TLB also has a pid field // associatively searched

- TLB’s valid bit is not the same segment table’s valid bit
Processor generates virtual address *segno:offset*

If TLB does not have *valid-segno* entry // TLB miss
- if \( \text{STBR.table}[\text{segno}].valid = 0 \) → invalid-segment exception
- create a *valid-segno* TLB entry from \( \text{STBR.table}[\text{segno}] \)
  - if TLB was full, overwrite an entry

Let \( x \) be *valid-segno* TLB entry // TLB hit
- if access invalid wrt \( x.access \) → invalid-access exception
- if \( offset > x.size \) → out-of-bounds exception
- \( \text{physaddr} \leftarrow x.base + offset \)
  - access physical memory location \( \text{physaddr} \)

Cache replacement policy: fast, crude if needed
- FIFO, random, LRU approx, ...
TLB is not the only cache in the processor

Physical memory cache: holds a portion of physical memory
- memory is divided into (small-sized) blocks
- valid (1 bit)  \(pa\) (phys addr)  \(data\) (of block \(pa\))
  
  | \(\cdot\) | \(\cdot\) | \(\cdot\) |
  
  (set) associatively searched on valid and \(pa\) fields

Virtual memory cache: holds a portion of virtual memory

Cache access order: virtual mem $\rightarrow$ TLB $\rightarrow$ phys mem

Multiple levels of caches
Overview
Segmentation
Non-demand Paging
Demand Paging
Paged-Segmentation
Sparse Virtual Space
IA-32: x86 – 32-bit
Virtual space is linear (not segmented)
Virtual space and physical memory divided into pages

<table>
<thead>
<tr>
<th>Virtual addr:</th>
<th>virtual page #</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>m-bit</td>
<td>n-bit</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical addr:</th>
<th>physical page #</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-bit</td>
<td>n-bit</td>
<td></td>
</tr>
</tbody>
</table>

virtual page # range: 0, \cdots, 2^m - 1
physical page # range: 0, \cdots, 2^p - 1
page size: 2^n bytes
Program space consists of virtual page ranges
- eg, 0–1000: code; 2000–4000: data; 7000–9000: stack; ...
- each page has access attributes (r, r/w, ...)

Virtual pages are mapped to physical pages

All virtual pages of an active process are mapped // for now

Paging increases physical memory utilization

**Internal fragmentation**: wasted space inside page
- usually in the last page of a program “segment”
- less wasteful than external fragmentation (between segments)
Each process has a page table in memory // part of PCB

<table>
<thead>
<tr>
<th>vpn</th>
<th>valid</th>
<th>ppn</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td>(virtual page #)</td>
<td>(1 bit)</td>
<td>(physical page)</td>
<td>(r, r/w, ...)</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Created when process is loaded

Row for every $j$ in $\{0, \cdots, 2^m - 1\}$ // descriptor of $j$

- **valid** (1 bit): 1 iff process has page $j$
- **ppn**: physical page # where $j$ is mapped (if valid = 1)
- **access**: allowed access to page $j$

Layout fixed by hardware
Processor has a PTBR

When a process is dispatched:

\[ \text{PTBR} \leftarrow \text{phys addr of the process’s page table} \]

In user mode, processor uses page table pointed to by PTBR

\[ \text{refer to this table as “PTBR.table”} \]

virtual address \[ \begin{array}{c} \text{vpn} \\ \text{offset} \end{array} \] \[ \rightarrow \] physical address \[ \begin{array}{c} \text{PTBR.table[vpn].ppn} \\ \text{offset} \end{array} \]

In kernel mode, paging optional

\[ \text{no paging: physical addr = virtual addr} \]
Address Translation w/o Cacheing

- Instruction generates virtual addr \[\text{vpn} \ | \ \text{offset}\] in user mode

- Processor does following
  - \(pd \leftarrow \text{PTBR.table}[\text{vpn}]\)
  - if \(pd\).valid = 0 \(\rightarrow\) invalid-page exception
  - if \(pd\).access violated \(\rightarrow\) invalid-access exception
  - \(\text{phyaddr} \leftarrow \text{sd.ppn} \ | \ \text{offset}\)
  - access physical memory location \(\text{phyaddr}\)

- Upon exception
  - processor enters kernel mode
  - executes exception handler

- Requires 2 memory accesses for every virtual address
  - Overcome with caches in the processor
TLB cache: holds a portion of the current page table

<table>
<thead>
<tr>
<th>valid</th>
<th>vpn (virtual page #)</th>
<th>ppn (phys page #)</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
</tr>
</tbody>
</table>

for an entry \( x \) with valid bit set:

\[
\text{PTBR.table[}x.vpn\].valid = 1 \quad \text{and} \quad x.[ppn, access] \text{ equals PTBR.table[}x.vpn.[ppn, access]]
\]

Hw (set) associatively searches on valid and vpn fields

OS clears all valid bits when a process is dispatched
  - unless TLB also has a pid field
    // associatively searched
Address Translation with TLB

- Processor generates virtual address \( vpn \) \( offset \)

- If TLB does not have valid-\( vpn \) entry \( // \) TLB miss
  - if \( PTBR.table[vpn].valid = 0 \) \( \rightarrow \) invalid-page exception
  - create a valid-\( vpn \) TLB entry from \( PTBR.table[vpn] \)
    - if TLB was full, overwrite an entry

Let \( x \) be valid-\( vpn \) TLB entry \( // \) TLB hit

- if access invalid wrt \( x.access \) \( \rightarrow \) invalid-access exception
- \( physaddr \leftarrow x.ppn \) \( offset \)
- access physical memory location \( physaddr \)

- Interaction with physical-memory and virtual-memory caches
Outline

Overview
Segmentation
Non-demand Paging
Demand Paging
Paged-Segmentation
Sparse Virtual Space
IA-32: x86 – 32-bit
Paging with virtual pages mapped to physical memory as needed
Hence can exploit spatial locality in programs to provide
- virtual space greater than physical memory size
- more processes simultaneously active

Page fault exception
- generated by access to virtual page not in memory (but in disk)
  - hw **undoes** any partial effects of page-faulting instruction
  - faulting virtual addr saved in a specified place

Page fault handler:
- bring in virtual page to physical memory
- if no physical page is free, evict a virtual page in memory
- if the evicted virtual page is dirty, write it back to disk
Page replacement policy:
- which virtual page to evict when free physical page needed
- goal: minimize page faults (to maximize cpu utilization)
- maintain "usage" info for physical pages
  - eg, order of last allocation  // sw-only
  - eg, order of least recently accessed  // hw-sw
  - per process (local) or across all processes (global)
- maintain dirty bit per physical page in memory  // hw-sw
  - set iff its vpage has changed from on-disk copy
- evict page based on usage and dirty
  - local or global
Page Table

- Page table per process

<table>
<thead>
<tr>
<th>vpn</th>
<th>valid</th>
<th>ppn</th>
<th>access</th>
<th>dirty</th>
<th>hw-usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Row for every possible virtual page number $j$ // as before
  - `valid` (1 bit): 1 iff page $j$ in memory
    - 0: page $j$ in disk or does not exist
  - `ppn`: physical page number // as before
  - `access`: r, w, x, ...
    // as before
  - `dirty` (1 bit): 1 iff page $j$ has changed from disk
  - `hw-usage`: hw-accessed usage info, if any
    - typically, `referenced bit`: set to 1 when hw accesses $j$
Additional state maintained by OS

- For each process: valid vpage numbers and their disk locations
  - can be kept in page table, if hw allows
- List of free physical pages
- Sw-accessed usage info for allocated physical pages
  - eg, order of last allocation
  - if info per process, can be kept in page table (if hw allows)
- For each shared phy page: links to vpage entries (in page tables)
- **Core map**: phy pages $\rightarrow$ global usage info, shared vpages
PTBR and TLB

Processor uses PTBR in user mode

\[
\begin{array}{c|c|c}
\text{vpn} & \text{offset} & \rightarrow \\
\end{array} \quad \begin{array}{c|c|c|c|c|c}
& & \text{PTBR.table[vpn].ppn} & \text{offset} \\
\end{array}
\]

Processor has TLB

\[
\begin{array}{c|c|c|c|c|c}
\text{valid} & \text{vpn} & \text{ppn} & \text{access} & \text{dirty} & \text{hw-usage} \\
\hline
\cdot & & & & & \\
\cdot & & & & & \\
\cdot & & & & & \\
\end{array}
\]

- \textit{hw-usage}: if present in page table
- when entry \( x \) is evicted

\[
\text{PTBR.table}[x.\text{vpn}].\text{dirty/hw-usage} \leftarrow x.\text{dirty/hw-usage}
\]
Address Translation (with TLB)

- Processor generates virtual address $\text{vpn} \ | \ \text{offset}$ in user mode

- If TLB does not have valid-$\text{vpn}$ entry // TLB miss
  - let $pd$ be PTBR.table[$\text{vpn}$]
  - if $pd$.valid = 0 $\rightarrow$ page-fault/invalid-page exception
  - create a valid-$\text{vpn}$ TLB entry from $pd$
    - if TLB was full, replace an entry and overwrite it

Let $x$ be valid-$\text{vpn}$ TLB entry // TLB hit

- if access invalid wrt $x$.access $\rightarrow$ invalid-access exception
- $\text{physaddr} \leftarrow x$.ppn $| \ \text{offset}$
- access physical memory location $\text{physaddr}$
Handling page-fault for virtual page \( vpn \) of process \( P \):

- save state of \( P \) and move its pcb to “pagewait” queue
- get a free physical page \( pp \)  
  // may block
- read vpage \( P.vpn \) from disk into \( pp \)
- update \( P \)’s page table
- move \( P \)’s pcb to ready queue

Generating free pages 
// executed asynchronously as needed

- select an allocated page \( x \) to free  
  // from usage info
- mark \( x \)’s vpage(s) as invalid  
  // in page table(s)
- if \( x \) is dirty, write \( x \)’s vpage(s) back to disk  
  // page cleaning
- mark \( x \) as free

Page cleaning can also be done asynchronously as needed:

- select an allocated dirty page, write to disk, mark as clean
Page Replacement: Local vs Global

- **Local**
  - each process $P$ is given a set of physical pages. free page in this set can only go to $P$
  - so per-process page usage info suffices
  - OS periodically adjusts the phy page allocation
    - maintains a page-fault “rate” for each process
      - eg, inverse of cpu-time between faults
      - not useful: inverse of wallclock-time between faults
    - moves pages from low-rate processes to high-rate processes

- **Global**
  - a free page can go to any process // need global usage info
  - no separate phy page allocation policy
Objective
- min # page faults given reference seqv and set of phy pages
- implementable with acceptably low overhead

Can be applied locally or globally

Optimal: evict the page that is used farthest in the future
- unrealizable, except in very constrained situations
- useful as a standard for evaluating other policies

LRU: evict the page not accessed for the longest time in past
- works well because the future is usually like the past
- usage info: order pages by latest reference
- hw needs to update the order on each access: impractical
- **FIFO**: evict the page that was mapped in earliest
  - usage info: order pages by mapping time
  - update only when page is swapped in:

- **FIFO + Second Chance** (also called “Clock”)
  - like FIFO but also requires referenced bit $R$ (hw-usage)
  - if head page has $R = 0$, free page (clean if needed)
  - if head page has $R = 1$, zero $R$ and move page to tail
  - “clock”: hand moving over pages ordered in a circle
    - at each step, current page is made free or its $R$ zeroed

- **LRU approximation**
Clock using referenced bit $R$ and dirty bit $D$
- if $R, D = 0, 0$: free page
- if $R, D = 0, 1$: clean page, $R, D \leftarrow 0, 0$
- if $R, D = 1, 0$: $R, D \leftarrow 0, 0$
- if $R, D = 1, 1$: $R, D \leftarrow 0, 1$

Clock using referenced counter $R$ (> 1 bit)
- hw increments $R$ at each access unless already at max
- if $R = 0$: free page
- if $R > 0$: $R \leftarrow R - 1$
- closer to LRU

Clock using referenced counter $R$ (> 1 bit) and dirty bit $D$
Belady’s Anomaly

- In FIFO, more memory may cause more page faults
  
  \[
  1 \ 2 \ 3 \ 4 \ 1 \ 2 \ 5 \ 1 \ 2 \ 3 \ 4 \ 5 \ \text{9 faults.}
  \]

  \[
  \begin{array}{cccc}
  \text{f1} & 1 & 4 & 5 \\
  \text{f2} & 2 & 1 & - \\
  \text{f3} & 3 & 2 & - \\
  \text{f4} & 4 & - & 3 \\
  \end{array}
  \]

  \[
  1 \ 2 \ 3 \ 4 \ 1 \ 2 \ 5 \ 1 \ 2 \ 3 \ 4 \ 5 \ \text{10 faults}
  \]

  \[
  \begin{array}{cccc}
  \text{f1} & 1 & - & 5 \\
  \text{f2} & 2 & - & 1 \\
  \text{f3} & 3 & - & 4 \\
  \text{f4} & 4 & - & 3 \\
  \end{array}
  \]

- LRU (and most others) do not suffer from Belady’s anomaly
Working set of a process

- A process exhibits locality of page accesses over “short” times.
- **Working set** at time $t$: virtual pages accessed in $[t - \delta, t]$
  - $\delta \approx$ few seconds$/10^5$ instructions
  - typical: working set stays fixed for some time, then changes

- Let $W(t)$ be the size of the working set at time $t$
  - typical: $W(t)$ is much less than the total # virtual pages

![Graph showing # instr executed between successive page faults vs. # phys pages]
Consider $P$ physical pages, $N$ identical cpu-intensive processes
- each process gets $P/N$ physical pages

As $N$ increases to $P/W$, cpu throughput increases

As $N$ increases beyond $P/W$, cpu throughput drops
  // goodput drops even more

To recover from thrashing, suspend some processes
Upon excessive thrashing, which process to kill/suspend
- one with most memory, one asking for memory, youngest process, ...
- probably not oldest process: init (root); gdm (user); ...

malloc() fails: out of virtual memory, not physical memory

Non-pageable pages
- Page table? Could swap out if process is suspended
- Current/active page tables
- Pages that an IO adapter is going to usek (DMA)
  - permanently allocated, eg, display
  - lock the page
- To improve heap locality: program mallocs a large chunk and divides it itself
- How large should a page be? Page size vs IO size? Page the Kernel?
- Large sparse virtual address space: multi-level paging, inverted page table
- Virtual space is segmented
- Segments and physical memory is paged

- Virtual addr: 
  | seg # | virtual page # | offset |
  | s-bit | m-bit | n-bit |

- Physical addr: 
  | physical page # | offset |
  | p-bit | n-bit |
Each process has a segment table in memory
For each valid segment, there is a page table in memory
Segment table entry as before except base contains physical address of associated page table
Page table as in demand-paging
Processor STBR: points to segment table of running process
Processor TLB: each entry has segment- and page-related fields

<table>
<thead>
<tr>
<th>valid</th>
<th>segment #</th>
<th>virtual page #</th>
<th>phys page #</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Overview
Segmentation
Non-demand Paging
Demand Paging
Paged-Segmentation
Sparse Virtual Space
IA-32: x86 – 32-bit
Sparse Virtual Space

- Large virtual address size $\Rightarrow$ infeasibly large page tables
  
  - 32-bit virtual addr, 4KB page
    $\Rightarrow$ \( \frac{2^{32}}{2^{12}} = 2^{20} = 1\text{M entries per page table} \)
  
  - 64-bit virtual address, 4KB pages
    $\Rightarrow$ \( \frac{2^{64}}{2^{12}} = 2^{52} = 1\text{G} \times 1\text{M entries in page table} \)
  
  - 64-bit virtual address, 4MB pages
    $\Rightarrow$ \( \frac{2^{64}}{2^{22}} = 2^{42} = 1\text{G} \times 1\text{K entries in page table} \)

- Solutions
  
  - Large or variable-size pages
  - Multi-level paging
  - Inverted page tables (Hashing)
Two-level Paging

- Virtual addr:
  - vpn1 #
  - vpn2 #
  - offset

- Physical addr:
  - physical page #
  - offset

In general, level $j + 1$ page table needed only for valid level $j$ entry
Inverted Page Tables

- Inverted page table
  - one table per OS, not one per process
  - row for every physical page
  - row \( j \) identifies the virtual page in physical page \( j \)
    
    \[
    j = \text{[process id (\( pid \)); virtual page number (\( vpn \)); access; ...]}
    \]

- Saves space if virtual mem > physical mem or \# processes > 1

- Given virtual address \( v \) \( offset \) and process id \( p \)
  - search table for a row \( j \) with matching \( [p, v] \)
  - if found \( \text{phys addr} \leftarrow [j, offset] \) else exception

- Typically use hashing for efficiency
  - so let \( j \) be \( \text{hash}(p, vpn) \), then proceed as above

- Difficult to accommodate page sharing between processes
Outline

Overview
Segmentation
Non-demand Paging
Demand Paging
Paged-Segmentation
Sparse Virtual Space

IA-32: x86 – 32-bit
IA-32 Address Translation

- Provides segmentation and optional two-level paging
- Virtual address: [segment selector (16-bit), offset (32 bit)]

- Linear address generation
  - segment selector: points to segment descriptor
  - segment descriptor: contains 32-bit base addr
  - linear addr = base addr + offset

- Without paging: physical addr = linear addr

- With paging: linear addr → phys addr via 2-level paging
  - level-1 page table: directory table
  - level-2 page table: page table
  - 32-bit offset = [dir (10-bit), vpn (10-bit), page offset (12-bit)]
  - dir: index into directory table; yields page table addr
  - vpn: index into page table; yields phys page addr
  - physical addr = z + page offset