

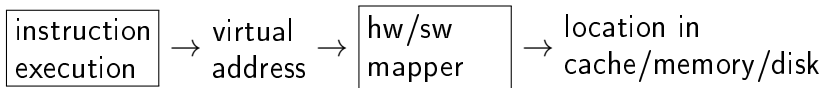
Operating Systems: Virtual Memory

Shankar

March 25, 2021

1. Overview
2. Segmentation
3. Non-demand Paging
4. Demand Paging
5. Paged-Segmentation
6. Sparse Virtual Space
7. IA-32: x86 – 32-bit

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



- if location in disk
 - hw traps → sw moves content to memory → resume instruction

- Program's address space starts at 0
- 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

- CPU 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 cacheing in CPU to achieve low overhead of mapping
 - TLB (translation-lookaside buffer): physical or virtual
- ...

- 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 part of phase i , for $i = 1, 2, \dots$
 - loads in phase $i + 1$, overwriting phase i
 - jumps to start of phase $i + 1$
- No kernel intervention
- No virtual address space: instructions generate physical addresses

1. Overview
2. Segmentation
3. Non-demand Paging
4. Demand Paging
5. Paged-Segmentation
6. Sparse Virtual Space
7. IA-32: x86 – 32-bit

- Virtual address:

seg#	offset
------	--------

m -bit	n -bit
----------	----------

 - seg#'s: $0, \dots, 2^m - 1$
 - seg max size: 2^n bytes
- Instructions generate virtual addresses
 - eg, load 2:0x030, reg1 // put data at 2:0x030 into reg1
- Physical address:

addr

- Mapping defined by a [segment table](#) in memory
- Program space consists of segments
 - eg, seg0: code; seg2: data; set5: stack; ...
 - each segment has a size (in $0, \dots, 2^n$)
 - each segment has access attributes (x, 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

- Kernel has a **segment table**
- Each process has a **segment table** // part of PCB

seg#	<i>valid</i>	<i>base</i>	<i>size</i>	<i>access</i>
0				
1				
	⋮			⋮

- Row for every j in $\{0, \dots, 2^m - 1\}$ // **descriptor** of j
 - valid* (1 bit): true 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.

- CPU has a STBR
 - CPU treats wherever it points as the current segment table
 - below, we refer to this table as “STBR.table”
- When a process or the kernel is dispatched:
STBR \leftarrow phys addr of the corresponding segment table

■ virtual address

<i>sn</i>	<i>offset</i>
-----------	---------------

\rightarrow

physical address

STBR.table[<i>sn</i>]. <i>base</i> + <i>offset</i>
--

- Instruction generates virtual addr

<i>segno</i>	<i>offset</i>
--------------	---------------

 in user mode
- CPU does following
 - $sd \leftarrow \text{STBR.table}[segno]$
 - if $sd.valid$ is false \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
 - CPU enters kernel mode
 - executes exception handler
- Requires 2 memory accesses for every virtual address
 - Overcome with caches in the CPU

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



<i>valid</i> (1 bit)	<i>sn</i> (segment #)	<i>base</i>	<i>size</i>	<i>access</i>
⋮				⋮

- for an entry x with *valid* bit set:
 STBR.table[$x.sn$].*valid* is true and
 [$x.base, x.size, x.access$] equals STBR.table[$x.sn$]
 - Hardware associatively searches on *valid* and *sn* fields
 - Kernel clears all *valid* bits when a process/kernel is dispatched
 - unless TLB also has a *pid* field // associatively searched
 - TLB's *valid* bit is not the same segment table's *valid* bit

- CPU generates virtual address *segno:offset*
- If TLB does not have *valid-segno* entry // TLB miss
 - if `STBR.table[segno].valid` false \rightarrow invalid-segment exception
 - create a *valid-segno* TLB entry from `STBR.table[segno]`
 - if TLB was full, overwrite an entry

Let x be *valid-segno* TLB entry // TLB hit

- if access invalid wrt $x.access$ \rightarrow invalid-access exception
 - if $offset > x.size$ \rightarrow out-of-bounds exception
 - $physaddr \leftarrow x.base + offset$
 - access physical memory location *physaddr*
-
- Cache replacement policy: fast and simple
 - FIFO, random, LRU approx, ...

- TLB is not the only cache in the CPU
- 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*)

<i>valid</i> (1 bit)	<i>pa</i> (phys addr)	<i>data</i> (of block <i>pa</i>)
⋮		⋮

- associatively searched on *valid* and *pa* fields
- Virtual memory cache: holds a portion of virtual memory
- Cache access order: virtual mem → TLB → phys mem
- Multiple levels of caches

1. Overview
2. Segmentation
3. Non-demand Paging
4. Demand Paging
5. Paged-Segmentation
6. Sparse Virtual Space
7. IA-32: x86 – 32-bit

- Virtual space is linear (not segmented)
- Virtual space and physical memory divided into **pages**

Virtual addr:

virtual page #	offset
----------------	--------

 m -bit n -bit

Physical addr:

physical page #	offset
-----------------	--------

 p -bit n -bit

- virtual page# range: $0, \dots, 2^m - 1$
- physical page# range: $0, \dots, 2^p - 1$
- 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 (x, 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)

- Kernel and each process has a **page table** in memory // part of PCB

<i>vpn</i> (virtual page#)	<i>valid</i> (1 bit)	<i>ppn</i> (physical page) #	<i>access</i> (r, r/w, ...)
0			
1			
	⋮		⋮

- Row for every j in $\{0, \dots, 2^m - 1\}$ // **descriptor** of j
 - valid* (1 bit): true iff process has page j
 - ppn*: physical page # where j is mapped (if *valid* true)
 - access*: allowed access to page j
- Layout fixed by hardware

- CPU has a PTBR
 - CPU treats wherever it points as the current page table
 - below, we refer to this table as “PTBR.table”
- When a process or kernel is dispatched:
PTBR \leftarrow phys addr of the corresponding page table



- Instruction generates virtual addr

<i>vpn</i>	<i>offset</i>
------------	---------------

 in user mode
- CPU does following
 - $pd \leftarrow \text{PTBR.table}[vpn]$
 - if $pd.valid$ is false \rightarrow invalid-page exception
 - if $pd.access$ violated \rightarrow invalid-access exception
 - $phyaddr \leftarrow$

$sd.ppn$	<i>offset</i>
----------	---------------
 - access physical memory location $phyaddr$
- Upon exception
 - CPU enters kernel mode
 - executes exception handler
- Requires 2 memory accesses for every virtual address
 - Overcome with caches in the CPU

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

<i>valid</i>	<i>vpn</i> (virtual page #)	<i>ppn</i> (phys page #)	<i>access</i>
⋮			⋮

- for an entry x with *valid* bit set:
 - $PTBR.table[x.vpn].valid$ is true and
 - $x.[ppn, access]$ equals $PTBR.table[x.vpn].[ppn, access]$
- Hardware associatively searches on *valid* and *vpn* fields
- Kernel clears all *valid* bits when a process is dispatched
 - unless TLB also has a *pid* field // associatively searched

- CPU generates virtual address

<i>vpn</i>	<i>offset</i>
------------	---------------
- If TLB does not have *valid-vpn* entry // TLB miss
 - if `PTBR.table[vpn].valid` false \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
 - $physaddr \leftarrow$

$x.ppn$	<i>offset</i>
---------	---------------
 - access physical memory location *physaddr*
-
- Interaction with physical-memory and virtual-memory caches

1. Overview
2. Segmentation
3. Non-demand Paging
4. Demand Paging
5. Paged-Segmentation
6. Sparse Virtual Space
7. 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 per process and kernel

<i>vpn</i>	<i>valid</i>	<i>ppn</i>	<i>access</i>	<i>dirty</i>	<i>hw-usage</i>
0					
	⋮				⋮

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

- 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 \longrightarrow global usage info, shared vpages

- CPU has PTBR that points to current page table



- CPU has TLB for fast address translation

<i>valid</i>	<i>vpn</i>	<i>ppn</i>	<i>access</i>	<i>dirty</i>	<i>hw-usage</i>
⋮					⋮

- *hw-usage*: if present in page table
- when entry x is evicted

PTBR.table[$x.vpn$].*dirty*/*hw-usage* \leftarrow x .*dirty*/*hw-usage*

- CPU generates virtual address

<i>vpn</i>	<i>offset</i>
------------	---------------

 in user mode

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

Let *x* be *valid-vpn* TLB entry // TLB hit

- if access invalid wrt *x.access* \rightarrow invalid-access exception
- *physaddr* \leftarrow

<i>x.ppn</i>	<i>offset</i>
--------------	---------------
- access physical memory location *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

■ 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
- Kernel 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)
// R is 1 means referenced
 - if head page's R is 0, free page (clean if needed)
 - if head page has R is 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

- In FIFO, more memory may cause more page faults
 - Referenced sequence (of vp#s): 1 2 3 4 1 2 5 1 2 3 4 5

3 physical pages, all initially free

	1	2	3	4	1	2	5	1	2	3	4	5	9 faults.
f1	1			4			5						-
f2		2			1			-		3			
f3			3			2			-		4		

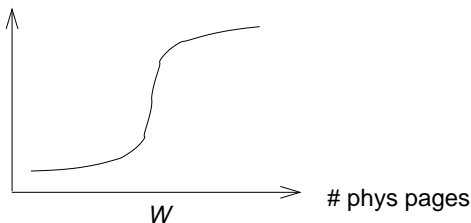
4 physical pages, all initially free

	1	2	3	4	1	2	5	1	2	3	4	5	10 faults
f1	1				-		5				4		
f2		2				-		1				5	
f3			3						2				
f4				4						3			

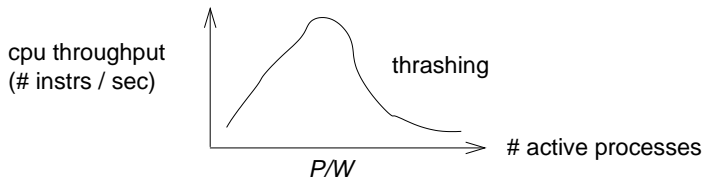
- LRU (and most others) do not suffer from Belady's anomaly

- 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

instr executed
between successive
page faults



- 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 use (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

1. Overview
2. Segmentation
3. Non-demand Paging
4. Demand Paging
5. Paged-Segmentation
6. Sparse Virtual Space
7. IA-32: x86 – 32-bit

- Virtual space is segmented
- Segments and physical memory is paged

■ Virtual addr:

seg #	virtual page #	offset
<i>s</i> -bit	<i>m</i> -bit	<i>n</i> -bit

■ Physical addr:

physical page #	offset
<i>p</i> -bit	<i>n</i> -bit

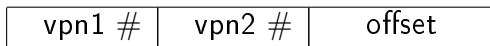
- Each process has a segment table in memory
- For each valid segment, there is a page table in memory
- Segment table entry as before except *base* contains physical address of associated page table
- Page table as in demand-paging
- CPU has STBR: points to segment table of running process
- CPU has TLB: each entry has segment- and page-related fields

<i>valid</i>	segment #	virtual page #	phys page #	<i>access</i>
•				•
•				•
•				•

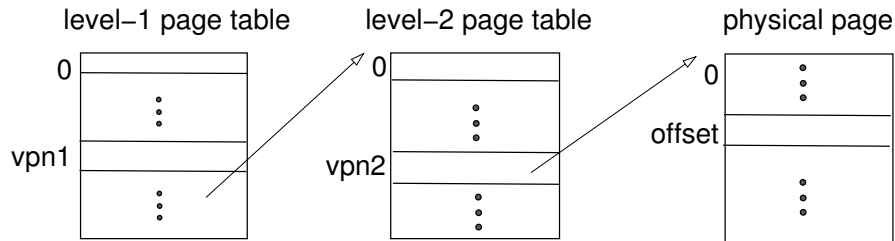
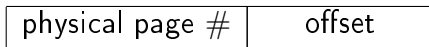
1. Overview
2. Segmentation
3. Non-demand Paging
4. Demand Paging
5. Paged-Segmentation
6. Sparse Virtual Space
7. IA-32: x86 – 32-bit

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

- Virtual addr:



- Physical addr:



- In general, level $j + 1$ page table needed only for **valid** level j entry

- Inverted page table
 - one table per kernel, not one per process
 - row for every **physical** page
 - row j identifies the virtual page in physical page j
= [process id (pid); virtual page number (vpn); *access*; ...]
- Saves space if virtual mem > physical mem or # processes > 1
- Given virtual address

v	<i>offset</i>
-----	---------------

 and process id p
 - search table for a row j with matching $[p, v]$
 - if found $phys\ addr \leftarrow [j, offset]$ else exception
- Typically use hashing for efficiency
 - so let j be $hash(p, vpn)$, then proceed as above
- Difficult to accomodate page sharing between processes

1. Overview
2. Segmentation
3. Non-demand Paging
4. Demand Paging
5. Paged-Segmentation
6. Sparse Virtual Space
7. IA-32: x86 – 32-bit

- 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 \rightarrow 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*