Memory Management

The Virtual Memory Abstraction

Physical Memory

- Unprotected address space
- Limited size
- Shared physical frames
- Easy to share data

Virtual Memory

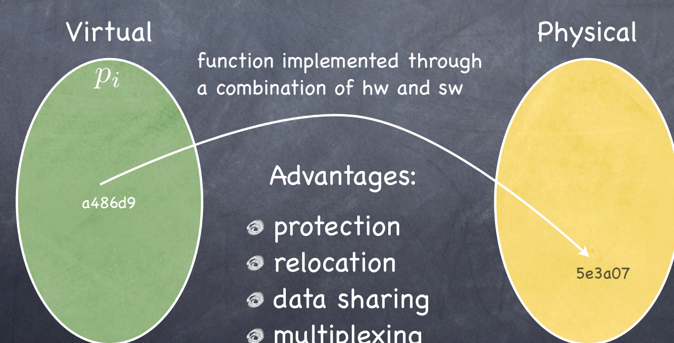
- Programs are isolated
- Arbitrary size
- All programs loaded at "0"
- Sharing is possible

Address spaces: Physical and Virtual

- Physical address space consists of the collection of memory addresses supported by the hardware
- Virtual address space consists of the collection of addresses that the process can "touch"
- Note: CPU generates virtual addresses

Address Translation

- A function that maps $\langle pid, virtual\ address \rangle$ into $physical\ address$



Protection

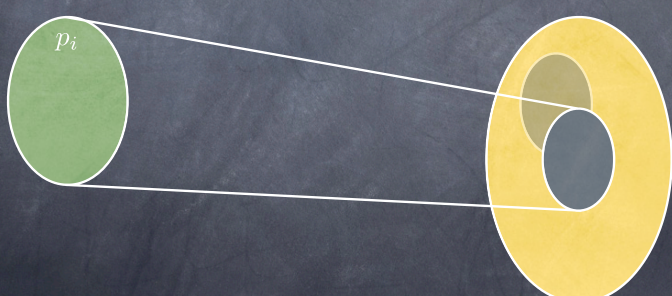
- At all times, the functions used by different processes map to disjoint ranges

Relocation

- The range of the function used by a process can change over time

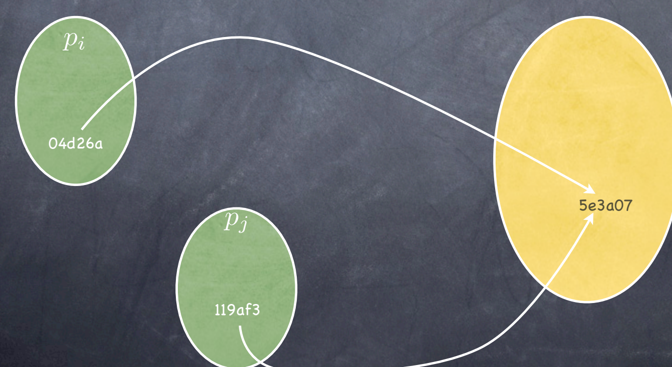
Relocation

- The range of the function used by a process can change over time



Data Sharing

- Map different virtual addresses of different processes to the same physical address

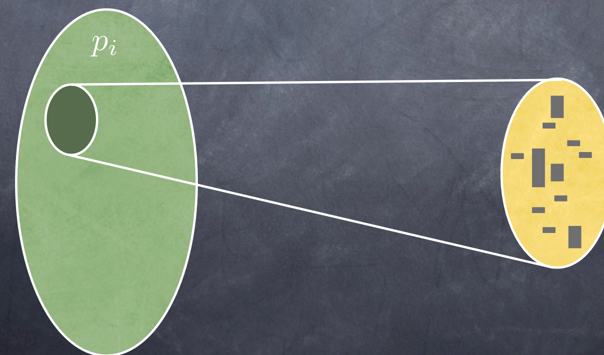


Contiguity

- Contiguous addresses in the domain need not map to contiguous addresses in the codomain

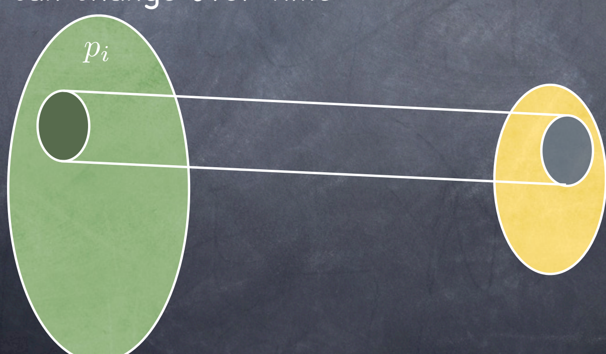
Contiguity

- Contiguous addresses in the domain need not map to contiguous addresses in the codomain



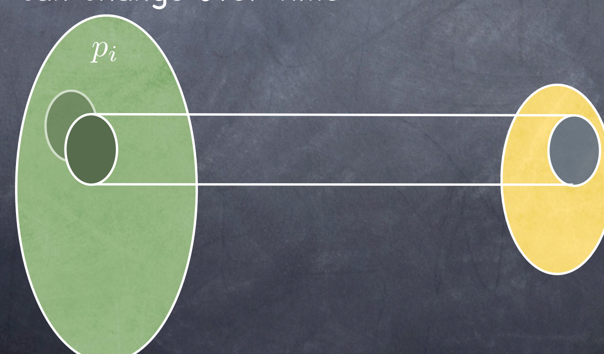
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



Multiplexing

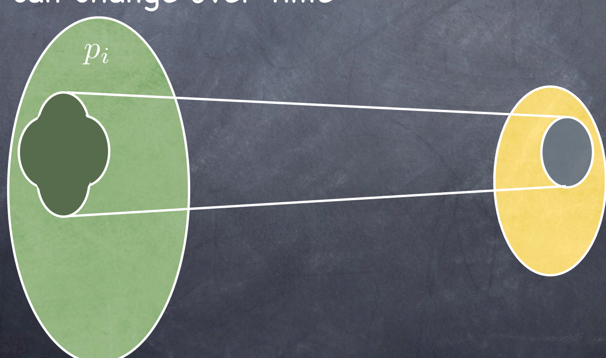
- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



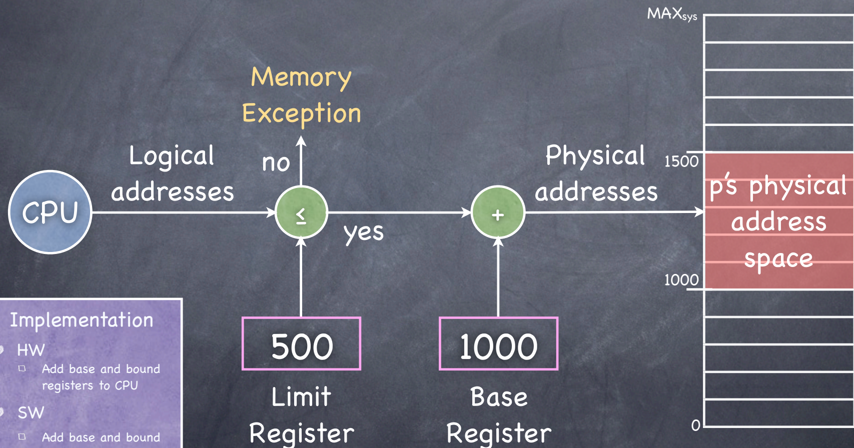
One idea, many implementations

- Base and limit
- Segment table
  - maps variable-sized ranges of contiguous VAs to a range of contiguous PAs
- Page table
  - map fixed-size ranges of contiguous VAs to fixed sized ranges of contiguous PAs
- Paged segmentation
- Multilevel page tables
- Inverted page table

It's all just a lookup...

Virtual Address	Physical Address
0	a30940
1	56bb03
10	240421
unmapped	
unmapped	
unmapped	
ffffff	d82a04

Base & Limit



Implementation

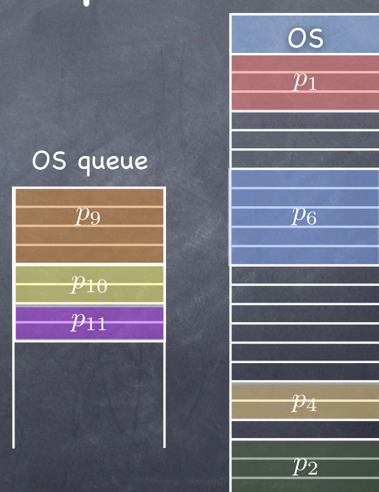
- HW
  - Add base and bound registers to CPU
- SW
  - Add base and bound registers to PCB
  - On context switch, change B&B (privileged)

On Base & Limit

- Contiguous Allocation: contiguous virtual addresses are mapped to contiguous physical addresses
- Protection is easy, but sharing is hard
  - Two copies of emacs: want to share code, but have data and stack distinct...
- Managing heap and stack dynamically is hard
  - We want them as far as as possible in virtual address space, but...

Contiguous allocation: multiple variable partitions

- OS keeps track of empty blocks ("holes")
- Initially, one big hole!
- Over time, a queue of processes (with their memory requirements) and a list of holes
- OS decides which process to load in memory next
- Once process is done, it releases memory

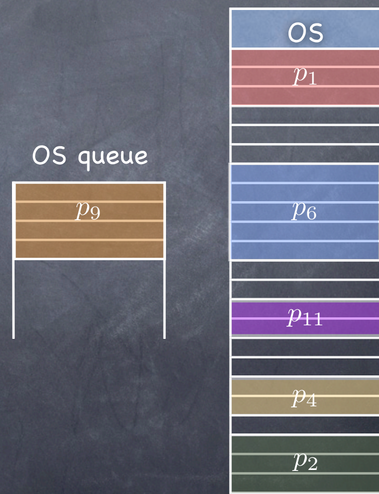


Strategies for Contiguous Memory Allocation

- First Fit
  - Allocate first big-enough hole
- Next Fit
  - As first fit, but start to search where you previously left off
- Best Fit
  - Allocate smallest big-enough hole

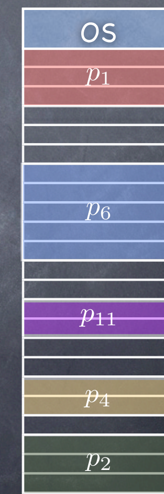
Fragmentation

- External fragmentation
  - Unusable memory between units of allocation



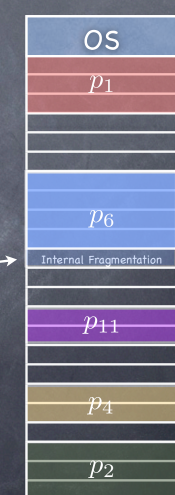
Fragmentation

- External fragmentation
  - Unusable memory between units of allocation
- Internal fragmentation
  - Unusable memory within a unit of allocation



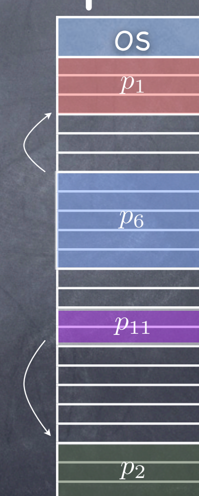
Fragmentation

- External fragmentation
  - Unusable memory between units of allocation
- Internal fragmentation
  - Unusable memory within a unit of allocation



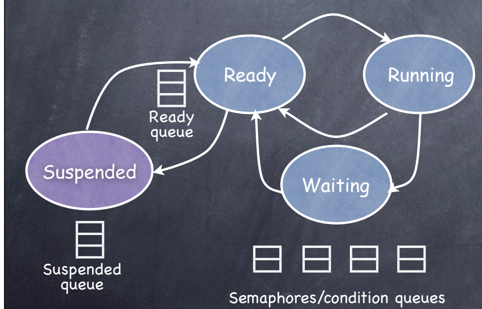
Eliminating External Fragmentation: Compaction

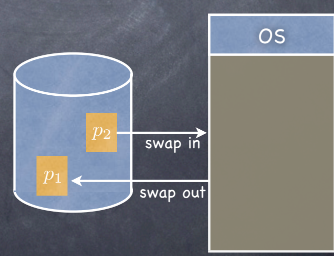
- Relocate programs to coalesce holes
- Problem with I/O
  - Pin job in memory while it is performing I/O
  - Do I/O in OS buffers



Eliminating External Fragmentation: Swapping

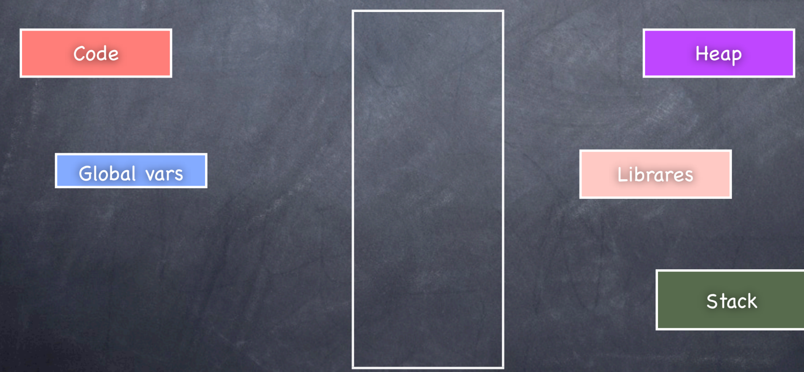
- Preempt processes and reclaim their memory
- Move images of suspended processes to backing store





E Pluribus Unum

- From a user's perspective, a process is a collection of distinct logical address spaces



E Pluribus Unum

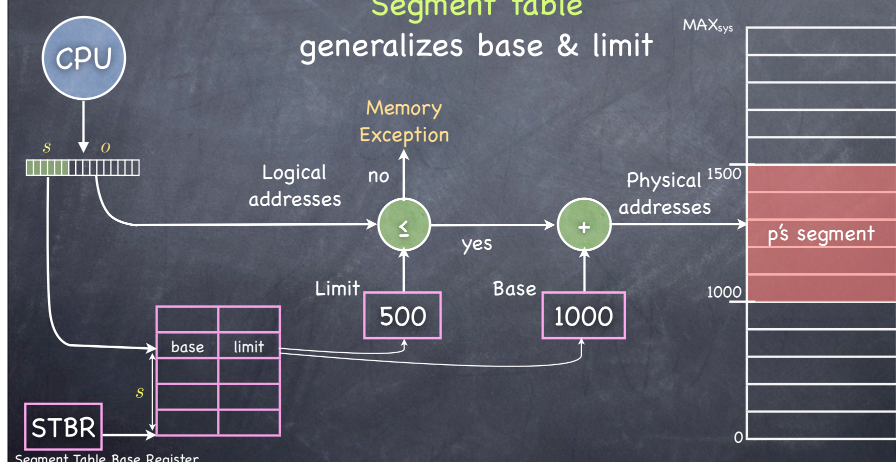
- From a user's perspective, a process is a collection of distinct logical address spaces

We call these logical address spaces segments

- Contiguous mapping of addresses within segment
- Holes in Virtual address space: a problem?
- Think of address as (s, o)
  - s is the segment number
  - o is the offset within the segment

Implementing Segmentation

Segment table generalizes base & limit



On Segmentation

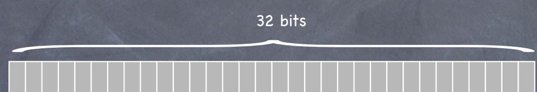
- Sharing a segment is easy!
- Protection bits control access to shared segments
- External fragmentation...
- Each process maintains a segment table, which is saved to PCB on a context switch
- Fast?
- How do we enlarge a segment?

base	limit
400	600
2900	200
2500	200
3200	500
1300	1000

Paging

- Allocate VA & PA memory in fixed-sized chunks (pages and frames, respectively)
  - memory allocation can use a bitmap
  - typical size of page/frame: 4KB to 16KB
- Gives illusion of contiguity...
  - ...but adjacent pages in VA need not map to contiguous frames in PA
- Of course, now internal fragmentation...

Virtual address



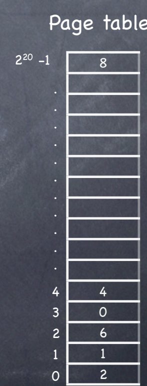
- Two components
  - page number
  - offset within page

Virtual address

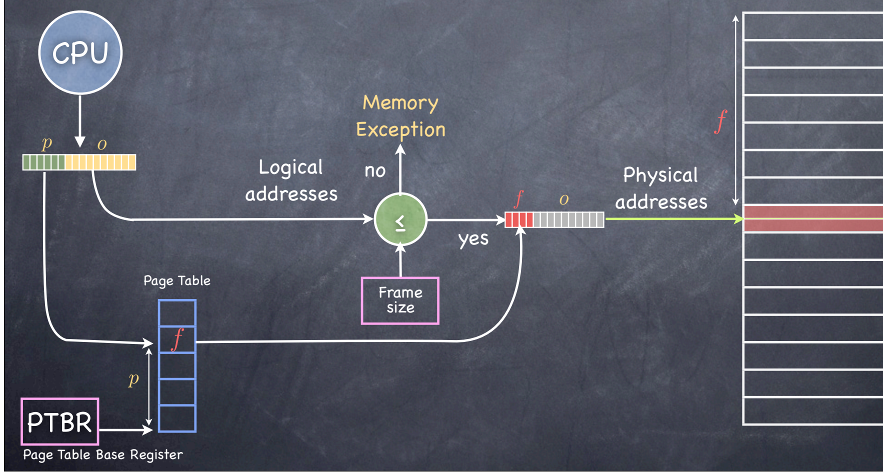
- Two components
  - page number – how many pages in the VA
  - offset within page – how large is a page?
- To access a piece of data
  - extract page number
  - extract offset
  - translate page number to frame number
  - access data at offset in frame

Virtual address

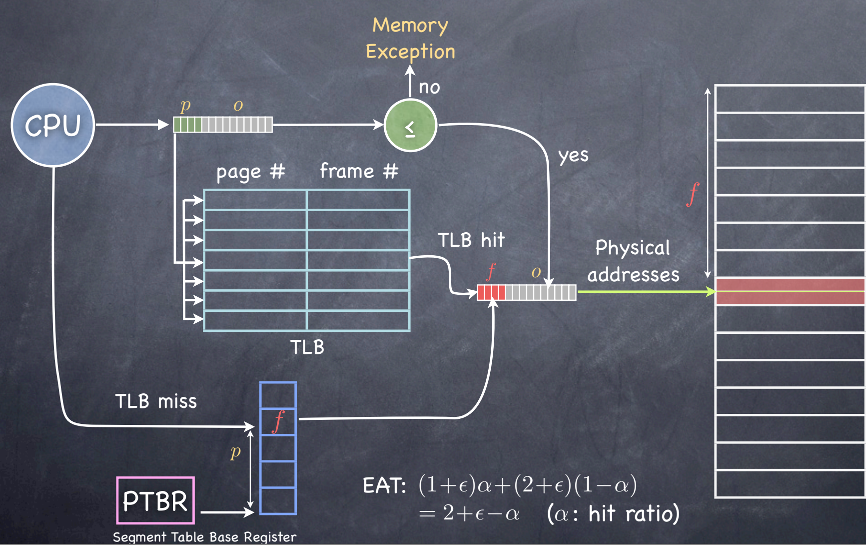
- Two components
  - page number - how many pages in the VA
  - offset within page - how large is a page?
- To access a piece of data
  - extract page number
  - extract offset
  - translate page number to frame number
  - access data at offset in frame



Basic Paging Implementation

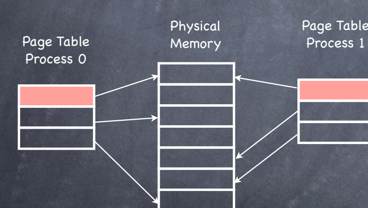


Speeding things up



EAT: $(1+\epsilon)\alpha+(2+\epsilon)(1-\alpha)$
$= 2+\epsilon-\alpha$ (α: hit ratio)

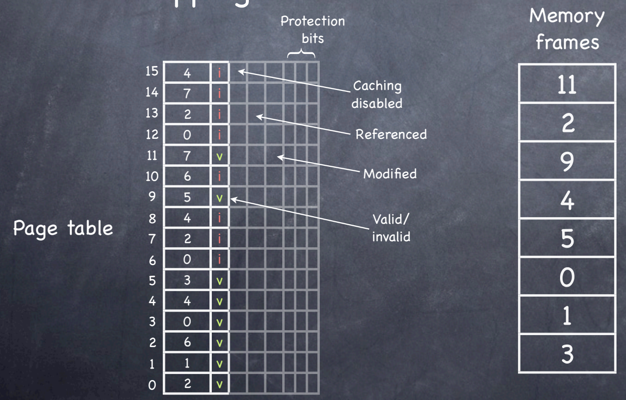
Sharing



- Processes share pages by mapping virtual pages tot he same memory frame
  - code segments of processes running same program can share pages with executables
- Fine tuning using protection bits (rwx)

Memory Protection

- Used valid/invalid bit to indicate which mappings are active



What happens on a TLB miss?

- Can be handled in software or hardware

Software

- TLB generates trap
- Switch to kernel mode
- OS does translation
- OS loads new TLB entry and returns from trap

On Context Switch

- Flush TLB or
  - add PID tag to TLB
  - add a CPU register
  - change PID register on context switch

Hardware

- HW includes PTB register
- HW follows pointer and does look up in page table
- Exception handler invoked only if no/bad mapping/permission

On Context Switch

- change value stored in PTB register
- flush TLB

Space overhead

- Two sources
  - data structure overhead (the page table!)
  - fragmentation
    - How large should a page be?

Overhead for paging:

(#entries x sizeofEntry) + (#"segments" x pageSize/2) =

= ((VA_Size/pagesize) x sizeofEntry) + (#"segments" x pageSize/2)

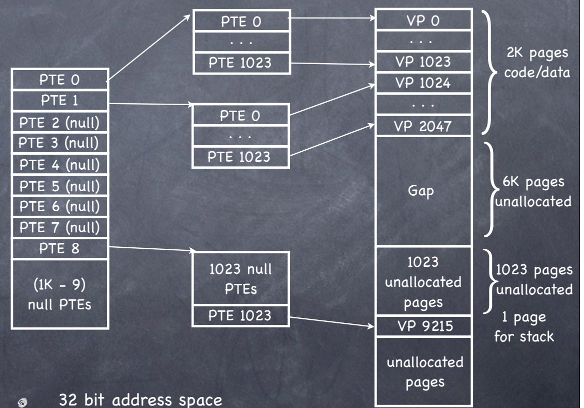
- Size of entry
  - enough bits to identify physical page ($\log_2$ (PA_Size / page size))
  - should include control bits (valid, modified, referenced, etc)
  - usually word or byte aligned

Computing paging overhead

- 1 MB maximum VA, 1 KB page, 3 segments (program, stack, heap)
  - $((2^{20} / 2^{10}) \times$ sizeofEntry$) + (3 \times 2^9)$
  - If I know PA is 64 KB then sizeofEntry = 6 bits (2^6 frames) + control bits
    - if 3 control bits, byte aligned size of entry: 16 bits

Oops...

- What is the size of the page table for a machine with 64-bit addresses and a page size of 4KB?
- Good news
  - much of the space is unused
- Use a smarter data structure to capture page table
  - tree!

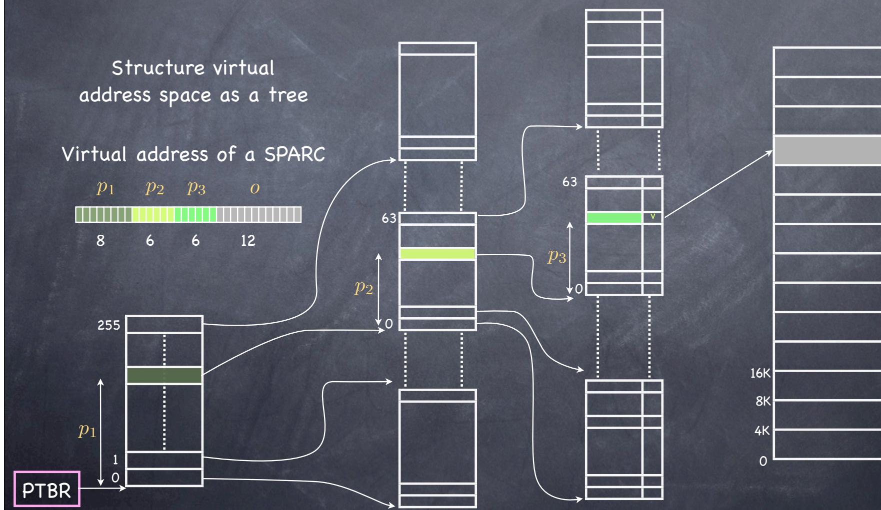


- 32 bit address space
- 4Kb pages
- 4 bytes PTE

Multi-level Paging

Structure virtual address space as a tree

Virtual address of a SPARC



Examples

- Two level paging

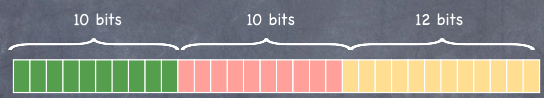
Examples

- Two level paging

  - Outer page table fits in a page
  - Rest of page table allocated in page-size chunks

Examples

- Two level paging



  - Outer page table fits in a page
  - Rest of page table allocated in page-size chunks
  - internal fragmentation (where?)
  - increased TLB miss time

Examples

- 64-bit VA; 2K page; 4 byte/entry
- How many levels?
  - each page table includes 512 entries (2^9)
  - number of pages = $2^{64}/2^{11}$
  - number of levels - 53/9 = 6 (rounded up)

The Challenge of Large Address Spaces

- With large address spaces (64-bits) page tables become cumbersome
  - 5/6 levels of tables
- A new approach---make tables proportional to the size of the physical, not the virtual, address space
  - virtual address space is growing faster than physical

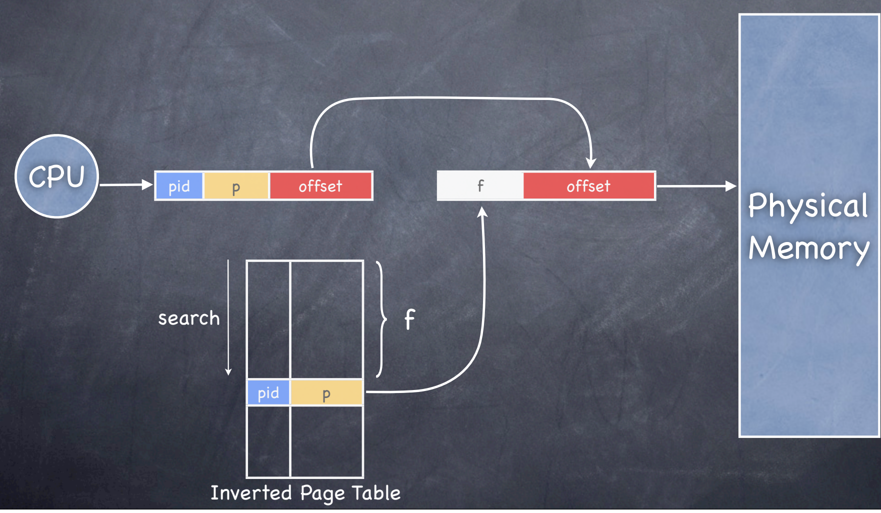
Page Registers (a.k.a. Inverted Page Tables)

- For each frame, a register containing
  - Residence bit
    - is the frame occupied?
  - Page # of the occupying page
  - Protection bits

Catch?

- An example
  - 16 MB of memory
  - Page size: 4k
  - # of frames: 4096
  - Used by page registers (8 bytes/register): 32 KB
  - Overhead: 0.2%
  - Insensitive to size of virtual memory

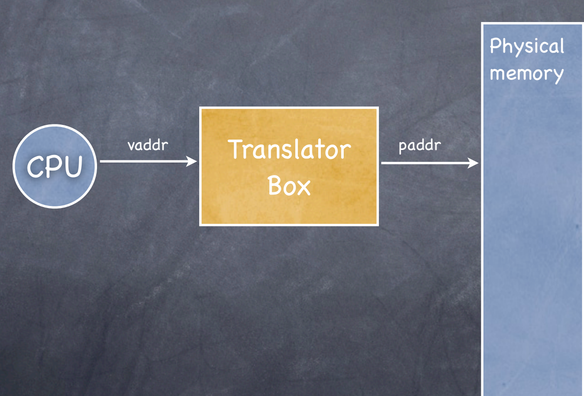
Basic Inverted Page Table Architecture



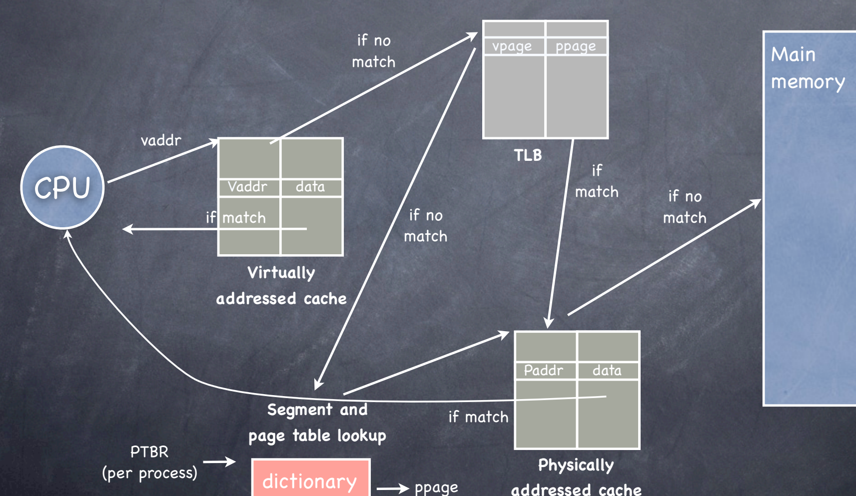
Where have all the pages gone?

- Searching 32KB of registers on every memory reference is not fun
- If the number of frames is small, the page registers can be placed in an associative memory---but...
- Large associative memories are expensive
  - hard to access in a single cycle.
  - consume lots of power

The BIG picture



The BIG picture



Time Overhead

- Average Memory Access Time (AMAT)
- $AMAT = T_{L1} + (P_{L1miss} \times T_{L1miss})$
- $T_{L1miss} = T_{TLB} + (P_{TLBmiss} \times T_{TLBmiss}) + T_{L2} + (P_{L2miss} \times T_{mem})$
- $T_{TLBmiss} = \#references \times (T_{L2} + P_{L2miss} \times T_{mem})$

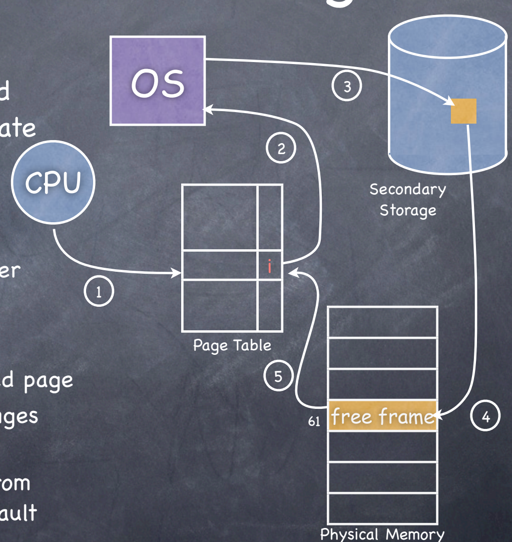
To fill TLB

Demand Paging

- Code pages are stored in a file on disk
  - some are currently residing in memory–most are not
- Data and stack pages are also stored in a file
- OS determines what portion of VAS is mapped in memory
  - this file is typically invisible to users
  - file only exists while a program is executing
- Creates mapping <u>on demand</u>

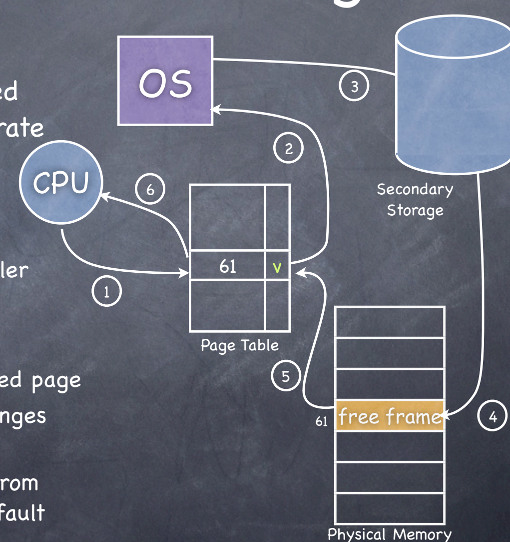
Page-Fault Handling

- References to a non-mapped page (i in page table) generate a page fault
- Handling a page fault:
  - Processor runs interrupt handler
  - OS blocks running process
  - OS finds a free frame
  - OS schedules read of unmapped page
  - When read completes, OS changes page table
  - OS restarts faulting process from instruction that caused page fault



Page-Fault Handling

- References to a non-mapped page (i in page table) generate a page fault
- Handling a page fault:
  - Processor runs interrupt handler
  - OS blocks running process
  - OS finds a free frame
  - OS schedules read of unmapped page
  - When read completes, OS changes page table
  - OS restarts faulting process from instruction that caused page fault



Taking a Step Back

- Physical and virtual memory partitioned into equal-sized units (respectively, frames and pages)
- Size of VAS decoupled to size of physical memory
- No external fragmentation
- Minimizing page faults is key to good performance

Page replacement

- Local vs Global replacement
  - Local: victim chosen from frames of faulty process
    - fixed allocation per process
  - Global: victim chosen from frames allocated to <u>any</u> process
    - variable allocation per process
- Many replacement policies
  - Random, FIFO, LRU, Clock, Working set, etc.
- Goal is minimizing number of page faults

FIFO Replacement

- First block loaded is first replaced
- Low overhead
- Commonly used

	a	b	a	d	g	a	f	d	g	a	f	c	b	g
F0	a	a	a	a	a	a	b							
F1		b	b	b	b	b	d							
F2				d	d	d	g							
F3					g	g	f							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	M	M	H	M	M	H	M							

FIFO Replacement

- First block loaded is first replaced
- Low overhead
- Commonly used

	a	b	a	d	g	a	f	d	g	a	f	c	b	g
F0	a	a	a	a	a	a	b	b	b	d	d	g	f	a
F1		b	b	b	b	b	d	d	d	g	g	f	a	c
F2				d	d	d	g	g	g	f	f	a	c	b
F3					g	g	f	f	f	a	a	c	b	g
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	M	M	H	M	M	H	M	H	H	M	H	M	M	M

LRU Replacement

- Replace block referenced least recently
- Reference stack
  - referenced block moved to top of stack
  - on page fault, block on bottom of stack is replacedand new block is placed on top of stack
- Difficult to implement

	a	b	a	d	g	a	f	d	g	a	f	c	b	g
F0	a	b	a	d	g	a	f	d	g	a	f	c	b	g
F1		a	b	a	d	g	a	f	d	g	a	f	c	b
F2				b	a	d	g	a	f	d	g	a	f	c
F3					b	b	d	g	a	f	d	g	a	f
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	M	M	H	M	M	H	M	H	H	H	H	M	M	M

Clock Replacement

- First-In-Not-Used -First-Out replacement
- Like FIFO, but add a "used" bit (*) for each queue entry and make queue circular
- Clock hand points to orange frame

	a	b	a	d	g	a	f	d	g	a	f	c	b	g
F0	a	a	a*	a*	a*	a*	a	a	a	a*	a*	a	a	g
F1		b	b	b	b	b	f	f	f	f	f*	f	f	f
F2				d	d	d	d	d*	d*	d*	d*	c	c	c
F3					g	g	g	g	g*	g*	g*	g	b	b
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	M	M	H	M	M	H	M	H	H	H	H	M	M	M

Optimal Replacement

- Replace block referenced furthest in future
- Minimum number of faults
- Impossible to implement

	a	b	a	d	g	a	f	d	g	a	f	c	b	g
F0	a	a	a	a	a	a	a	a	a	a	a	c	b	b
F1		b	b	b	b	b	f	f	f	f	f	f	f	f
F2			d	d	d	d	d	d	d	d	d	d	d	d
F3					g	g	g	g	g	g	g	g	g	g
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	M	M	H	M	M	H	M	H	H	H	H	M	M	H

Working Set Replacement

- Global replacement policy
- WS_t = set of pages referenced in (t-T+1, t)
- A page is replaced at t if it does not belong to WS_t
  - pages not necessarily replaced at page fault time!
  - adapts allocation to changes in locality

T = 4

	a	b	a	d	g	a	f	d	g	a	f	c	b	g
F0	a	a	a	a	a	a	a	a	a	a	a	a	a	g
F1		b	b	b	b	d	d	d	d	d	d	c	c	c
F2				d	d	g	g	g	g	g	g	g	b	b
F3					g		f	f	f	f	f	f	f	f
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	M	M	H	M	M	H	M	H	H	H	H	M	M	M

Thrashing

- If too much multiprogramming, pages tossed out while needed
- one program touches 50 pages
  - with enough pages, 100ns/ref
  - if too few and faults every 5th reference
    - 10ms for disk IO
    - one reference now costs 2ms: 20,000 times slowdown

T = 3	a	b	a	d	g	a	f	d	g	a	f	c	b	g
F0	a	a	a	a	a	a	a	a	g	g	g	c	c	c
F1		b	b	b	g	g	g	d	d	d	f	f	f	g
F2				d	d	d	f	f	f	a	a	a	b	b
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	M	M	H	M	M	H	M	M	M	M	M	M	M	M