Virtual Memory

Brian Bershad

Virtual Memory

- **VM allows a program to run with less than its full VAS in primary memory.**
- **So, a program can run on a machine with less memory than it "needs".**
  - » Can also run on a machine with "too much" physical memory.
- **Many programs don't need all of their code and data at once (or ever), so this saves space and might permit more programs to shared primary memory.**
- **The amount of real memory that a program needs to execute can be adjusted dynamically to suit the programs behavior.**
- **VM requires hardware support, as we saw previously, plus OS management algorithms.**

VM

- **VM should be completely invisible to the programmer.**
- **It's the responsibility of the OS to load those parts of the program that are needed, and to remove from memory parts that are not needed. How it does this is a *policy* decision.**
- **"Virtual Memory" is not free -- it must exist somewhere. Those program pages that are not stored in primary memory must be stored on "elsewhere".**

The Memory Hierarchy

Where	Time (cycles)	Space (bytes)
CPU	0	0
Registers	1	10-100
Cache	10^1	1K-1000K
Primary Memory	10^2	1M-1G
Secondary Storage	10^7	1G++
Tertiary Storage	10s of seconds	unlimited

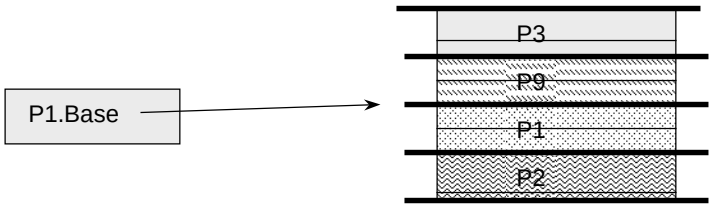
Issues

- History
  - » what did “virtualized store” used to look like
- Mechanism
  - » understanding how to make something “not there” appear there.
- Fetch strategies
  - » WHEN to bring something into primary memory
    - – **demand**
    - – **anticipatory**
- Replacement strategies
  - » WHICH page to throw out when you fetch something

History

- Uniprogramming with overlays
  - » manual
  - » no protection
- Multiprogramming
  - » more than 1 job in memory at a time
  - » fixed partition
    - – **resulted in internal fragmentation**
  - » variable partition
    - – **external fragmentation**
  - » protection
    - – **base (and) bounds registers**

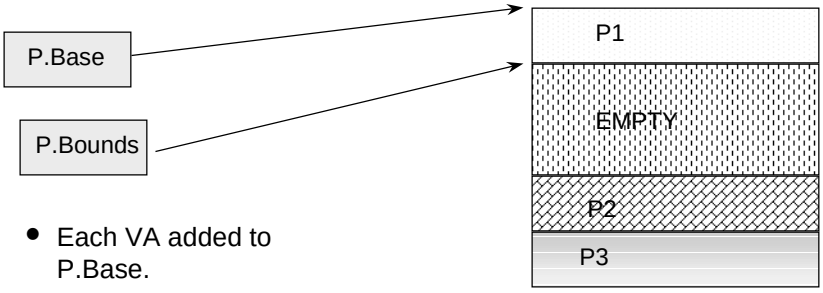
Early Memory management used Fixed Partitions



Easy
but limited.
Add a processes virtual address
to it's base register.
The size of each segment is fixed.

Internal fragmentation
within each allocated segment.

Variable partition was a step forward



- Each VA added to P.Base.
- Checked against P.Bounds.
- If less then, access allowed to derived physical address.

Could have external fragmentation.

Vanilla Multiprogramming

- Programs stay in main memory until done.
- Memory utilization was a problem
  - » external fragmentation meant we had enough, but not in the right place.
  - » processes could “hang out” for a long time and gobble up memory.
- Swapping was the answer.
  - » copy a process’s memory to disk
  - » free up the memory for another process.
  - » don’t run the swapped process until there is more memory
    - – **or someone hits a key...**
  - » Drastic, but simple.

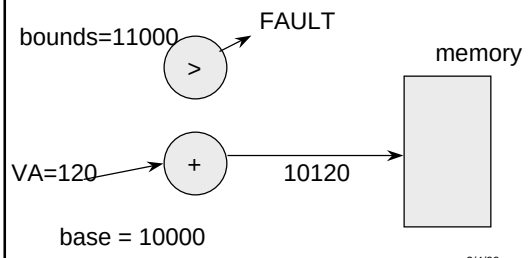
Virtual Memory

- Separate generated address from referenced address
  - » P = F(V)
    - – **where P is a physical address**
    - – **V is a virtual address**
    - – **F is an arbitrary function.**
- Motivation
  - » have > 1 process in memory at a time
    - – **F is process disjoint**
  - » Allow sizeof(V) to be >> sizeof(P)
    - – **F is many to one.**
  - » Allow sizeof(P) to be >> sizeof(V)
    - – **F is one to many**
  - » Sharing
    - – **F is many to many**
  - » Protection

Dynamic Relocation Registers

- Associate with each process a base and bounds register
  - » Add base to virtual address
  - » If result is > bounds, fault.
- Reload relocation register on context switch.

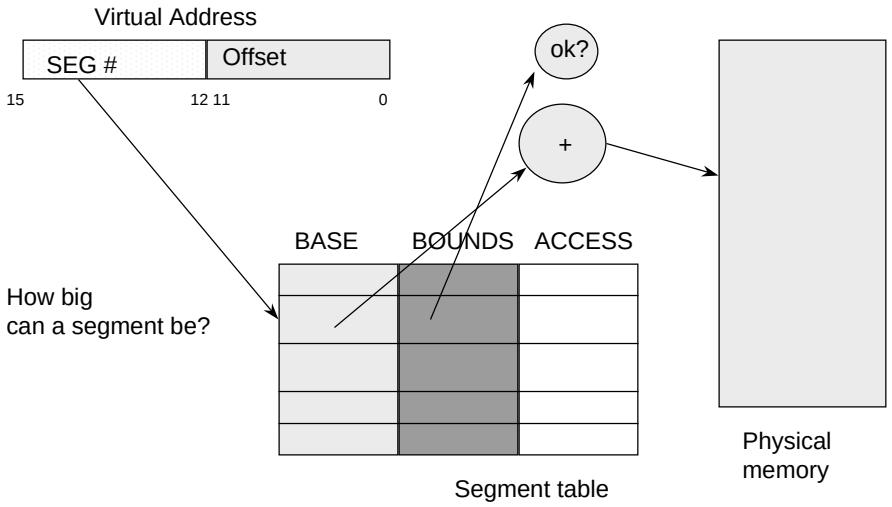
LD R3, @120 # load R3 with contents of memory location 120



Segmentation

- Have > 1 base,bounds register pair per process
  - » call each a “segment”
- Split process into multiple segments
  - » a segment is a collection of logically related data
    - – **code, module, stack, file**
- Put the segment registers into a table associated with each process.
- Include in the virtual address the segment number through which you are referencing memory.
- Bonus: add protection bits per segment into the table
  - » No Access, Read, Write, Execute

Segment Table

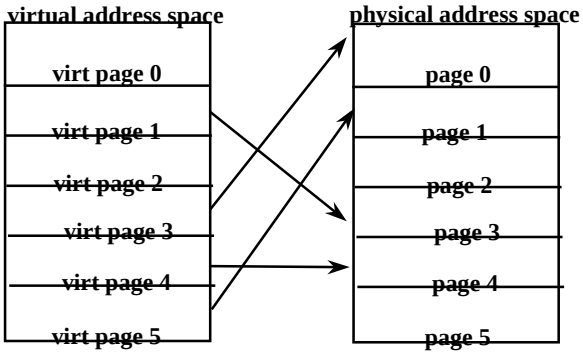


The Segment Table

- Can have one segment table per process
- To share memory, just share by putting the same translation into the base and bounds pair.
- Can share with different protections.
- Cross-segment names can be tough to deal with
  - » Segments need to have the same names in multiple processes if you want to share pointers.
- If the segment table is big, should keep it in main memory
  - » but then access is slow.
- So, keep a subset of the referenceable segments in a small on-chip memory and look up translation there.
  - » can be either automatic or manual.

Paging

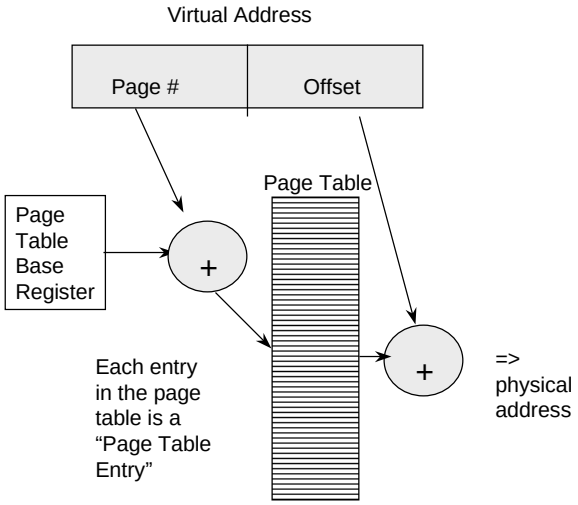
Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory.



10

Paging

- Goals
  - » make allocation and swapping easier
- Make all chunks of memory the same size
  - » call each chunk a "PAGE"
  - » example page sizes are 512 bytes, 1K, 4K, 8K, etc
  - » pages have been getting bigger with time
  - » we'll discuss reasons why pages should be of a certain size as the week progresses.



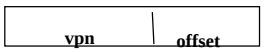
2/4/99

16

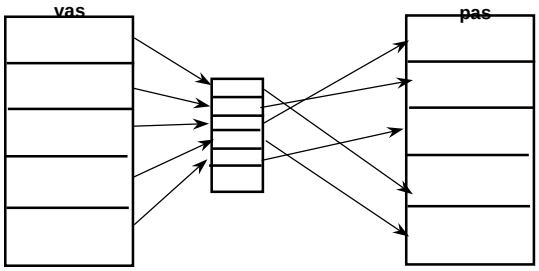
Paging

In a paged system, each virtual address has two parts: virtual page number and offset.

Page size is determined by hardware.



A Page Table contains the mappings from virtual page number (VPN) to physical page frame number (PFN).



11

Paging

- **User view memory as one contiguous address space from 0 through n.**
- **In reality, pages are scattered throughout physical storage.**
- **The mapping is invisible to the program; it is maintained by the OS and used by the hardware on each reference by the program.**
- **Protection is provided because a program cannot reference memory outside of its VAS.**
- **Hardware always contains a TLB to speedup the page table lookups.**

2/4/99 18

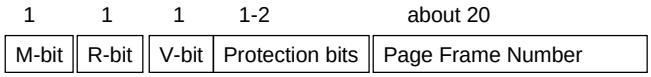
Sharing

- **Paging introduces the possibility of sharing memory.**
- **Several processes can share one or more pages, possibly with different Protection.**
- **A shared page may exist in different parts of the VAS of each process. (This is difficult if it contains internal addresses.)**

An Example

- Pages are 1024 bytes long
  - » this says that bottom 10 bits of the VA is the offset
- PTBR contains 2000
  - » this says that the first page table entry for this process is at physical memory location 2000
- Virtual address is 2256
  - » this says “page 2, offset 256”
- Physical memory location 2004 contains 8192
  - » this says that each PTE is 4 bytes (1 word)
  - » and that the second page of this process’s address space can be found at memory location 8192.
- So, we add 256 to 8192 and we get the real data!

What does a PTE contain?



- The Modify bit says whether or not the page has been written.
  - » it is updated each time a WRITE to the page occurs.
- The Reference bit says whether or not the page has been touched
  - » it is updated each time a READ or a WRITE occurs
- The V bit says whether or not the PTE can be used
  - » it is checked each time the virtual address is used
- The Protection bits say what operations are allowed on this page
  - » READ, WRITE, EXECUTE
- The Page Frame Number says where in memory is the page

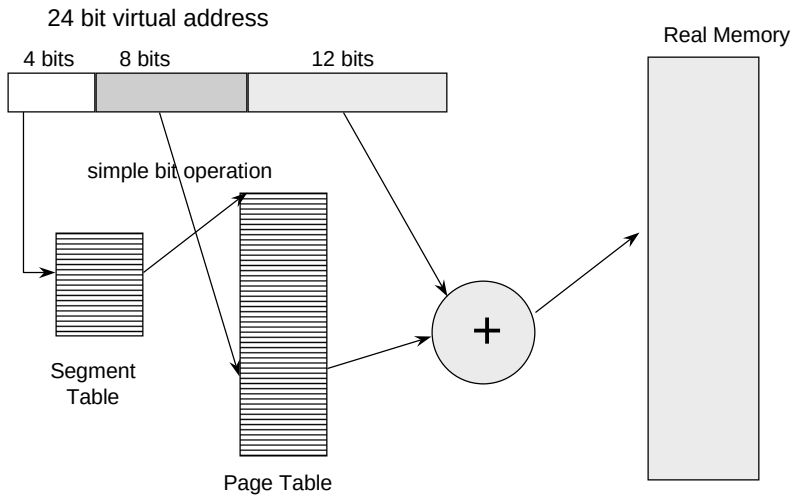
Evaluating Paging

- Easy to allocate memory
  - » memory comes from a free list of fixed size chunks.
  - » to find a new page, get anything off the free list.
  - » external fragmentation not a problem
- easy to swap out pieces of a program
  - » since all pieces are the same size.
  - » use valid bit to detect references to swapped pages
  - » pages are a nice multiple of the disk block size.
- Can still have internal fragmentation
- Table space can become a serious problem
  - » especially bad with small pages
    - – **eg, w/a 32bit address space and 1k size pages, that's 2^{22} pages or that many ptes which is a lot!**
- Memory reference overhead can be high
  - » 2 refs for every one

Segmentation and Paging at the Same Time

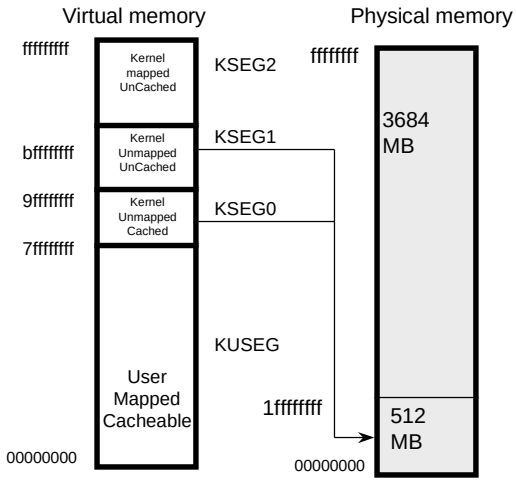
- Provide for two levels of mapping
- Use segments to contain logically related things
  - » code, data, stack
  - » can vary in size but are generally large.
- Use pages to describe components of the segments
  - » makes segments easy to manage and can swap memory between segments.
  - » need to allocate page table entries only for those pieces of the segments that have themselves been allocated.
- Segments that are shared can be represented with shared page tables for the segments themselves.
  - » examples include the VAX and the MIPS

An Early Example -- IBM System 370



MIPS R3000 VM Architecture

- User mode and kernel mode
  - » 2GB of user space
  - » When in user mode, can only access KUSEG.
- Three kernel regions; all are globally shared.
  - » KSEG0 contains kernel code and data, but is unmapped. Translations are direct.
  - » KSEG1 like KSEG0, but uncached. Used for I/O space.
  - » KSEG2 is kernel space, but cached and mapped. Contains page tables for KUSEG.
    - – **Implication is that the page tables are kept in VIRTUAL memory!**

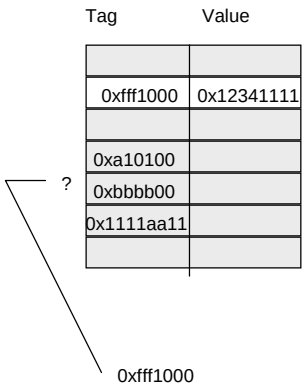


Lookups

- Each memory reference can be 3
  - » assuming no fault
- Can exploit locality to improve lookup strategy
  - » a process is likely to use only a few pages at a time
- Use Translation Lookaside buffer to exploit locality
  - » a TLB is a fast associative memory that keeps track of recent translations.
- The hardware searches the TLB on a memory reference
- On a TLB miss, either a hardware or software exception can occur
  - » older machines reloaded the TLB in hardware
  - » newer RISC machines tend to use software loaded TLBs
    - – **can have any structure you want for the page table**
    - – **fast handler computes and goes. Eg, the MIPS.**

A TLB

- A small fully associative cache
- Each entry contains a tag and a value.
  - » tags are virtual page numbers
  - » values are physical page table entries.
- Problems include
  - » keeping the TLB consistent with the PTE in main memory
    - – **valid and ref bits, for example**
  - » keeping TLBs consistent on an MP.
  - » quickly loading the TLB on a miss.
- Hit rates are important.



Software Loaded TLB

- The MIPS TLB contains 64 entries
  - » fully associative
  - » random replacement
- On a TLB miss to KUSEG, a trap occurs
  - » control transfers to UTLBMISS handler
- If a miss occurs in the miss handler, a second fault occurs.
  - » control transfers to a KTLBMISS handler
  - » the missed VA is “pte”
  - » the translation for the page table is loaded into the TLB.

```
UTLBmiss(vm_offset_t va)
{
  PTE_t pte;
  pte = PCB.PTBR;
  pte = pte +  (va / PAGE_SIZE);
  TLB_dropin(va,
         *pte);  /* could miss here */
 }
```

Selecting a page size

- Small pages give you lots of flexibility but at a high cost.
- Big pages are easy to manage, but not very flexible.
- Issues include
  - » TLB coverage
    - – **product of page size and # entries**
  - » internal fragmentation
    - – **likely to use less of a big page**
  - » # page faults and prefetch effect
    - – **small pages will force you to fault often**
  - » match to I/O bandwidth
    - – **want one miss to bring in a lot of data since it will take a long time.**