Virtual Memory

Lessons from History…

There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management.

Gordon Bell and Bill Strecker
speaking about the PDP-11 in 1976

A partial list of successful machines that eventually starved to death for lack of address bits includes the PDP 8, PDP 10, PDP 11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, Motorola 6800, AMI 6502, Zilog Z80, Cray-1, and Cray X-MP.

Hennessy & Patterson

Why? Address size determines minimum width of anything that can hold an address: PC, registers, memory words, HW for address arithmetic (BR/JMP, LD/ST). When you run out of address space it's time for a new ISA!

Top 10 Reasons for a BIG Address Space

10. Keeping Micron and Rambus in business.
9. Unique addresses within every internet host.
8. Generating good 6.004 quiz problems.
7. Performing 32-bit ADD via table lookup
6. Support for meaningless advertising hype
5. Emulation of a Turning Machine's tape.
4. Storing MP3s.
3. Isolating ISA from IMPLEMENTATION
   - *details of HW configuration shouldn't enter into SW design*
2. Usage UNCERTAINTY
   - *provide for run-time expansion of stack and heap*
1. Programming CONVENIENCE
   - *create regions of memory with different semantics: read-only, shared, etc.*
   - *avoid annoying bookkeeping*

Squandering Address Space

Address Space

CODE, large monolithic programs (eg, Office, Netscape)....

- only small portions might be used
- add-ins and plug-ins
- shared libraries/DLLs

•••

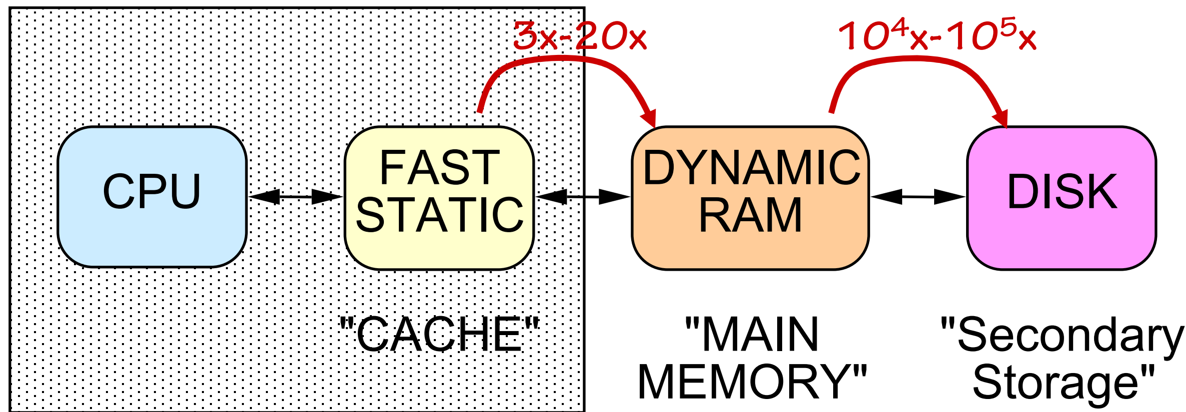
STACK: How much to reserve? (consider RECURSION!)

OBSERVATIONS:

- Can't BOUND each usage... without compromising use.
- Actual use is SPARSE
- Working set even MORE sparse

HEAP: N variable-size data records...
Bound N? Bound Size?

Extending the Memory Hierarchy



So, we've used SMALL fast memory + BIG slow memory to fake BIG FAST memory.

Can we combine RAM and DISK to fake DISK size at RAM speeds?

VIRTUAL MEMORY

- use of RAM as cache to much larger storage pool, on slower devices
- TRANSPARENCY - VM locations "look" the same to program whether on DISK or in RAM.
- ISOLATION of RAM size from software.

Virtual Memory

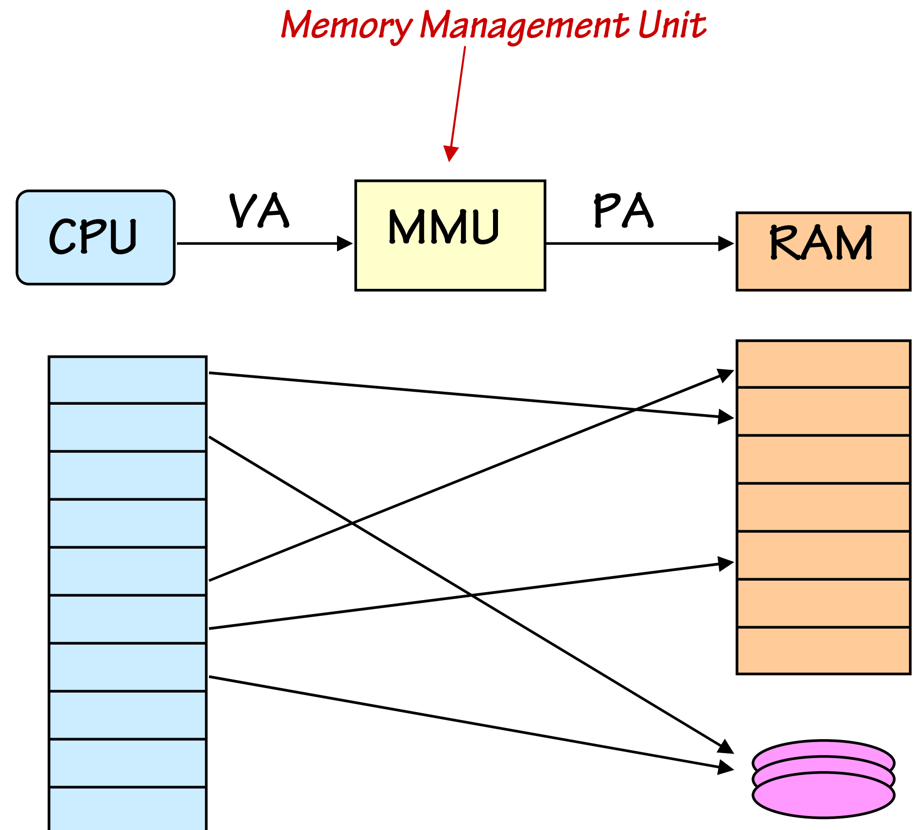
ILLUSION: Huge memory
(2^{32} bytes? 2^{64}bytes?)

ACTIVE USAGE: small fraction
(2^{24} bytes?)

HARDWARE:
- 2^{26} (64M) bytes of RAM
- 2^{32} (4 G) bytes of DISK...

... maybe more, maybe less!

ELEMENTS OF DECEIT:
- Partition memory into "Pages" (2K-4K-8K)
- MAP a few to RAM, others to DISK
- Keep "HOT" pages in RAM.

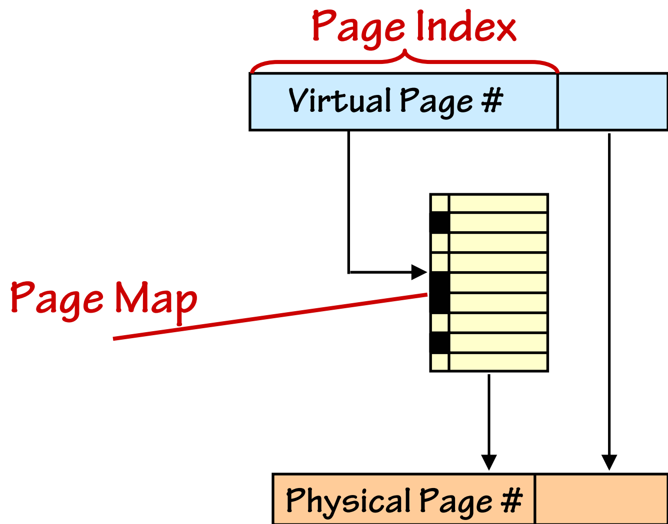


Demand Paging

Basic idea:

- Start with all of VM on DISK ("swap area"), MMU "empty"
- Begin running program… each VA "mapped" to a PA
  - Reference to RAM-resident page: RAM accessed by hardware
  - Reference to a non-resident page: traps to software handler, which
    - Fetches missing page from DISK into RAM
    - Adjusts MMU to map newly-loaded virtual page directly in RAM
    - If RAM is full, may have to replace ("swap out") some little-used page to free up RAM for the new page.
- Working set incrementally loaded, gradually evolves…

Simple Page Map Design



FUNCTION: Given Virtual Address,

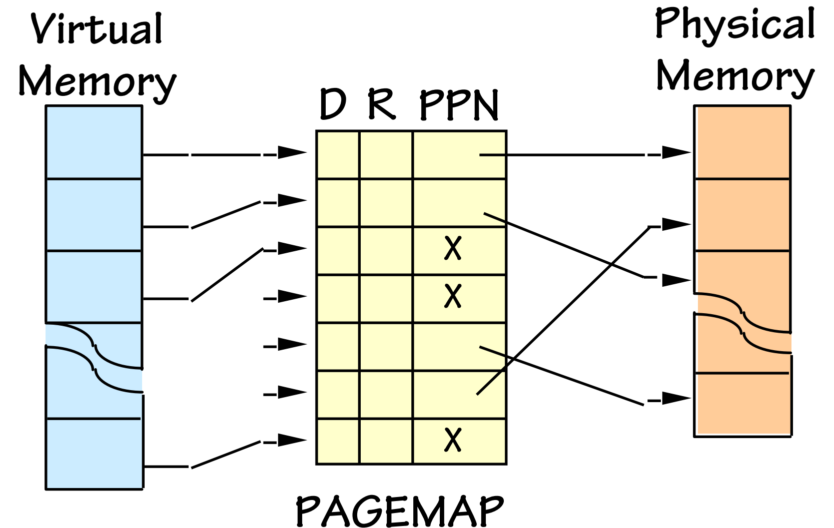
- Map to PHYSICAL address

OR

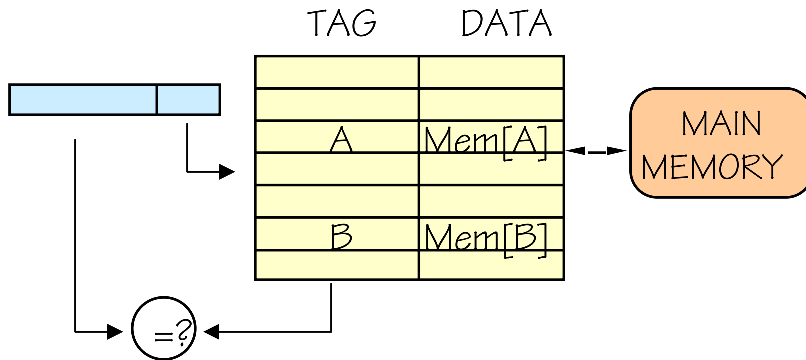
- Cause PAGE FAULT allowing page replacement

Why use HIGH address bits to select page?
... LOCALITY.
Keeps related data on same page.

Why use LOW address bits to select cache line?
... LOCALITY.
Keeps related data from competing for same cache lines.

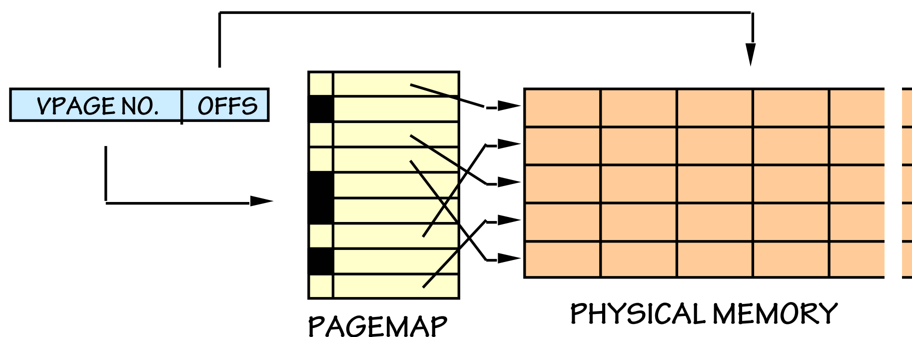


Virtual Memory vs. Cache



Cache:

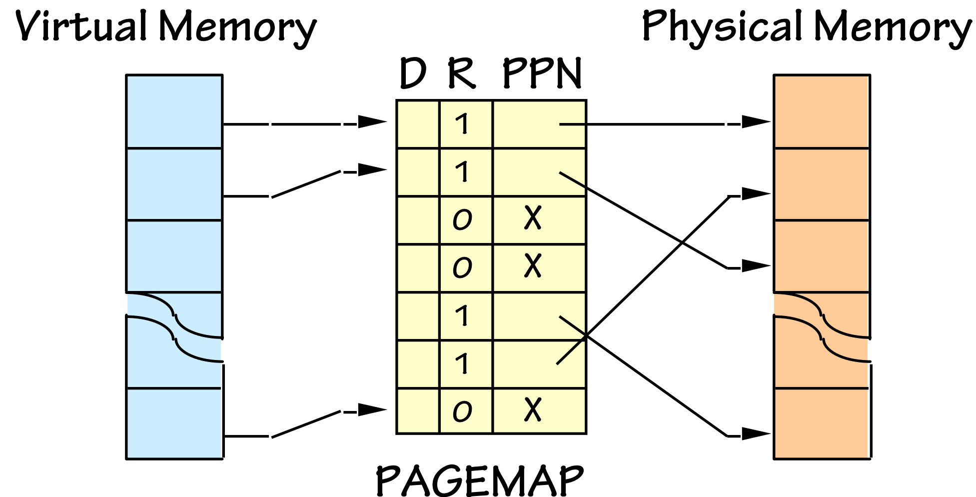
- Relatively short blocks
- Few lines: scarce resource
- miss time: 3x-20x hit times



Virtual memory:

- disk: long latency, fast xfer
  - → miss time: ~10^5 x hit time
  - → write-back essential!
  - → large pages in RAM
- lots of lines: one for each page
- tags in page map, data in physical memory

Virtual Memory: the VI-1 view

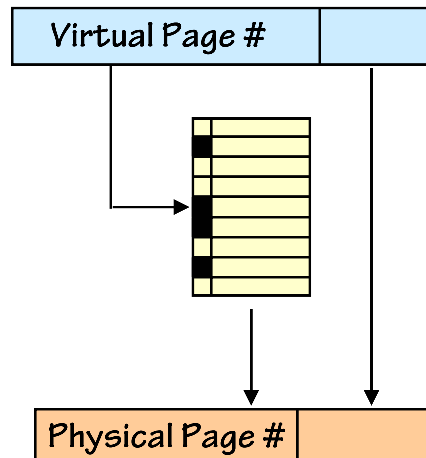


Pagemap Characteristics:

- One entry per virtual page!
- RESIDENT bit = 1 for pages stored in RAM, or 0 for non-resident (disk or unallocated). Page fault when R = 0.
- Contains PHYSICAL page number (PPN) of each resident page
- DIRTY bit says we've changed this page since loading it from disk (and therefore need to write it to disk when it's replaced)

Virtual Memory: the VI-3 view

Problem: Translate
VIRTUAL ADDRESS
to PHYSICAL ADDRESS




```
int VtoP(int VPageNo,int PO) {
 if (R[VPageNo] == 0)
    PageFault(VPageNo);
 return (PPN[VPageNo] << p) | PO;
}

/* Handle a missing page... */
void PageFault(int VPageNo) {
  int i;

  i = SelectLRUPage();
  if (D[i] == 1)
    WritePage(DiskAdr[i],PPN[i]);
  R[i] = 0;

  PPN[VPageNo] = PPN[i];
  ReadPage(DiskAdr[VPageNo],PPN[i]);
  R[VPageNo] = 1;
  D[VPageNo] = 0;
}
```

The HW/SW Balance

IDEA:

- *devote* HARDWARE to high-traffic, performance-critical path
- use (slow, cheap) SOFTWARE to handle *exceptional cases*

hardware

```
int VtoP(int VPageNo,int PO) {
 if (R[VPageNo] == 0)PageFault(VPageNo);
 return (PPN[VPageNo] << p) | PO;
}
```

software

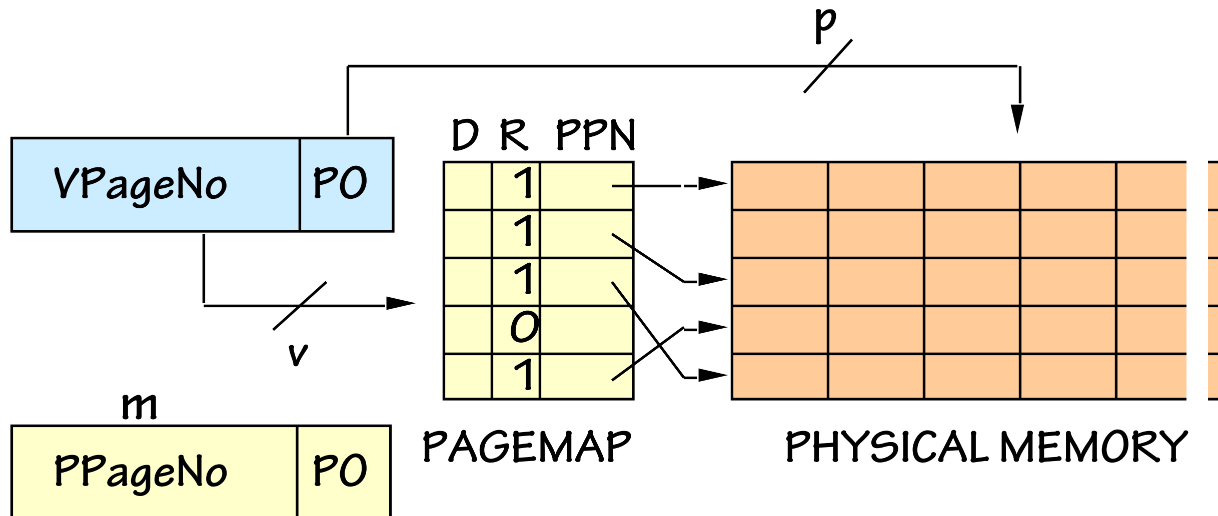
```
/* Handle a missing page... */
void PageFault(int VPageNo) {
  int i = SelectLRUPage();
  if (D[i] == 1) WritePage(DiskAdr[i],PPN[i]);
  R[i] = 0;

  PA[VPageNo] = PPN[i];
  ReadPage(DiskAdr[VPageNo],PPN[i]);
  R[VPageNo] = 1;
  D[VPageNo] = 0;
}
```

HARDWARE performs address translation, detects page faults:

- running program interrupted ("suspended");
- PageFault(…) is forced;
- On return from PageFault; running program continues

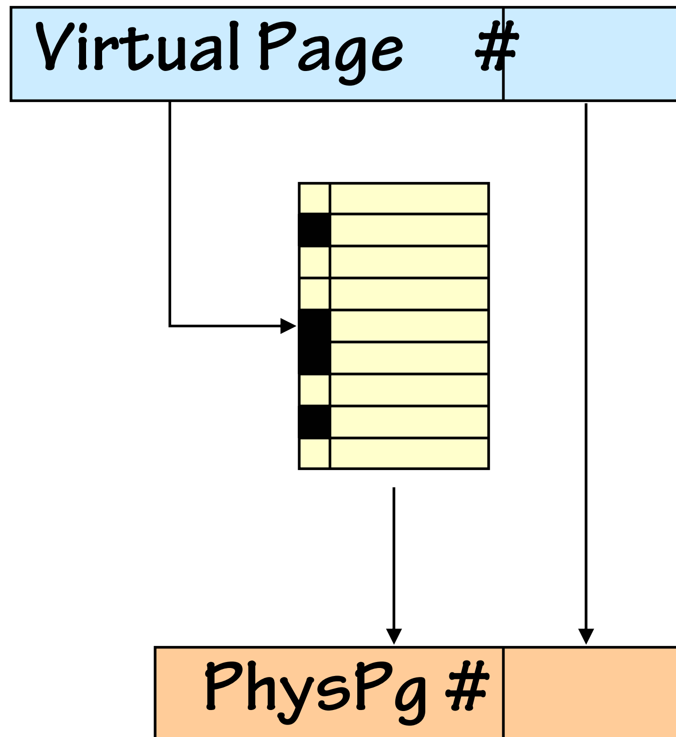
Page Map Arithmetic



$(v + p)$	bits in virtual address
$(m + p)$	bits in physical address
2^v	number of VIRTUAL pages
2^m	number of PHYSICAL pages
2^p	bytes per physical page
2^{v+p}	bytes in virtual memory
2^{m+p}	bytes in physical memory
$(m+2)2^v$	bits in the page map

Typical page size: 1K – 8K bytes
Typical (v+p): 32 (or more) bits
Typical (m+p): 26 – 30 bits
(64 – 1024 MB)

Example: Page Map Arithmetic



SUPPOSE...

- 32-bit Virtual address
- 2^{12} page size (4 KB)
- 2^{30} RAM max (1 GB)

THEN:

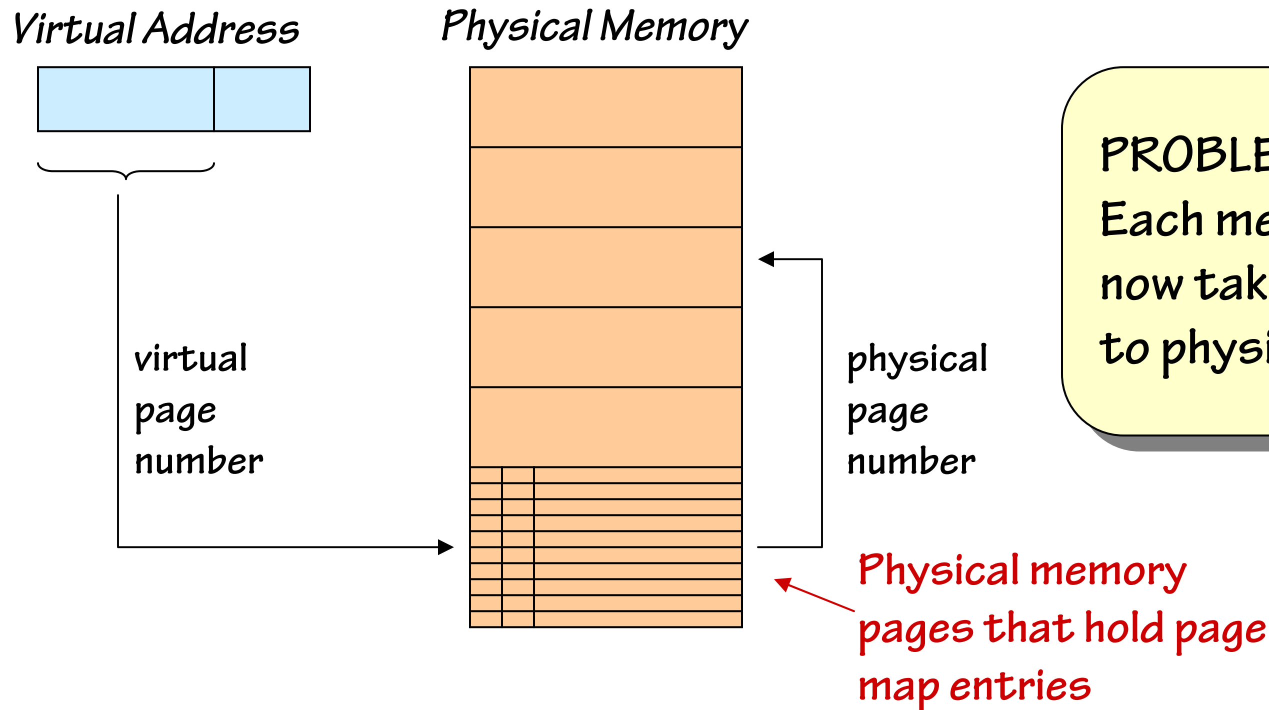
- # Physical Pages = $2^{18} = 256K$
- # Virtual Pages = 2^{20}
- # Page Map Entries = $2^{20} = 1M$
- # Bits In pagemap = $20*2^{20} \cong 20M$

Use SRAM for page map??? OUCH!

RAM-Resident Page Maps

SMALL page maps can use dedicated RAM… gets expensive for big ones!

SOLUTION: Move page map to MAIN MEMORY:

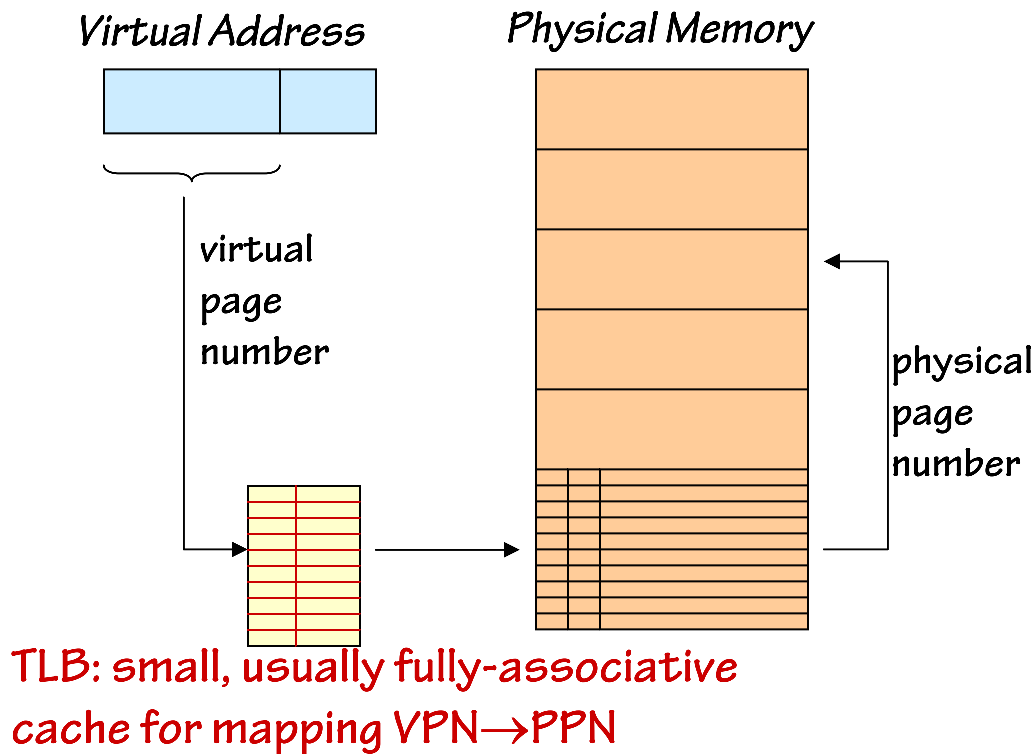


PROBLEM:
Each memory references
now takes 2 accesses
to physical memory!

Translation Look-aside Buffer (TLB)

PROBLEM: 2x performance hit… *each memory reference now takes 2 accesses!*

SOLUTION: *CACHE the page map entries*



TLB: small, usually fully-associative cache for mapping VPN→PPN

IDEA:
LOCALITY in memory reference patterns → SUPER locality in reference to page map

VARIATIONS:
- sparse page map storage
- paging the page map

Example: mapping VAs to PAs

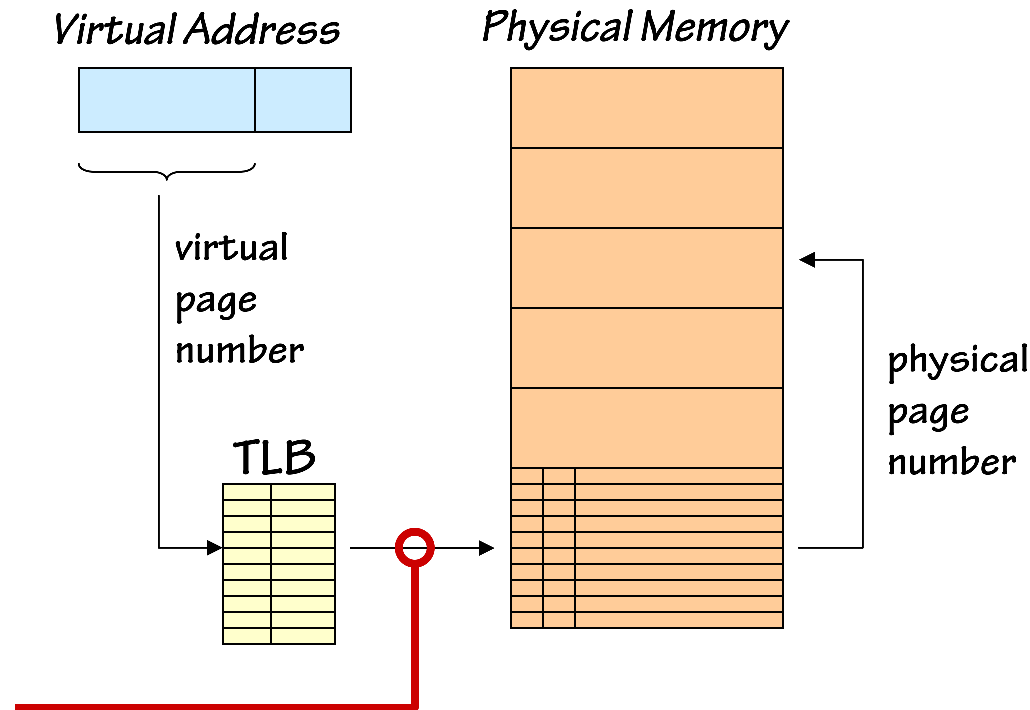
Suppose

- virtual memory of 2^{32} bytes
- physical memory of 2^{24} bytes
- page size is 2^{10} (1 K) bytes

```
VPN | R D PPN
----+--------
 0  | 0 0  7
 1  | 1 1  9
 2  | 1 0  0
 3  | 0 0  5
 4  | 1 0  5
 5  | 0 0  3
 6  | 1 1  2
 7  | 1 0  4
 8  | 1 0  1
 …
```

1. How many pages can be stored in physical memory at once?
2. How many entries are there in the page table?
3. How many bits are necessary per entry in the page table? (Assume each entry has PPN, resident bit, dirty bit)
4. How many pages does the page table require?
5. What's the largest fraction of VM that might be resident?
6. A portion of the page table is given to the left. What is the physical address for virtual address 0x1804?

Optimizing Sparse Page Maps



On TLB miss:

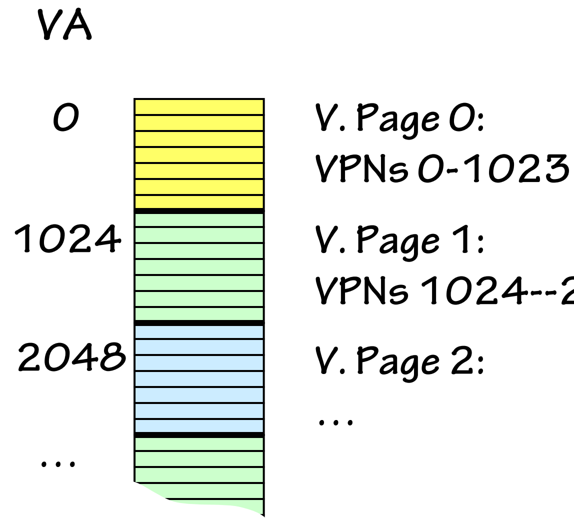
- look up VPN in "sparse" data structure (e.g., a list of VPN-PPN pairs)
- use hash coding to speed search
- only have entries for ALLOCATED pages
- allocate new entries "on demand"
- time penalty? LOW if TLB hit rate is high…

Should we do this in HW or SW?

Moving page map to DISK...

Given HUGE virtual memory, even storing pagemap in RAM may be too expensive...

... seems like we could store little-used parts of it on the disk.



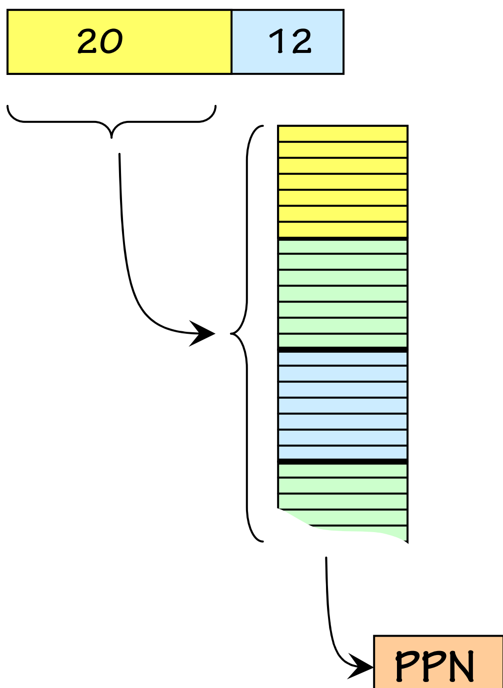
SUPPOSE we store page map in virtual memory

- starting at (virtual) address 0
- 4 bytes/entry
- 4KB/page

Then there's ________ entries per page of pagemap... $2^{10} = 1024$

Pagemap entry for VP v is stored at virtual address v*4

Multi-level Page Map – 6-3 View



VFetch(VA) =

Split VA into 20-bit VPN plus 12-bit Offset

If VPN is 0 then return Mem[Offset]

PageMap entry = VFetch(VPN*4)

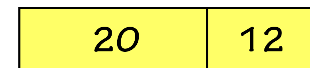
PPN = low 18 bits of PageMap entry

PA = PPN concatenated with Offset

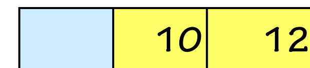
Return Mem[PA]

Recursion:

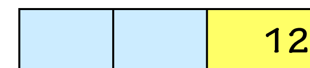
VFetch(32-bit adr A) =>



VFetch(22-bit adr VPN[A] * 4)



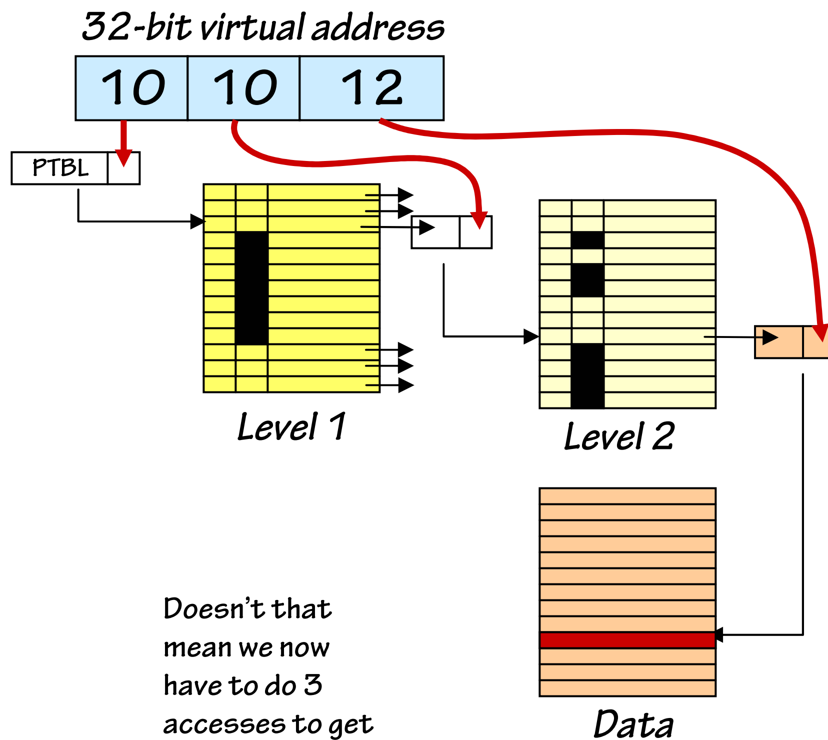
VFetch(12-bit adr VPN[...]*4



from VPN 0

- VP 0 wired to PA 0
- VP 0 can't be paged – terminates recursion
- Recursion depth ≤ 2

Multi-level Page Map – 6-1 View



We've built a 2-level pagemap:

Each VA breaks down into

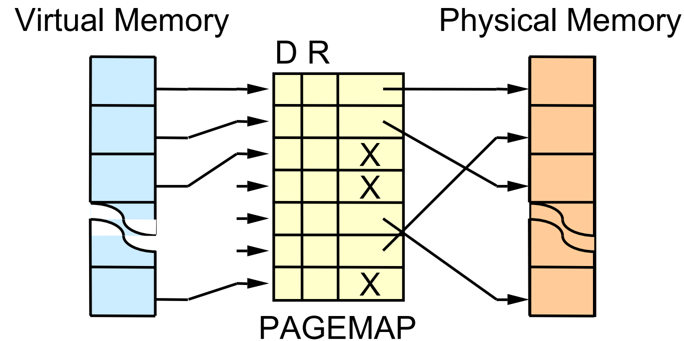
- High 10 bits: addresses an entry in "root" page of pagemap entries – always resident.
- Next 10 bits: addresses an entry in the addressed 2nd level page (which may be non-resident).

Tricks:

- Most of pagemap non-resident
- Use TLBs to eliminate most pagemap accesses

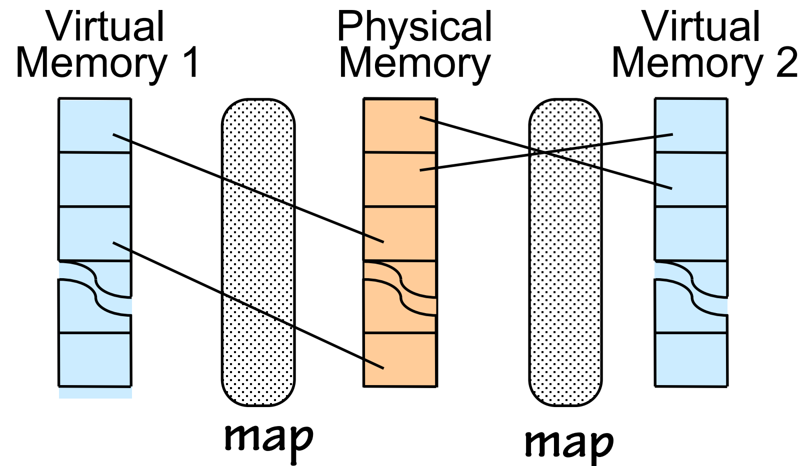
Contexts

A *context* is a mapping of VIRTUAL to PHYSICAL locations, as dictated by contents of the page map:

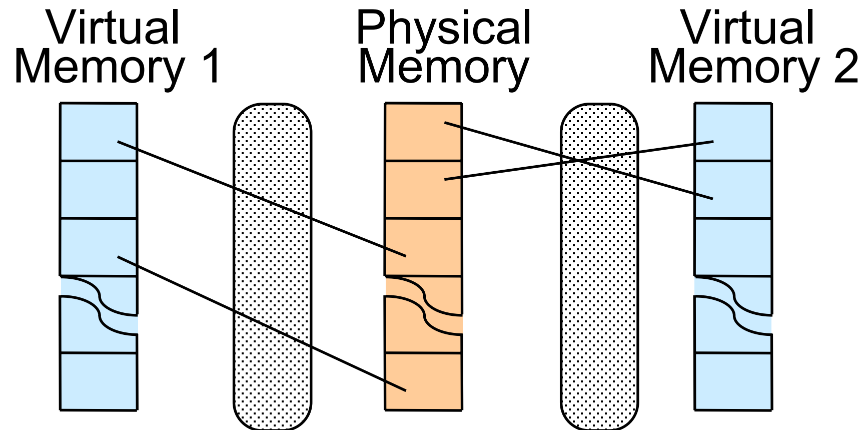


Several programs may be simultaneously loaded into main memory, *each in its separate context:*

"Context switch":
reload the page map!



Contexts: A Sneak Preview



Every application can be written as if it has access to all of memory, without considering where other applications reside.

First Glimpse at a VIRTUAL MACHINE

1. TIMESHARING among several programs --
   - Separate context for each program
   - OS loads appropriate context into pagemap when switching among pgms

2. Separate context for OS "Kernel" (eg, interrupt handlers)...
   - "Kernel" vs "User" contexts
   - Switch to Kernel context on interrupt;
   - Switch back on interrupt return.

   HARDWARE SUPPORT: 2 HW pagemaps

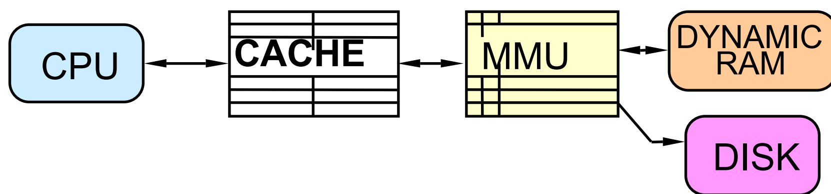
Using Caches with Virtual Memory

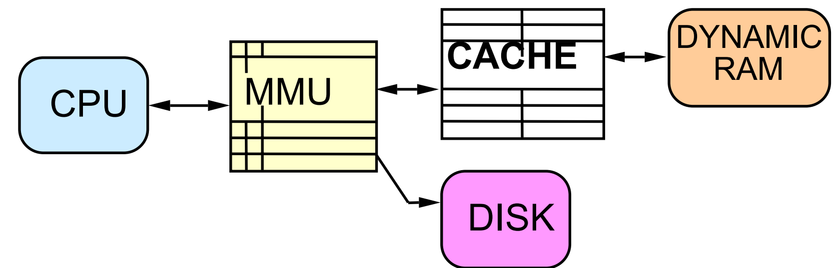
Virtual Cache

Tags match virtual addresses

Physical Cache

Tags match physical addresses

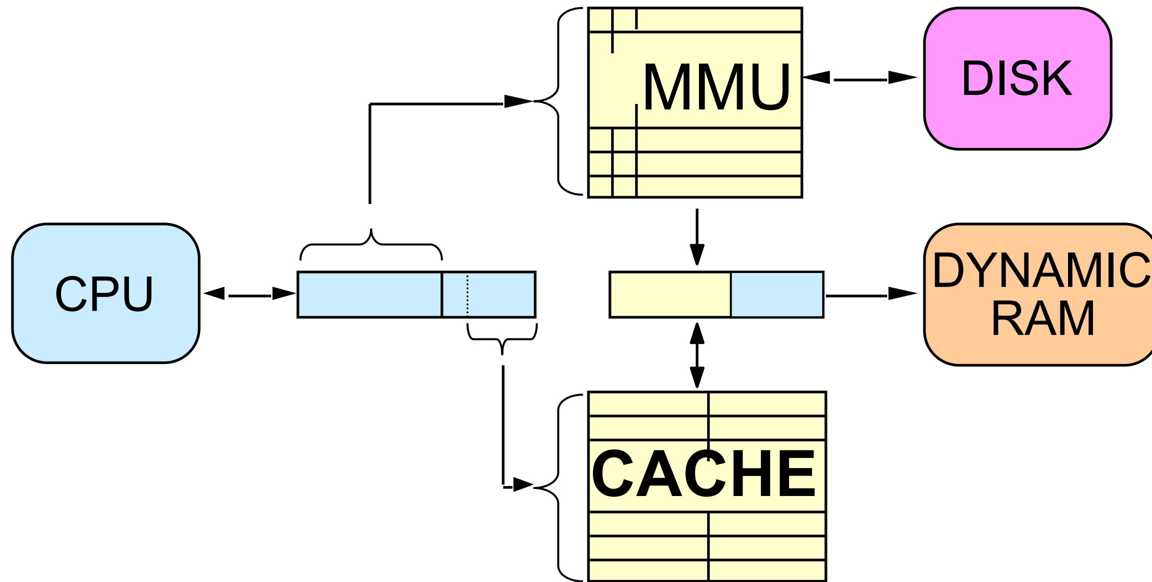




- Problem: cache invalid after context switch
- FAST: No MMU time on HIT

- Avoids stale cache data after context switch
- SLOW: MMU time on HIT

Best of both worlds



OBSERVATION: If cache line selection is based on *unmapped* page offset bits, RAM access in a physical cache can *overlap* page map access. Tag from cache is compared with physical page number from MMU.

Want "small" cache index → go with more associativity

Summary

Exploiting locality on a large scale…

- Programmers want a large, flat address space…

  … but they'll use it sparsely, unpredictably!
- Key: Demand Page sparse working set into RAM from DISK
- IMPORTANT: Single-level pagemap, arithmetic, operation…
  - Access loaded pages via fast hardware path
  - Load virtual memory (RAM) on demand: page faults
- Various optimizations…
  - Moving pagemap to RAM, for economy & size
  - Translation Lookaside Buffer (TLB), to regain performance
  - Moving pagemap to DISK (or, equivalently, VM) for economy & size
- Cache/VM interactions: can cache physical or virtual locations

Semantic consequence:

- CONTEXT: a mapping between V and P addresses – we'll see again!