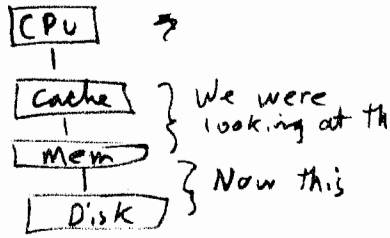
Lecture #25, 26

Virtual Memory



Allow programs to use more memory than actually exists on the machine.

Main memory effectively functions as a cache for secondary storage.

Advantages:

1. Programs can use ~infinite amounts of memory
2. Programs don't have to be aware of amount of physical memory
   - → programs don't have to be rewritten when more memory is added
   - → same program can be run on machines with different amounts of memory
3. Translate virtual addresses to physical addresses so different programs each have their "own" address space
4. Provide protection mechanism

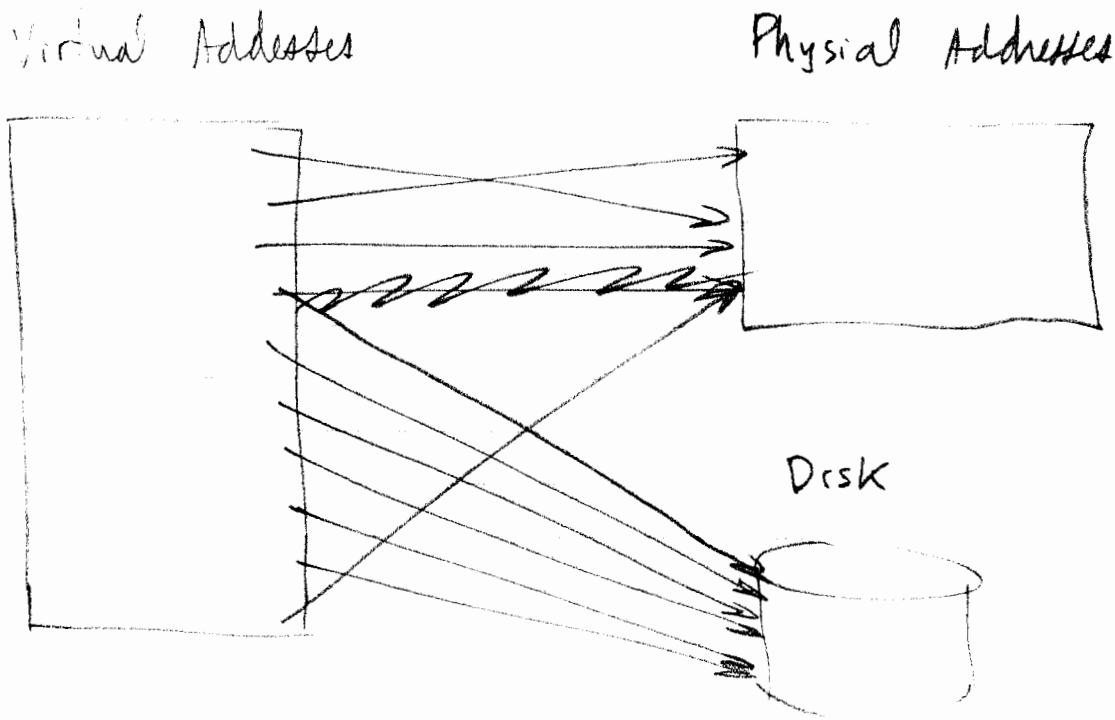
Disadvantages:

1. For it to work, programs must exhibit spatial and/or temporal locality in their memory use (usually not a problem)
2. Memory system becomes significantly more complicated (and possibly slower)

Seymour Cray:
"Virtual Memory is for Weenies"

Conceptually how it works

Memory is divided into pages (different name for block)
If a page is not in main memory, a page fault (cache miss) occurs, and the page must be fetched from disk.

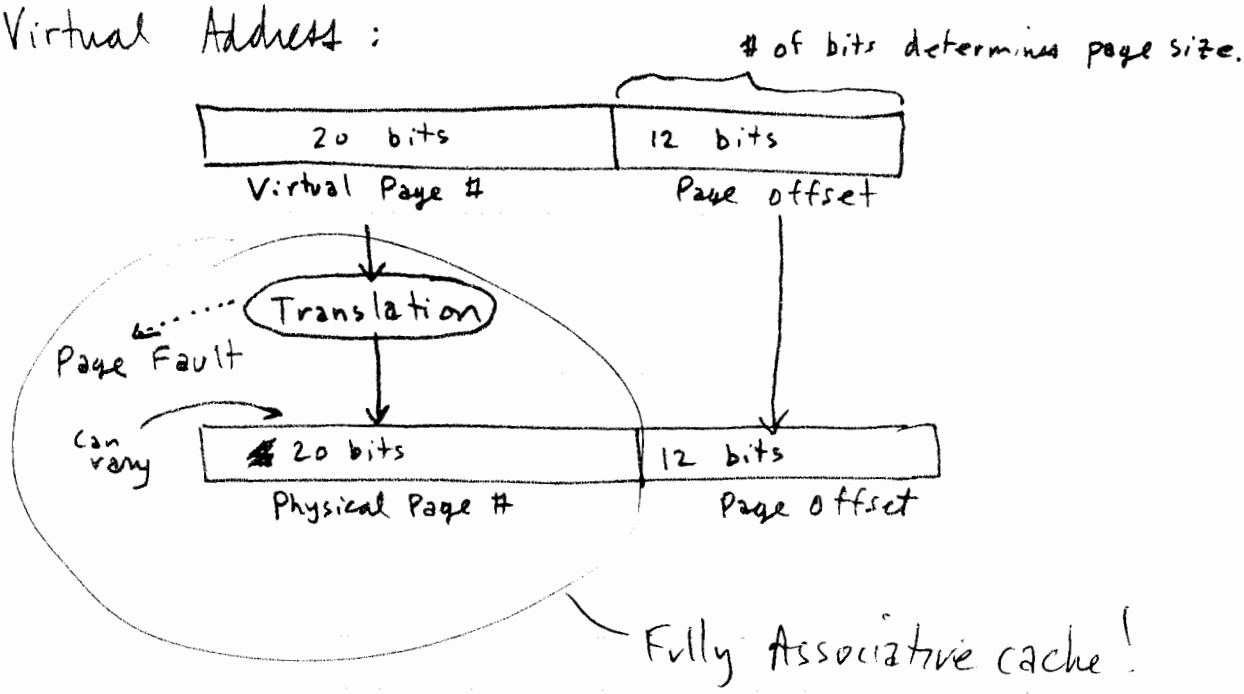


"Memory Mapping" / "Address Translation"

Another advantage: A program's contiguous chunk of memory does not have to be contiguous in physical memory ⇒ minimizes fragmentation problems

More detail:

Virtual Address:



Can have a different translation table for each process

(4)

Gory Details

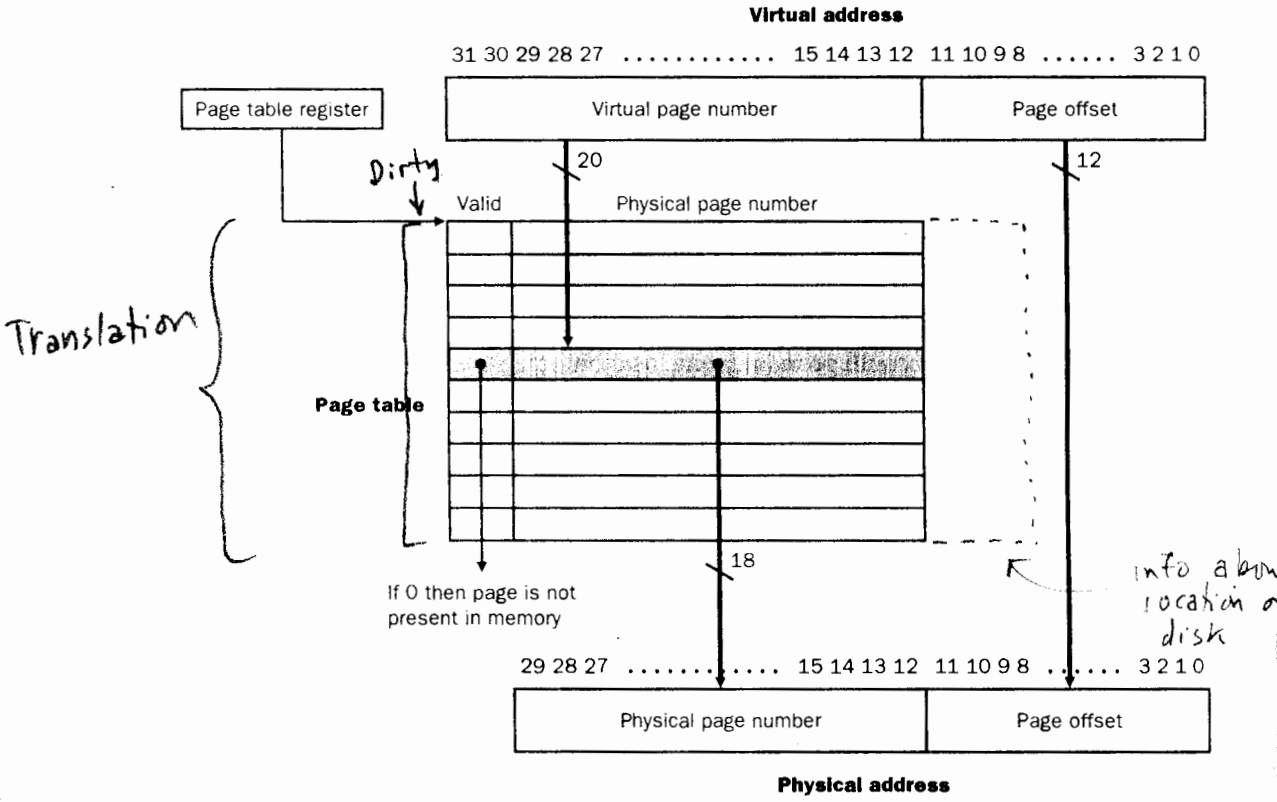


FIGURE 7.16 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address. The starting address of the page table is given by the page table pointer. In this figure, the page size is 2^{12} bytes = 4 KB. The virtual address space is 2^{32} or 4 GigaBytes, and the physical address space is 2^{30} bytes, which allows main memory of up to 1 GigaByte. The number of entries in the page table will be 2^{20} or 1 million entries. The valid bit for each entry indicates whether the mapping is legal. If it is off, then the page is not present in memory.

The Page Table Resides in main memory

Page Table Register: Points to start of page table
(is changed for each process — each process has its own page table)

Valid Bit: Indicates whether or not page is in memory
if yes, use Physical Page #
~~if no, use disk location to retrieve it~~
~~can exception is generated & OS handles it~~
if no, PAGE FAULT (exception → OS)

Key Points about Virtual Memory

- Very high cost for a page fault (a "miss")
- Cost of a miss is almost entirely that of getting 1st word -- ie, dominated by disk head seek time

Results:

(1) Pages should be large — amortize miss time

(2) Use fully-associative mapping of virtual pages to physical pages so that we get maximum use from ~~real~~ physical memory (page table does this)

(3) Use most effective algorithms possible for determing which ~~pages~~ to replace when a page fault occurs — algorithm time is inconsequential in comparison to miss time.
ex: LRU (need reference count, etc.)

(4) Use write back, not write-thru

~~Complications~~:
- Pa

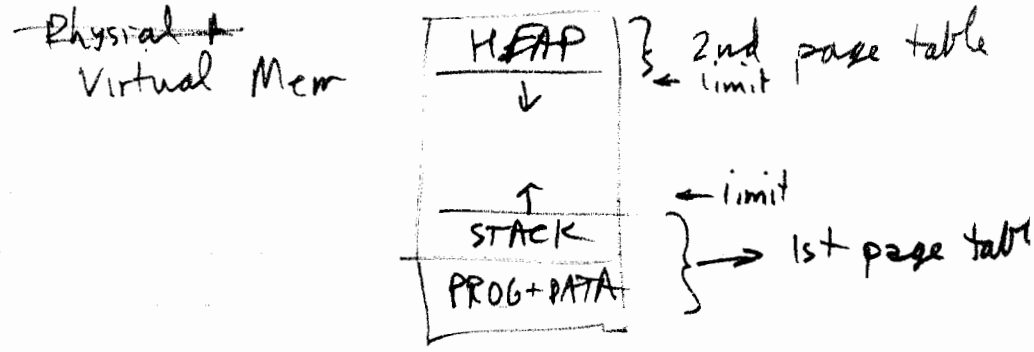
Complications from use of page table:
It's HUGE

32 bit virtual address
4 KB pages
4 bytes / page table entry
} ⇒ 4 MB page table (per process)

Strategies:

① Page out the page table
(obviously this can be messy...)

② Don't have entire page table.
Programs typically only use low & high addresses (heap, stack + prog), so just keep high & low portions of page table



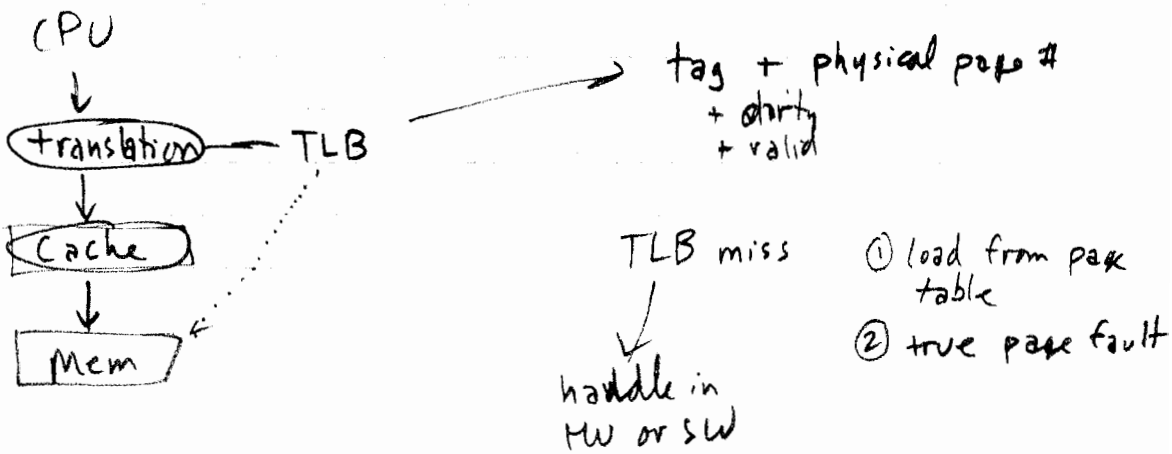
So, now we can fit page tables (mostly) in memory.
But, we still need two physical mem accesses per virtual mem access.
(one to do lookup in page table, 2nd to actually read)
This is bad.

CPU
↓
Translation
↓
cache

Solution:

So, have a special cache just for page table entries ...

It's called a TLB (Translation Lookaside Buffer)
(faster than using main cache)



[Can also put cache before translation, but that ~~this~~ introduces complications] → "virtual cache"
→ aliases - different processes

More detail on page faults

Valid bit in page table is off
→ Exception
- Processor determines page, saves EPC
- ~~carries~~ Decides which page currently in memory to replace
- Writes it if dirty
- Reads new page
- Restarts instruction

} runs another process while these occur.

Protection:

We'd like to be able to prevent processes from writing (and sometimes reading) other processes address space.

→ Don't put entries for unauthorized physical pages in page table

Must Insure that the process can't change its page table,

~~→ no entries~~

But, the O/S must be able to change the page table!

Hardware support:

(1) Two (or more) CPU modes:
- Kernal — O/S
- User

(2) Page table ptr, kernal/user flag can only be changed in kernal mode

(3) Provide mechanism (syscall, int or MIPS) to go from user to kernal mode in a controlled manner. → exception

(4) Page table contains a write bit. If set, process may R/W page. Otherwise, only read it.

→ useful for read-only data (e.g. instructions)
→ protect program from accidental overwrite

Summary: Sources of Cache Misses / Page Faults:

(1) Compulsory: At start up, nothing is in the cache – all accesses miss

(2) Capacity: Cache can't contain all blocks that are needed, so some are discarded & later retrieved

(3) Conflict: In any non fully associative cache, different blocks may want to occupy the same cache index. A block will have to be discarded & later retrieved.