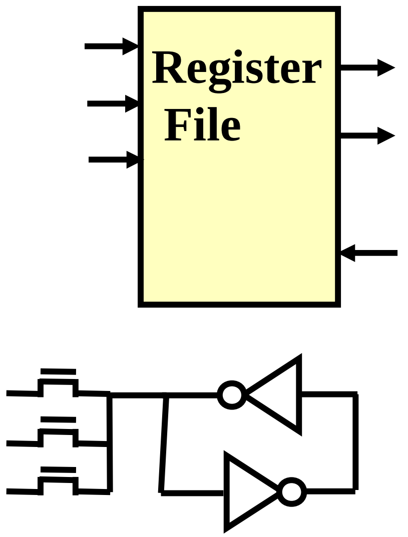
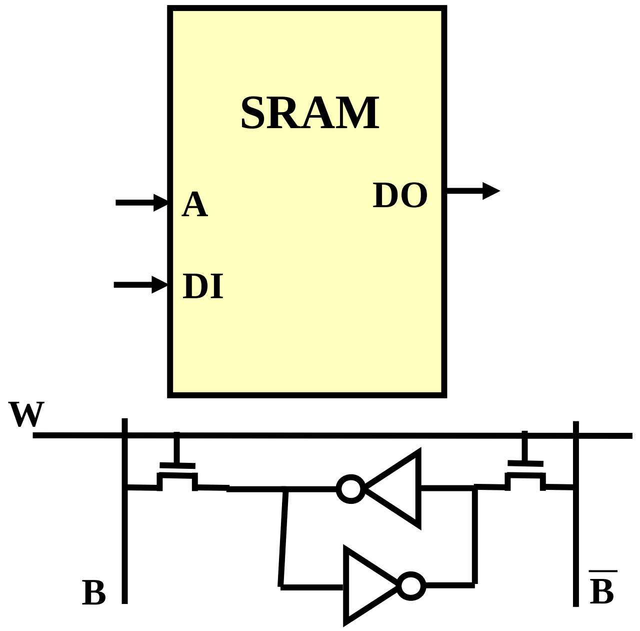
Virtual Memory

Giorgio Richelli

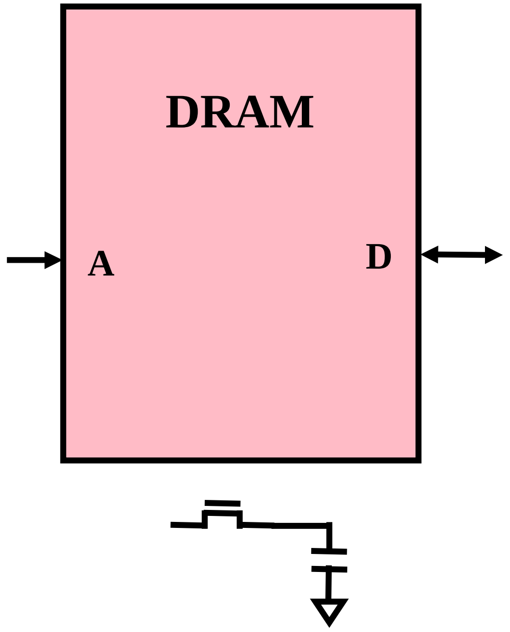
Typical Memory Technologies



- Integrated into CPU
- Fast, many ports

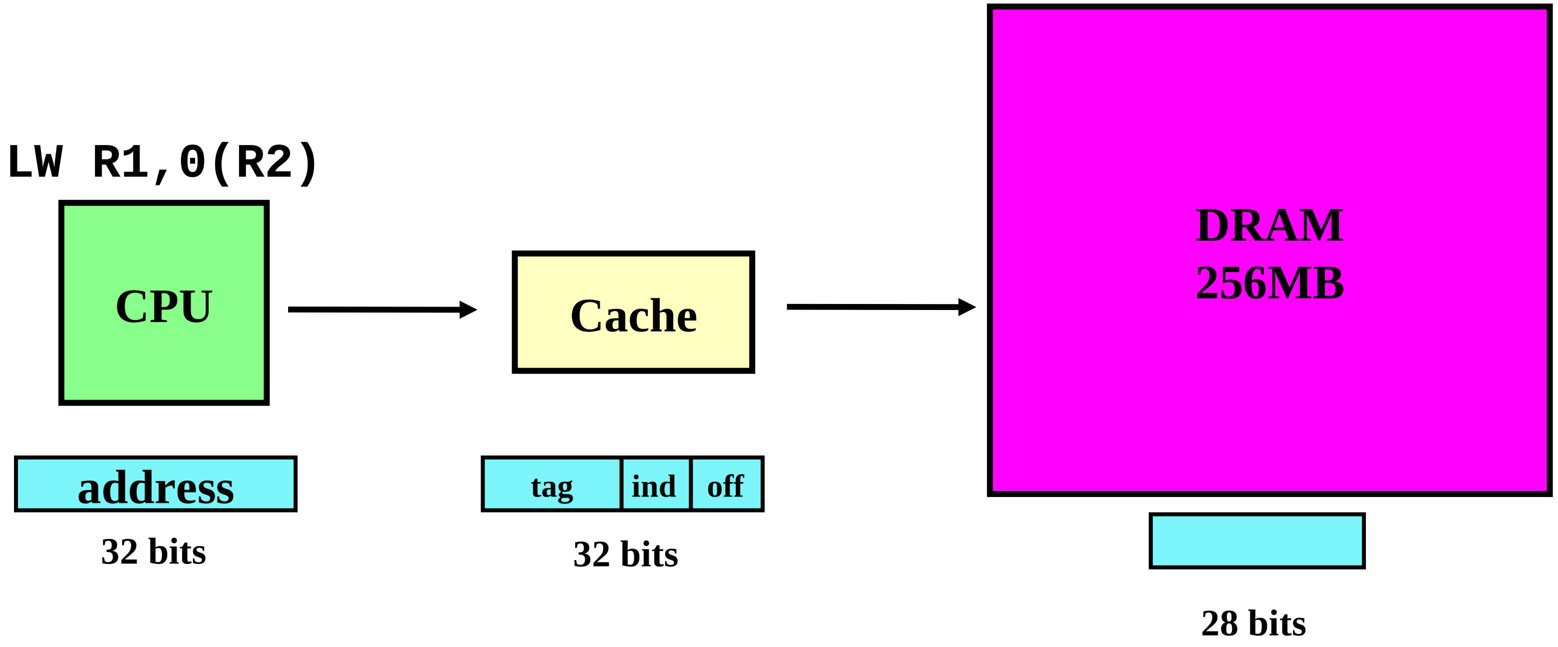


- Caches
  - On chip L1
  - On/off chip L2
  - Off chip L3

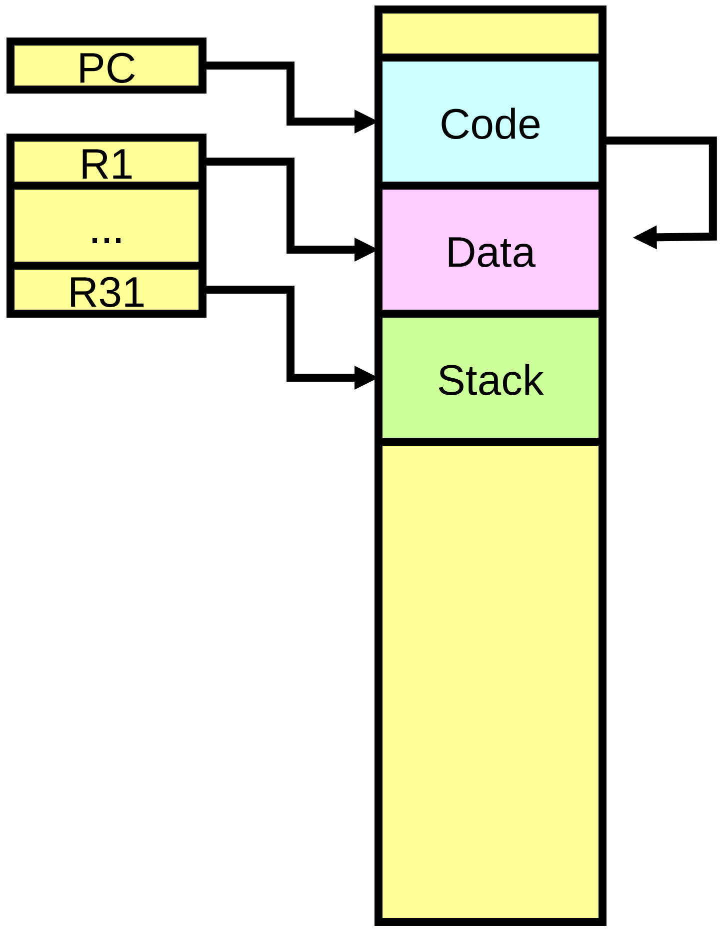


- Main Memory
- Very dense
- Slower to access, one port
- Must be refreshed

Physical Memory Addressing

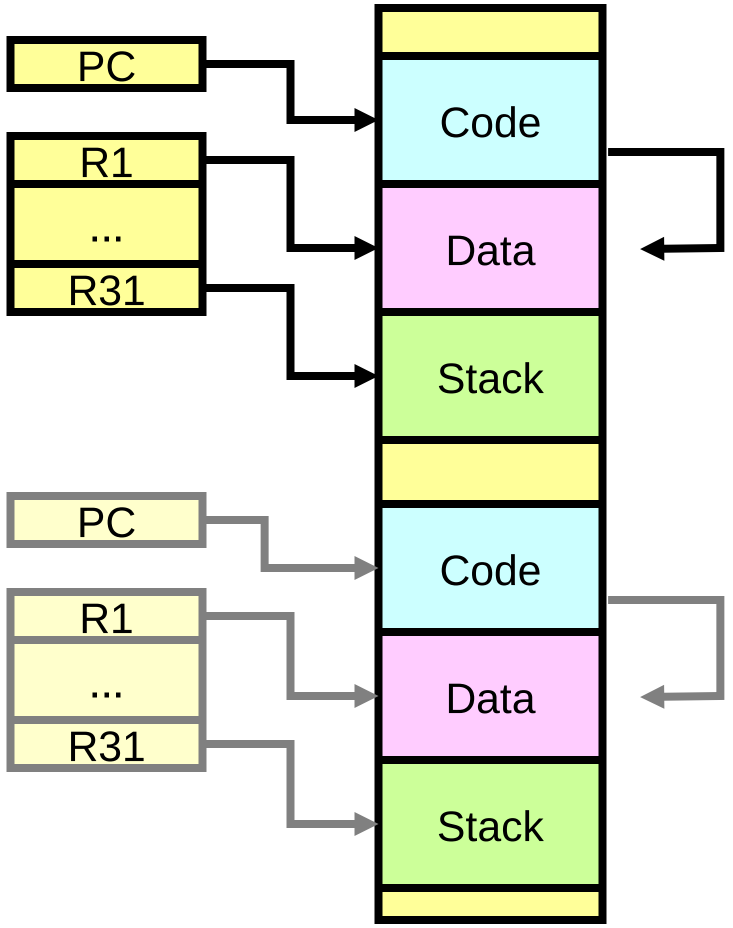


Simple View of Memory



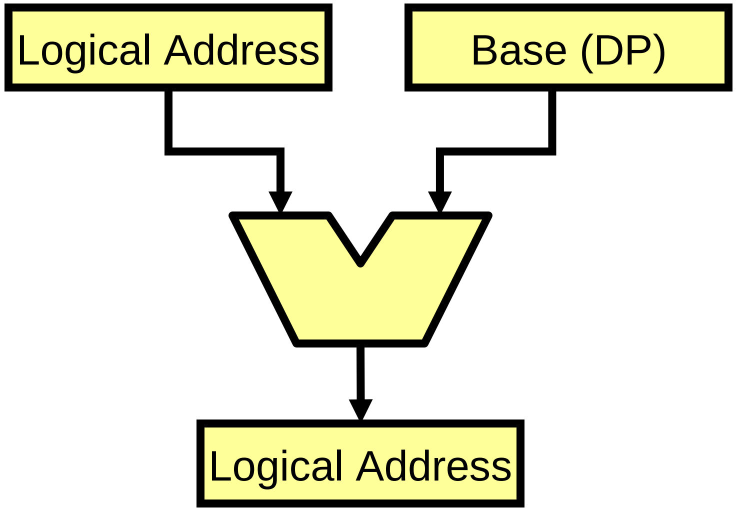
- Single *program* runs at a time
- Code and static data are at fixed locations
  - code starts at fixed location, e.g., 0x100
  - subroutines may be at fixed locations (absolute jumps)
- data locations may be *wired* into code
- Stack accesses relative to stack pointer.

Running Two Programs (Relocation)



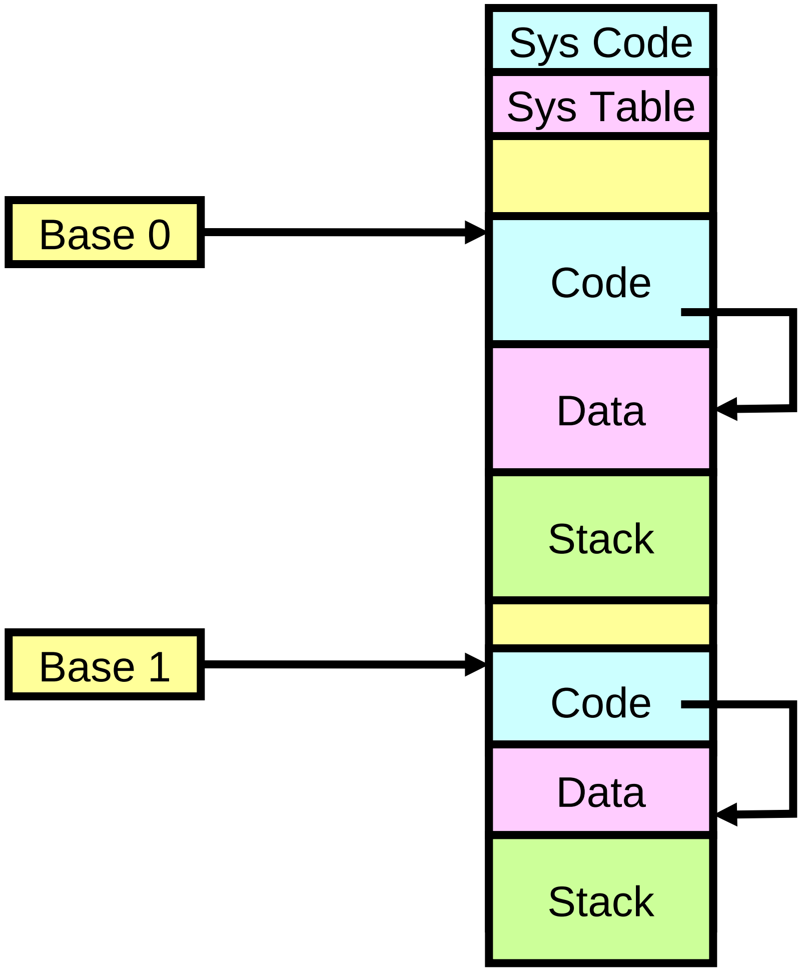
- Need to relocate *logical* addresses to *physical* locations
- Stack is already relocatable
  - all accesses relative to SP
- Code can be made relocatable
  - allow only relative jumps
  - all accesses relative to PC
- Data segment
  - can calculate all addresses relative to a DP
    - expensive
  - faster with hardware support
    - base register

Base-Register Addressing



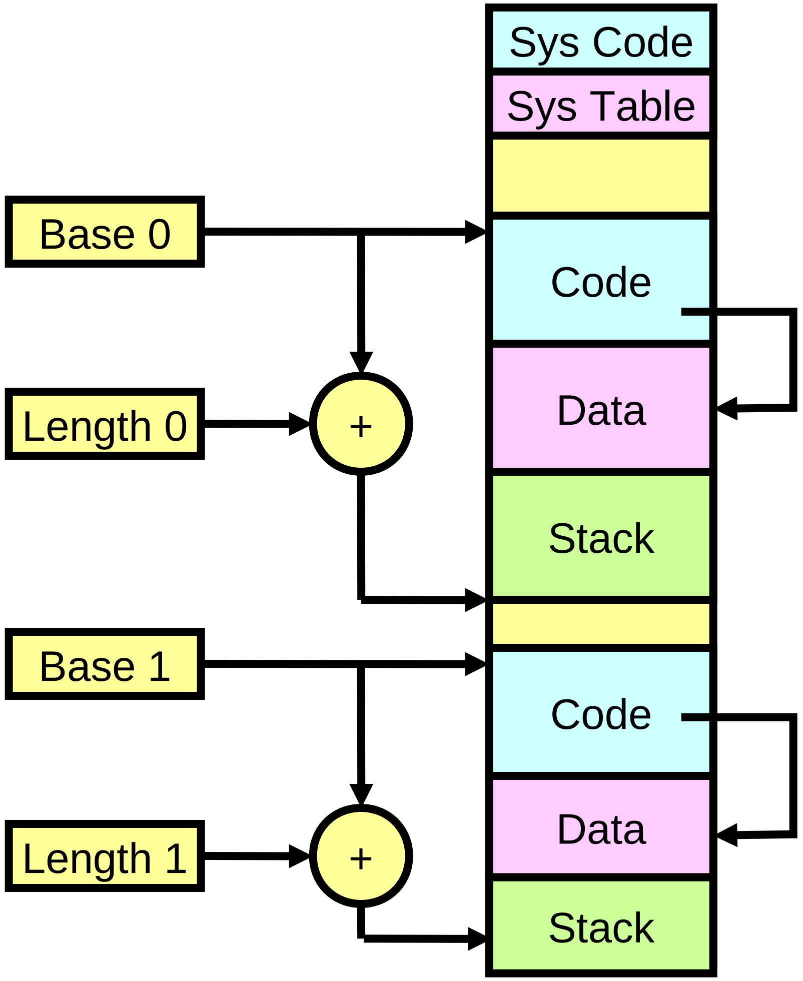
- Add a single base register, BR, to hardware
- Base register loaded with data pointer (DP) for current program
- All data addresses added to base before accessing memory
  - Can relocate code too
- Often implemented with a three-input adder
  - Addr, Offset, Base
- Need to bypass base register to access system tables for program switching

Base Register Addressing



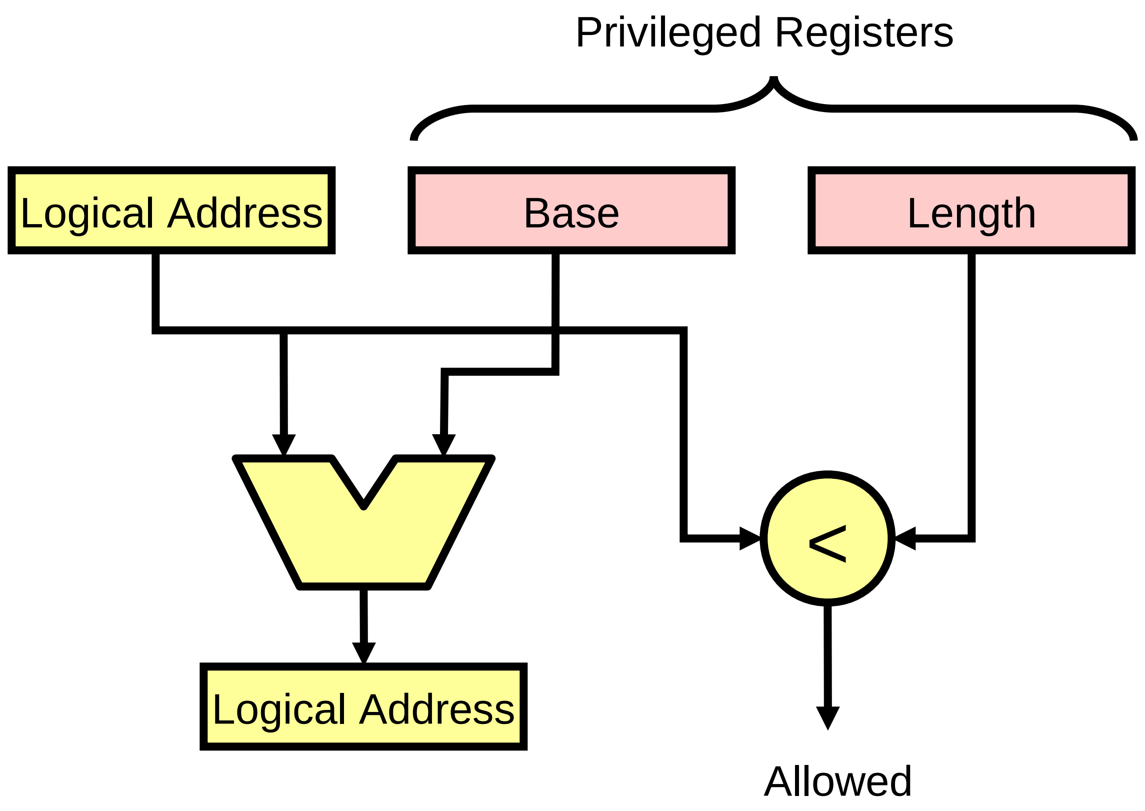
- System code handles switching between programs
- System table contains base address of each program
- Saved state of non-running programs

Providing Protection Between Programs (Length Registers)



- Add a *Length Register* LR to the hardware
- A program is only allowed to access memory from BR to BR+Length-1
- A program cannot set BR or LR
  - they are *privileged registers*

Base + Length Addressing



Issues

A program can be loaded into different places in memory each time it runs

- Relocation

Multiple programs may run concurrently

- Protection

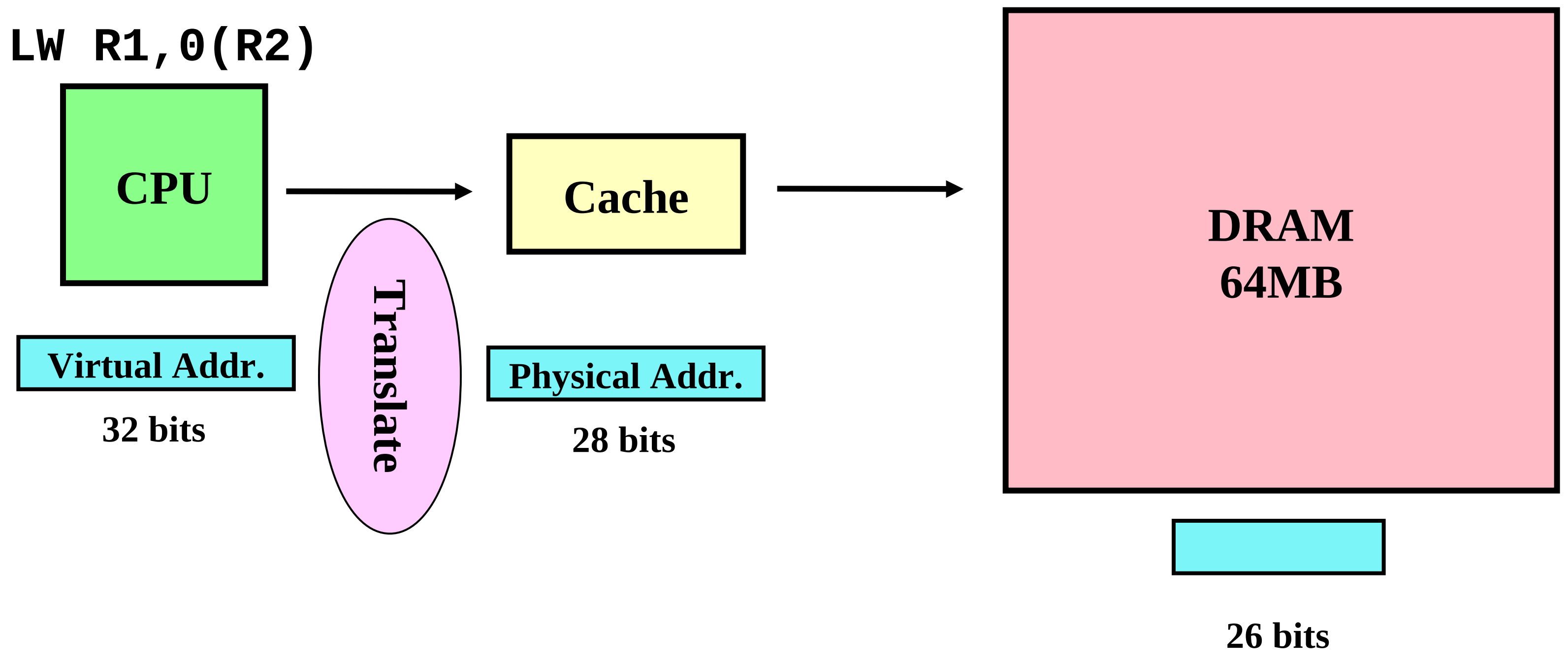
A program may wish to use more memory than physically exists

- Paging

Virtual Memory

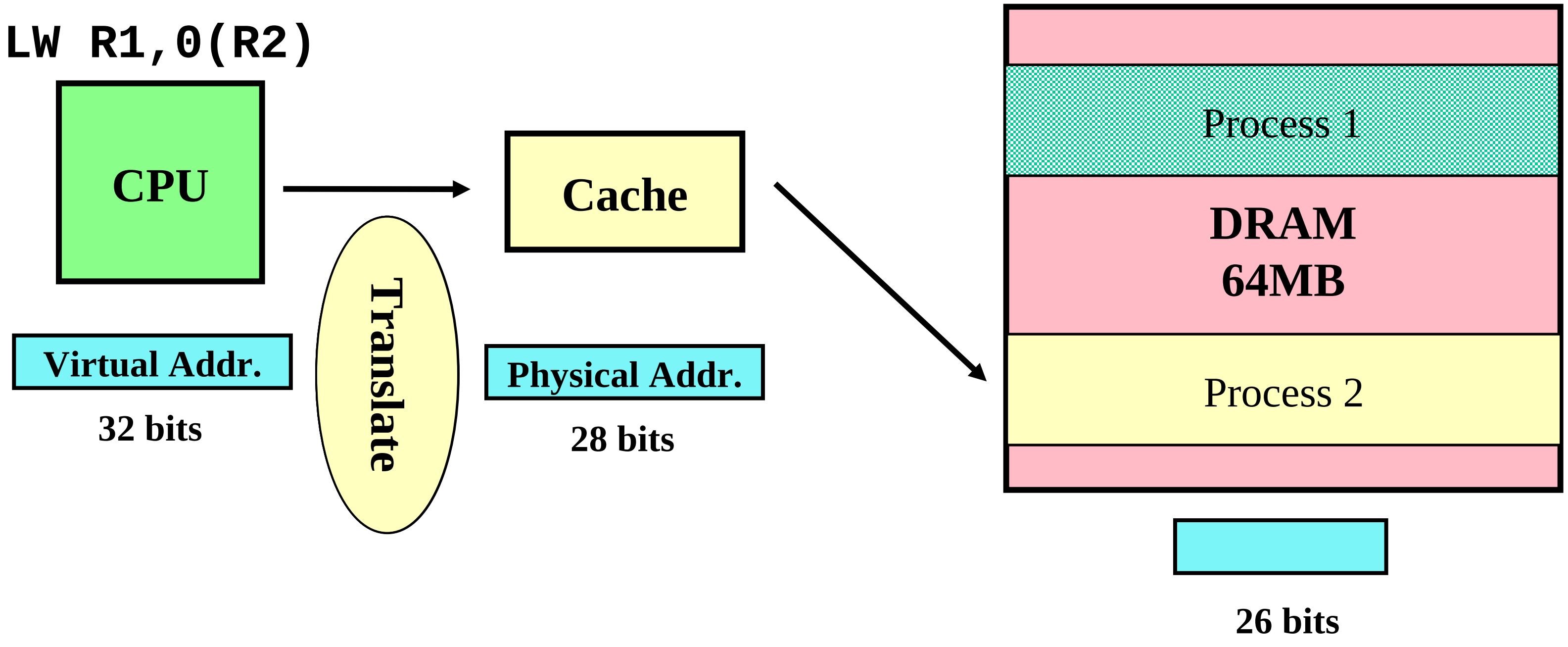
- A technology that permits to:
  - Simplify protection
  - Enable relocation
  - Extend the physical memory capacity

Virtual Memory



- Translate from virtual space to physical space
  - VA ⇒ PA
  - May need to go to disk

Virtual Memory

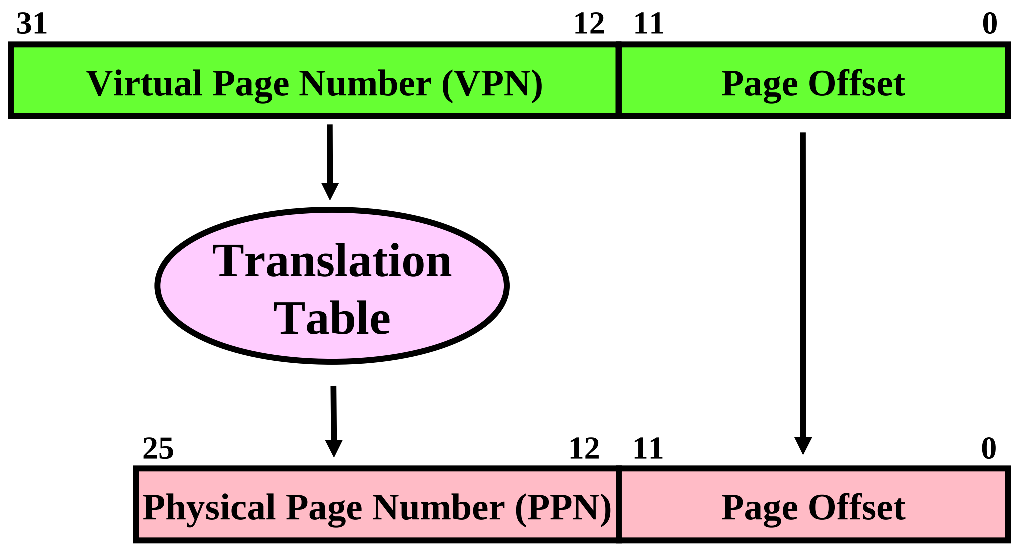


- Both programs can use the same set of addresses
  - Change translation tables to point same VA to different PA for different programs

Paging and Protection

- How to ensure that processes can’t access each other’s data
  - Put them in separate virtual address spaces
  - Control the mappings of VA to PA for each process
    - Separate page tables
- How can you share data between processes
  - Give them each a VA mapping to the same PA
    - Similar entry in each process’ page table

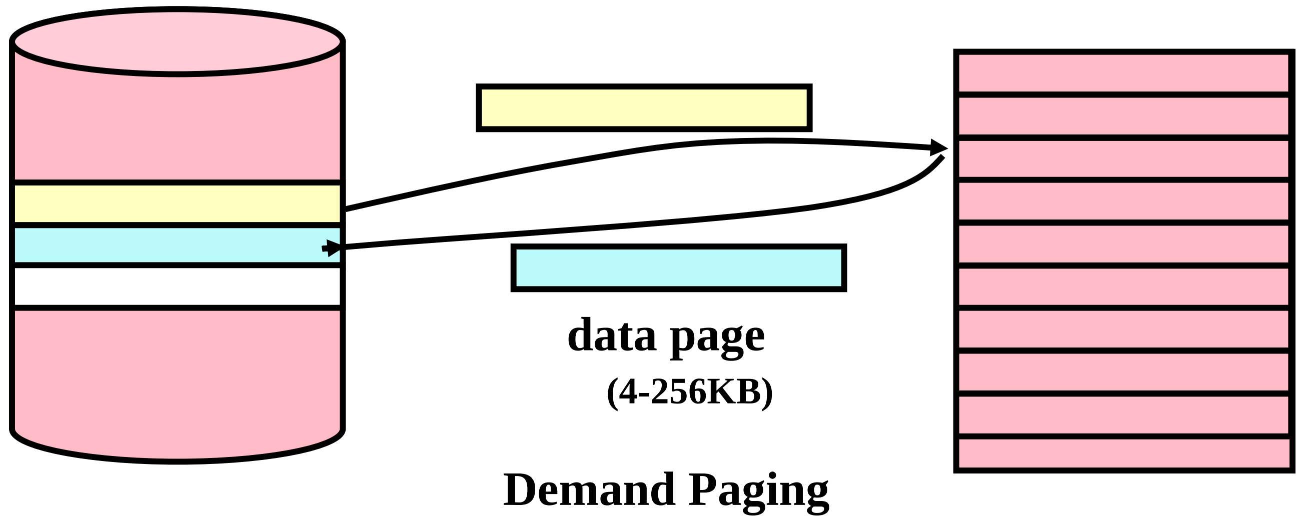
Virtual Address Translation



- Main Memory = 64MB
- Page Size = 4KB
- VPN = 20 bits
- PPN = 14 bits

- Translation table
  - aka "Page Table"

Paging: Main Memory as a Cache for Disk



- 32 bit addresses = 4GB, Main Memory = 64MB
- Dynamically adjust what data stays in main memory
  - Page similar to cache block
- Note: file system >> 4GB, managed by O/S

Virtual Memory Access (+ Fault)

1) Examine page table

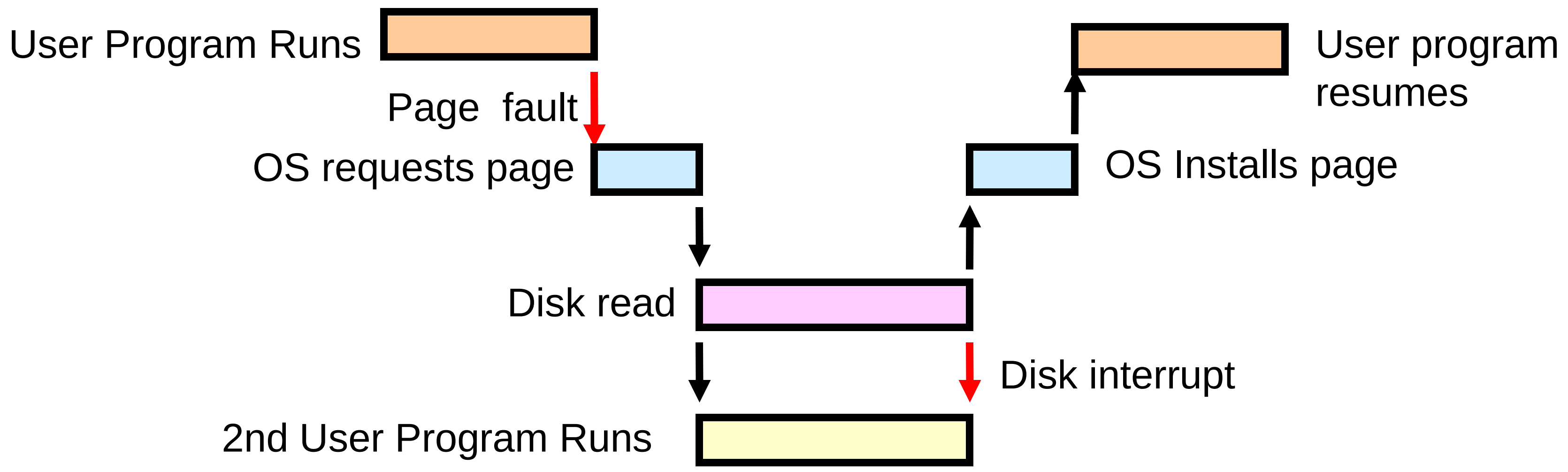
2) Discover that no mapping exists

3) Select page to evict, store back to disk

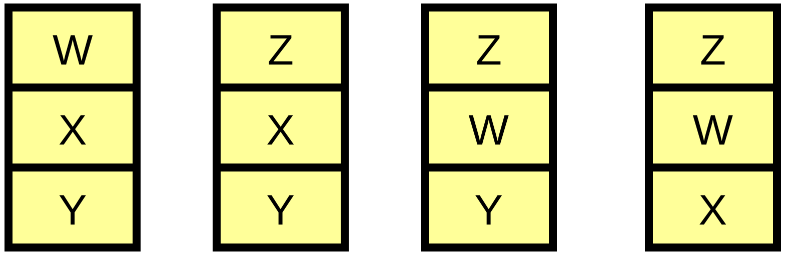
4) Bring in new page from disk

5) Update page table

Page Fault


User Program Runs
Page fault
OS requests page
Disk read
2nd User Program Runs
Disk interrupt
OS Installs page
User program resumes

Page Management and Thrashing



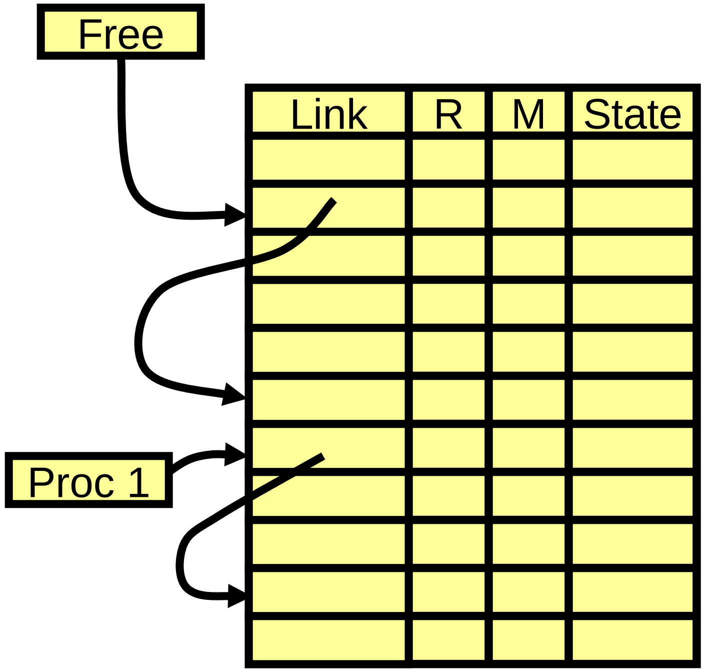
Reference four pages in sequence, mapped to three page frames

- Need to keep a process' *working set* in memory or *thrashing* will occur
- Find working set size by increasing page frame allocation until PF/s falls below limit
- Swap out whole process if insufficient page frames for working set
  - Historically used in Unix Systems

Virtual Memory Requirements

- Restartable (or resumable) instructions
  - must be able to resume program after recovering from a page fault
- Ability to mark a page *not present*
  - and raise a page fault when referencing such a page
- Maintain status bits per page
  - R - referenced - for use by replacement algorithm
  - M - modified - to determine when page is *dirty*

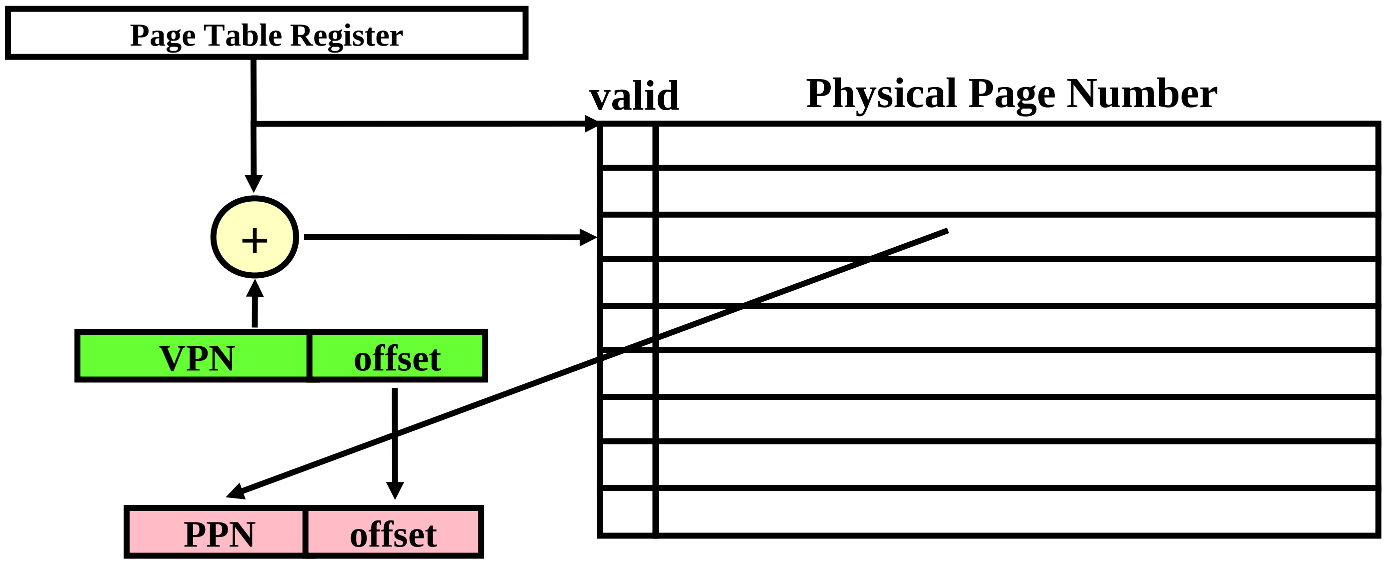
Page Frame Management



Page Frame Table

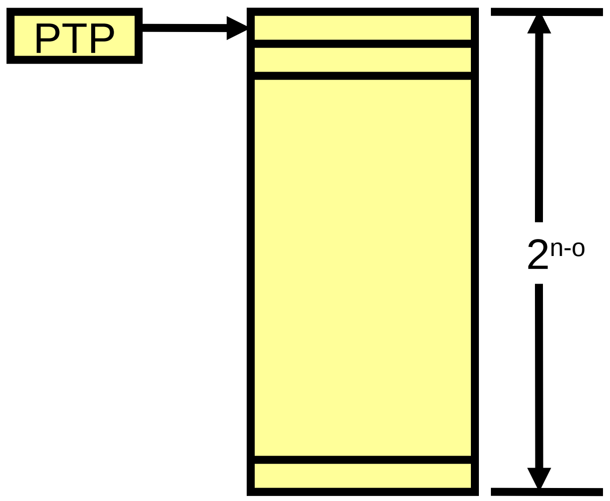
- OS maintains
  - **page table** for each user process
  - **page frame table**
  - free page list
    - pages evicted when number of free pages falls below a *low water mark*.
  - pages evicted using a *replacement policy*
    - random, FIFO, LRU, *clock*
  - if M-bit is clear, need not copy the page back to disk

Page Table Construction



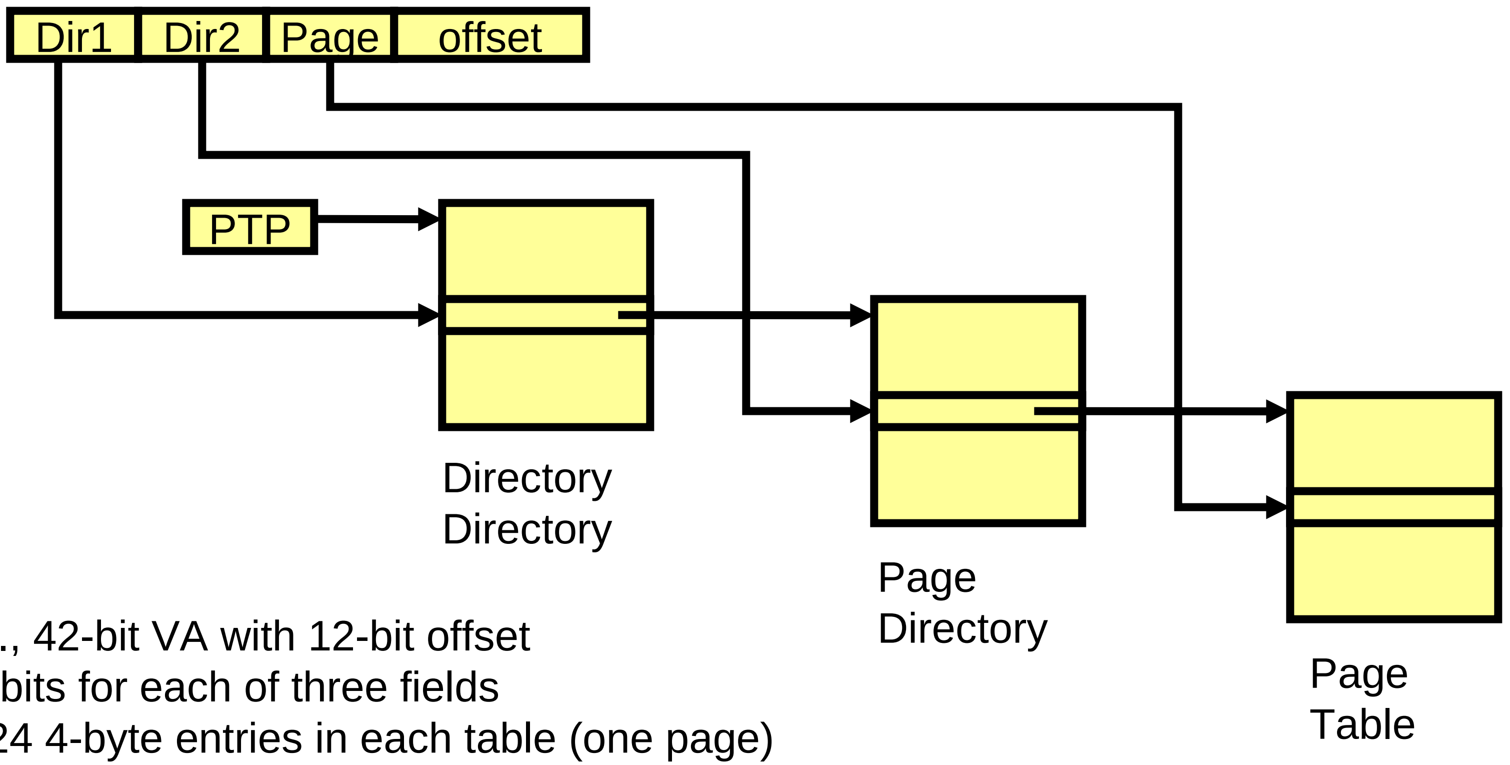
- Page table size
  - (14 + 1) bits => 4 bytes
  - 4 * 2^{20} = 4MB (for 32 bit addresses) …
- Where to put the page table?

Page Table Organization



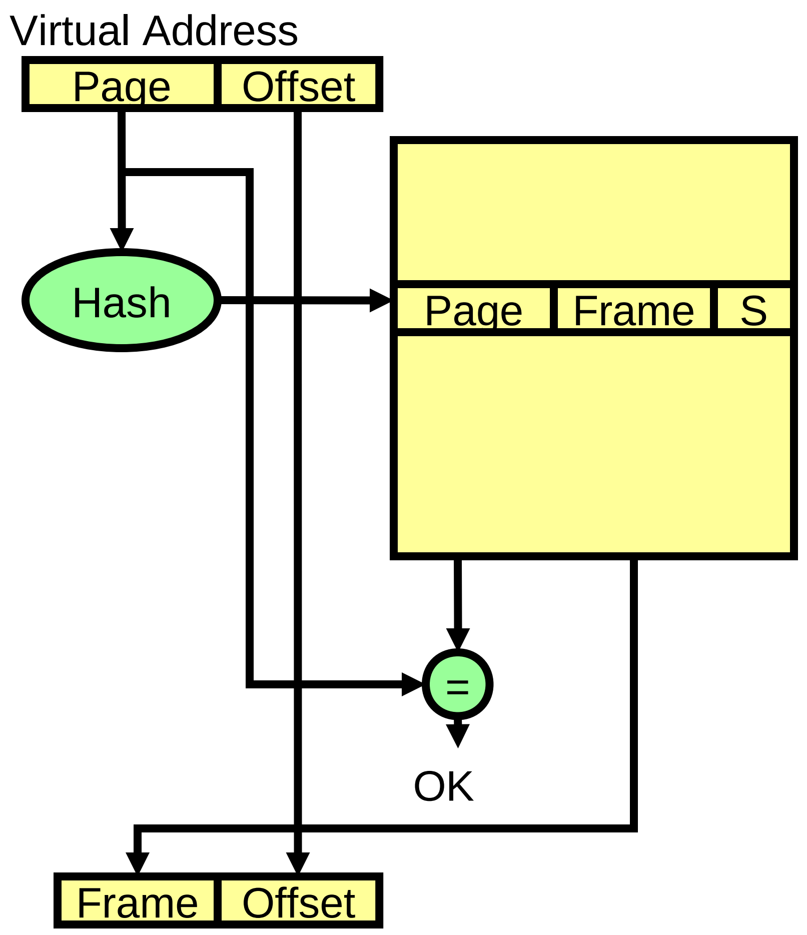
- Flat page table has size proportional to size of *virtual* address space
  - can be very large for a machine with 64-bit addresses and several processes
- Three solutions
  - page the page table
  - multi-level page table (lower levels paged - Tree)
  - inverted page table (hash table)

Multi-Level Page Table



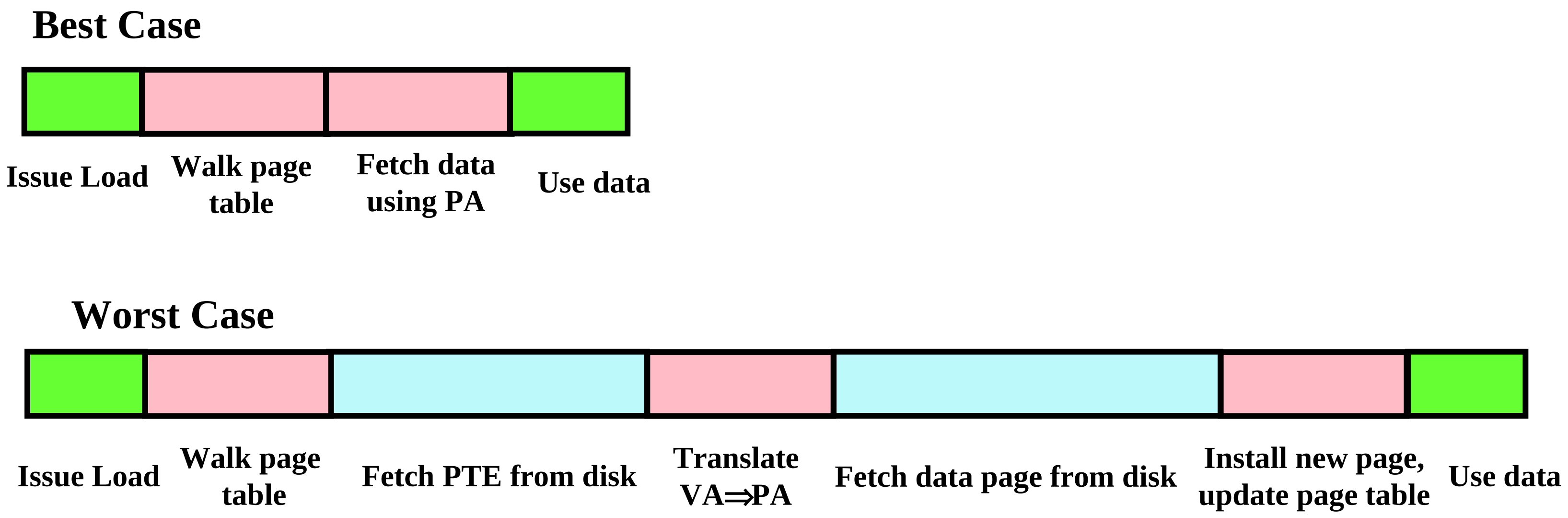
e.g., 42-bit VA with 12-bit offset
10-bits for each of three fields
1024 4-byte entries in each table (one page)

Inverted Page Tables



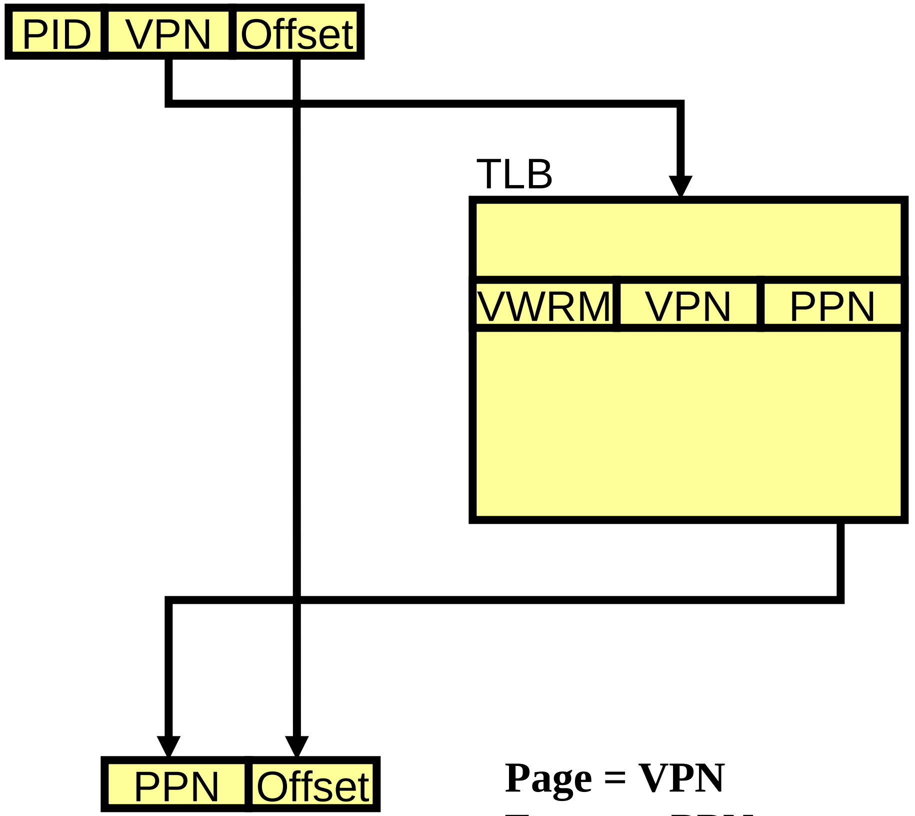
- Store only PTEs for pages *in* physical memory
- Miss in page table implies page is on disk
- Need KP entries for P page frames (usually K > 2)

How Long does it Take to Access VM?



- Problem: Multiple memory (and potentially disk) accesses

Translation Lookaside Buffers



Page = VPN
Frame = PPN

- Store most frequently used translations in small, fast memory (cache for page table entries)
- Valid, Writeable, Referenced, Modified
  - Access protection
  - Replacement strategies
- Size: often 128+ entries
- Highly associative (sometimes fully assoc.)

Behavior in VM system

- TLB Miss
  - Translation is not in TLB – but everything could be in memory
  - Two approaches
    - Hardware state machine *walks* the page table
      - fast but inflexible
    - Exception raised and software walks the page table

- Page Fault
  - Entry not in TLB and target page not in main memory

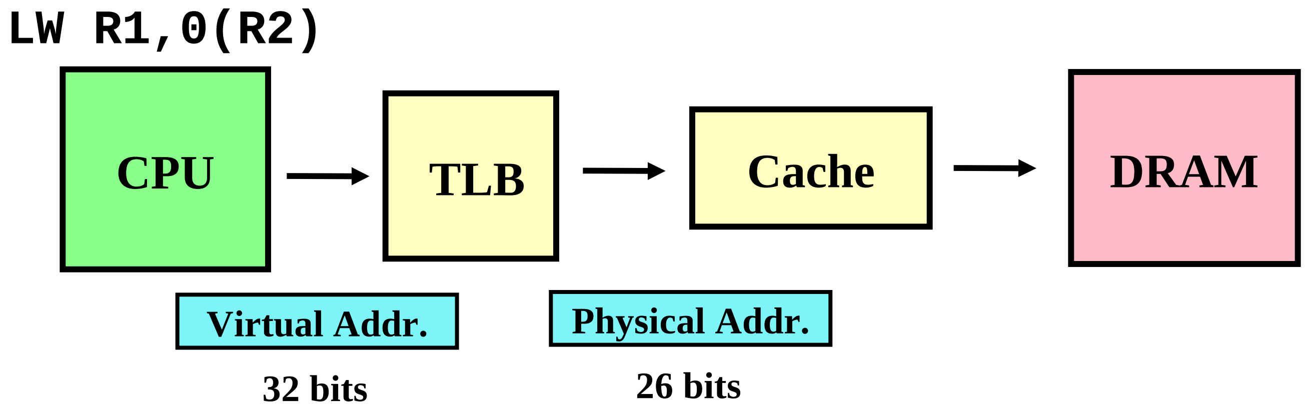
Reducing TLB misses

- Same type of optimizations as for cache
  - Associativity (many TLBs are fully associative)
  - Capacity – TLBs tend to be 32-256 entries
- Adjust page size
  - Small pages
    - Reduces internal fragmentation
    - Speeds page movement to/from disk
  - Large pages
    - Can cover more physical memory with same number of TLB entries
  - Solution: typically have a variable page size
    - Select by OS, 4KB-16MB (superpages)

Virtual Memory + Caching

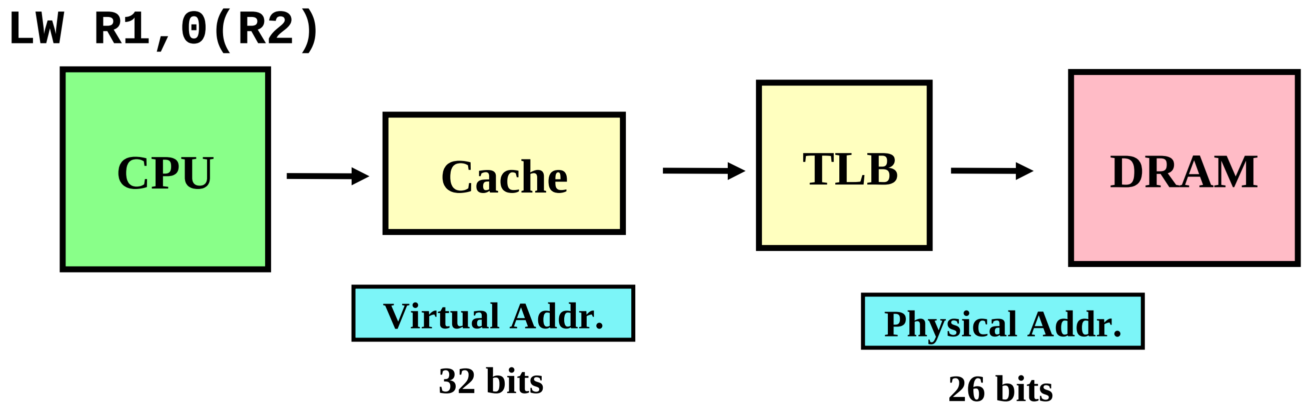
- Conflicting demands:
  - Convenience of flexible memory management (translation)
  - Performance of memory hierarchy (caching)
- Requires cooperation of O/S
  - Data in cache implies that data is in main memory
- Combine VM and Caching

Physically Addressed Cache



- Translate first from VA $\Rightarrow$ PA
- Access cache with PA

Virtually Addressed Cache

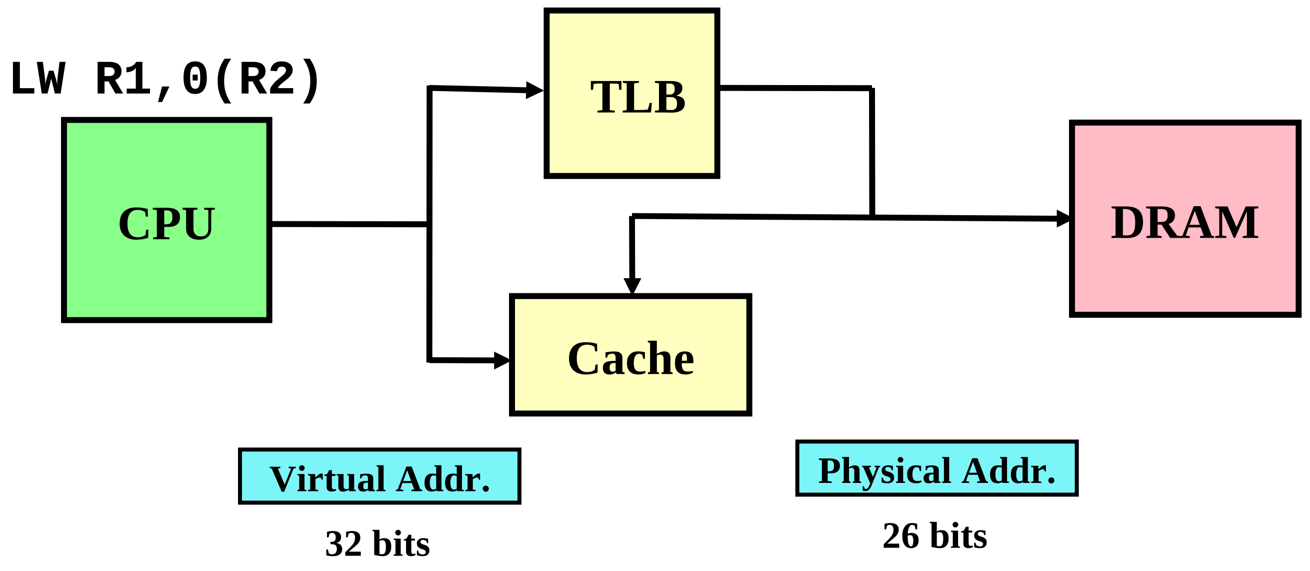


- Access cache first
- Only translate if going to main memory

Aliasing

- Can occur when switching among multiple address spaces
- Synonym aliasing
  - Different VAs point to the same PA
  - Occurs when data shared among multiple address spaces
  - One solution: always translate before going to the cache
- Homonym aliasing
  - Same VA point to different PAs
  - Occurs on context switching
  - Two solutions:
    - Flush TLB on process switch/system call
    - TLB includes process ID

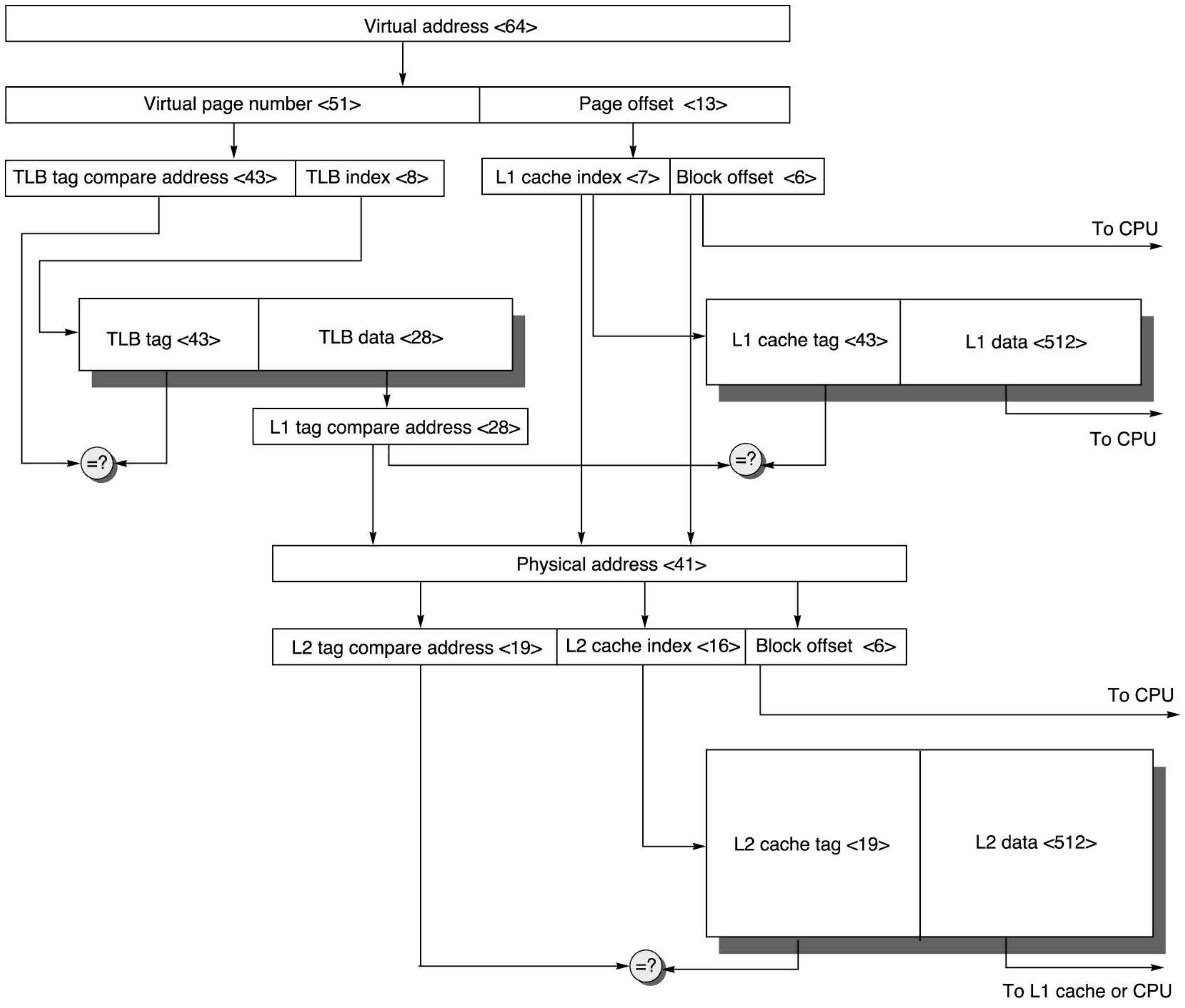
Best of Both Worlds:



Virtually addressed, Physically Tagged

- **Parallel Access**
- **Eliminate synonym problem**

Virtual Index, Physical Tag


Virtual address <64>
Virtual page number <51>
Page offset <13>
TLB tag compare address <43>
TLB index <8>
L1 cache index <7>
Block offset <6>
To CPU
TLB tag <43>
TLB data <28>
L1 cache tag <43>
L1 data <512>
L1 tag compare address <28>
To CPU
=?
=?
Physical address <41>
L2 tag compare address <19>
L2 cache index <16>
Block offset <6>
To CPU
L2 cache tag <19>
L2 data <512>
=?
To L1 cache or CPU