Memory

Operating System Lab-in-a-Box

Previous

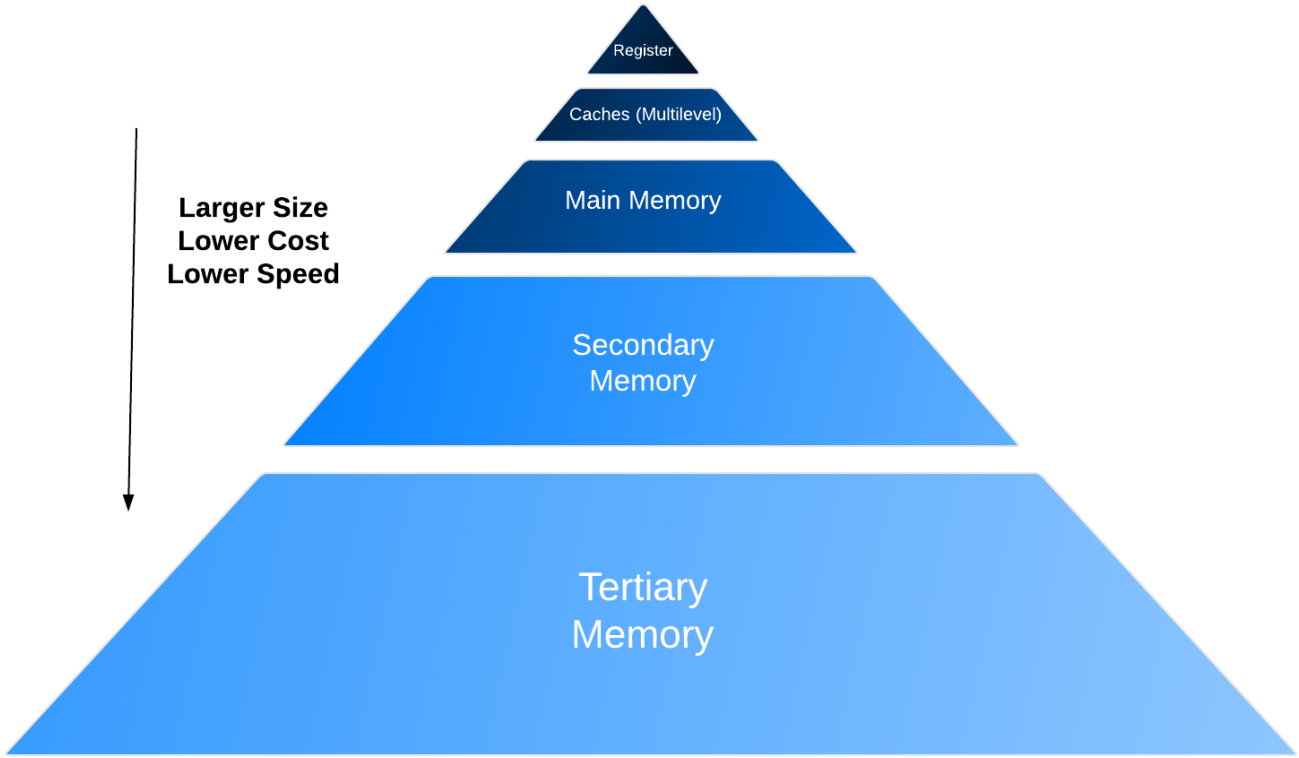
- Concurrency
- Lab

Current Presentation Content

- Memory Hierarchy
- Cache,
  - Entries, metrics, blocks,
  - Direct mapped cache, N-way set associative Cache, Fully associative Cache
  - Block replacement policy
- Memory map
- Memory management
- Address binding and dynamic binding
- Relocation register, Memory Partitioning
- Buddy memory and non-contiguous memory allocation
- Paging, Translation Look aside Buffer (TLB), Multilevel Page Table
- Segmentation with Paging

Memory Hierarchy

- Register: usually one CPU cycle to access
- Cache: CPU cache, TLB, Page cache
  - Static RAM
- Main Memory: the “memory” memory
  - Dynamic RAM
  - Volatile
- Secondary Memory: Hard disk
- Tertiary Memory: Tape libraries



- Temporal locality
- Spatial locality

- Memory Hierarchy – to exploit the memory locality

arm

CPU Cache

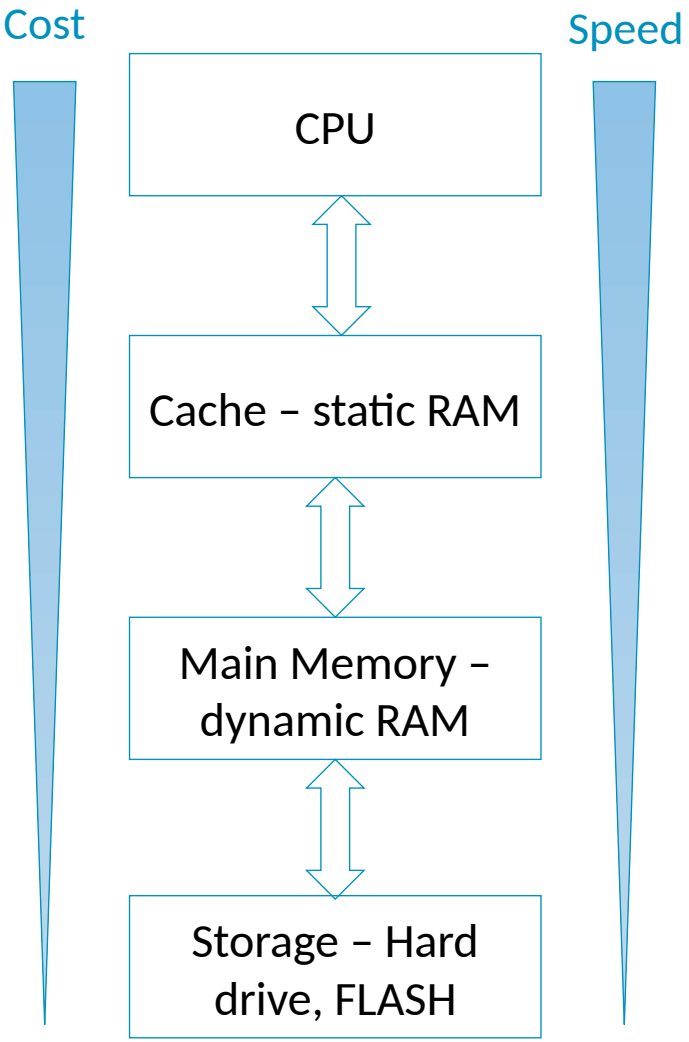
CPU Cache - Why do we use cache

- 90/10 Rule
  - A typical processor spends 90% of its execution time in 10% of the total program code.
  - This means we can store the critical 10% of the code in a fast- accessible memory whereas leave the rest 90% on a storage disk

- Locality of reference (or Principle of locality):
  - In many cases, a process accesses the same value, or related storage locations much more frequently than others
    - Temporal Locality - a process tends to reference in the near future those locations that have been referenced shortly before
      - For example, a for (;;) loop
    - Spatial Locality - a process tends to reference a portion of the address space in the neighbourhood of its last reference
      - For example, a data array

CPU Cache Hierarchy

- The CPU makes use of small, fast, and very expensive registers.
- Main memory is cheaper and greater capacity but takes much longer to access.
- The CPU cache is used to exploit locality – keeping copies of data likely to be used again in faster access memory to reduce the average time taken to access the data.
- The cache sits in-between the fast static CPU registers, and the slow memory both in terms of cost and speed.



Cache Entries

- The cache operates as follows:
  - Whenever the CPU needs to read from a location residing in main memory, it first checks the cache for any matching entries.
  - If the location exists in the cache it is simply returned directly to the CPU – this is known as a cache hit.
  - If the location doesn't exist in the cache, also known as a cache miss, the cache allocates a new entry for the location, copies the contents from main memory, and then fulfils the request from the contents in the cache.
- Writing works in a similar fashion – the contents are written to the cache and then periodically propagated through to main memory.
- The proportion of accesses that result in hits, as opposed to misses, is known as the hit rate and is a useful measure of the effectiveness of the cache.

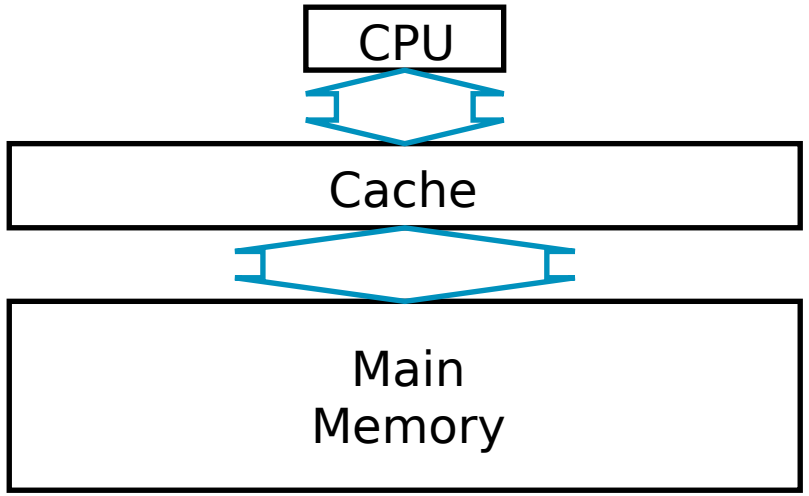
Some Cache Metrics

- Memory cycle time
  - The minimum delay between two memory operations
- Memory access time
  - The elapsed time between start and finish of memory access
- Bandwidth – data throughput
  - Bandwidth = bit rate * number bits
- Cache Hit:
  - Cache Hit Time: elapsed time in finding and retrieving data from the cache
  - Cache Hit Rate: proportion of cache hits to cache misses
- Cache Miss:
  - Cache Miss Penalty: time required to retrieve data from main memory (as opposed to cache)
  - Cache Miss Rate = 1 – Cache Hit Rate
- Average memory access time, or AMAT, can be calculated as follows:

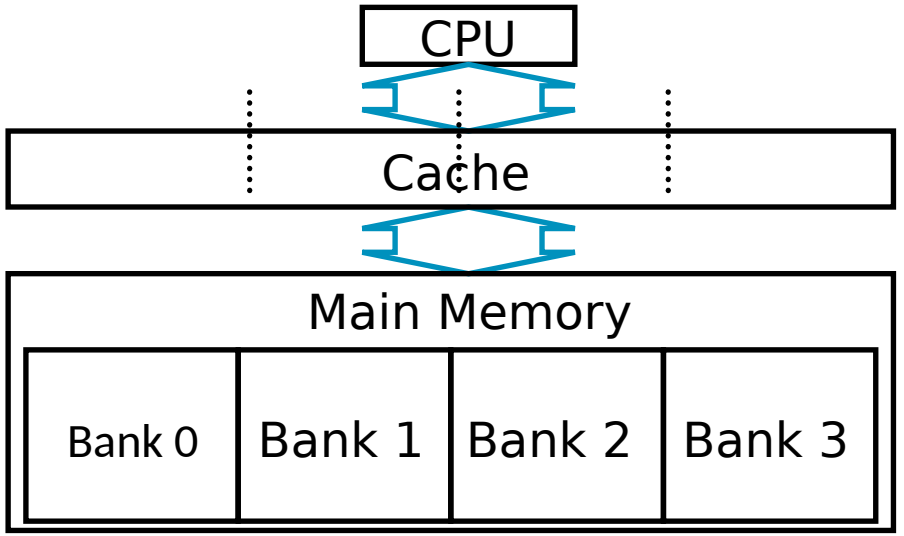
AMAT = Cache Hit time + (Miss rate x Miss penalty)

Decreasing the Miss Penalty

- Operate the main memory in ‘Burst Mode’ – once an address has been accessed, subsequent addresses can be accessed with a much lower overhead
- Widen the memory and its cache interface so multiple words can be read in one shot
- Organise the memory into banks that allow multiple reads and writes in parallel using interleaving – cheaper than implementing a wider bus for much the same speed



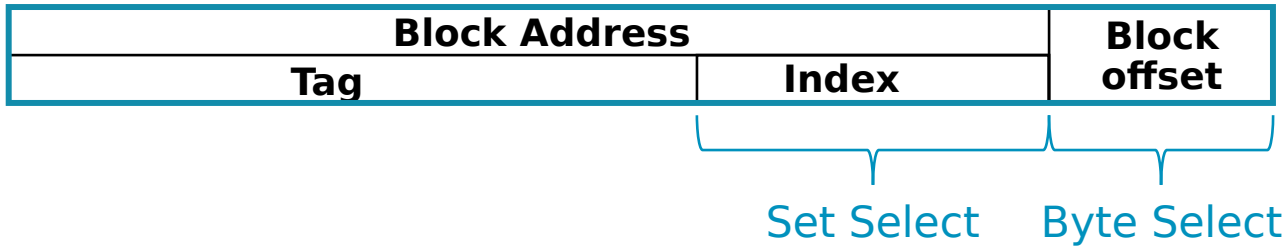
Wide Bus Memory



Interleaved Memory

Cache Blocks

- Often data is moved from main memory to the cache in blocks as **Blocks** rather than single bytes to take advantage of spatial locality – a prediction that access to one address will be followed by access to a nearby address.
- The cache blocks contain only a subset of the main memory, therefore multiple memory addresses map to the same cache location. How do we tell which one is in there?
  - Answer: divide memory address into three fields:



  - The block **index** determines which set to look in when searching for a location.
  - The block **tag** distinguishes between all the main memory locations that map to the same cache index.
  - The block **offset** identifies the byte offset within the block.

  - A **set** is a grouping of slots in which a block can reside. For example if sets contain two blocks the cache can hold two memory blocks that map to the same cache location.

Placement

- The cache is limited in size, therefore it must be determined where newly fetched blocks should be placed in the cache, here we will look at three common placement policies:
  - Direct Mapped
  - Set Associative
  - Fully Associative

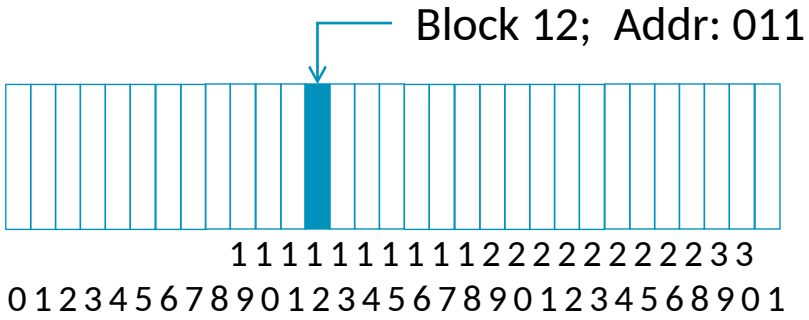
Direct Mapped Cache

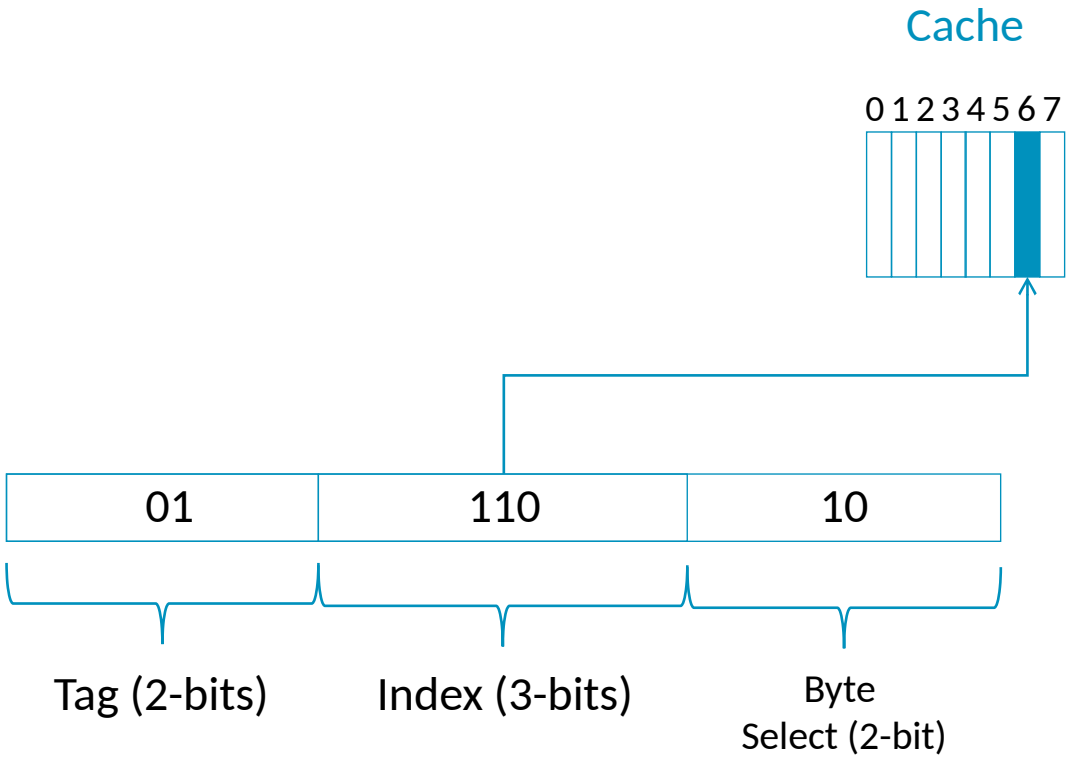
- In a Direct Mapped Cache each block has only one slot (set size of 1) in which it can reside in the cache.
- This results in a simple design, but can be problematic when memory locations repeatedly map to the same cache slot and knock each other out.

- Pros:
  - Simple design - inexpensive
  - Low number of indexes – quicker to search – therefore fast
- Cons:
  - Low flexibility can lead to low hit-rate.

Direct Mapped Cache

- Example: 32-block (5-bit) address space; 4-byte words; 8-block cache
- Access byte 2 of block 12 (01110).





- 3-bit index maps each block to a single cache slot.
- 4 memory blocks map to each cache slot, differentiated by 2-bit tag.

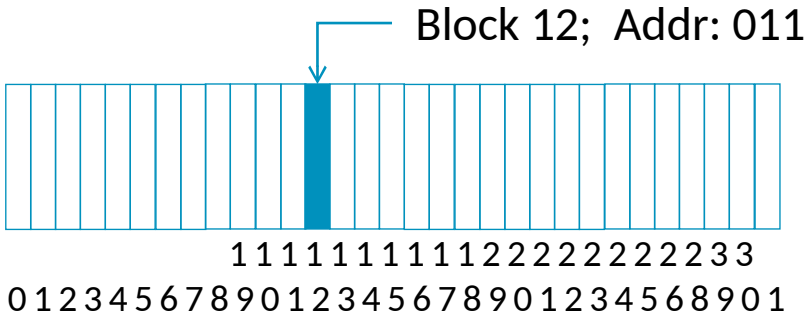
N-Way Set Associative Cache

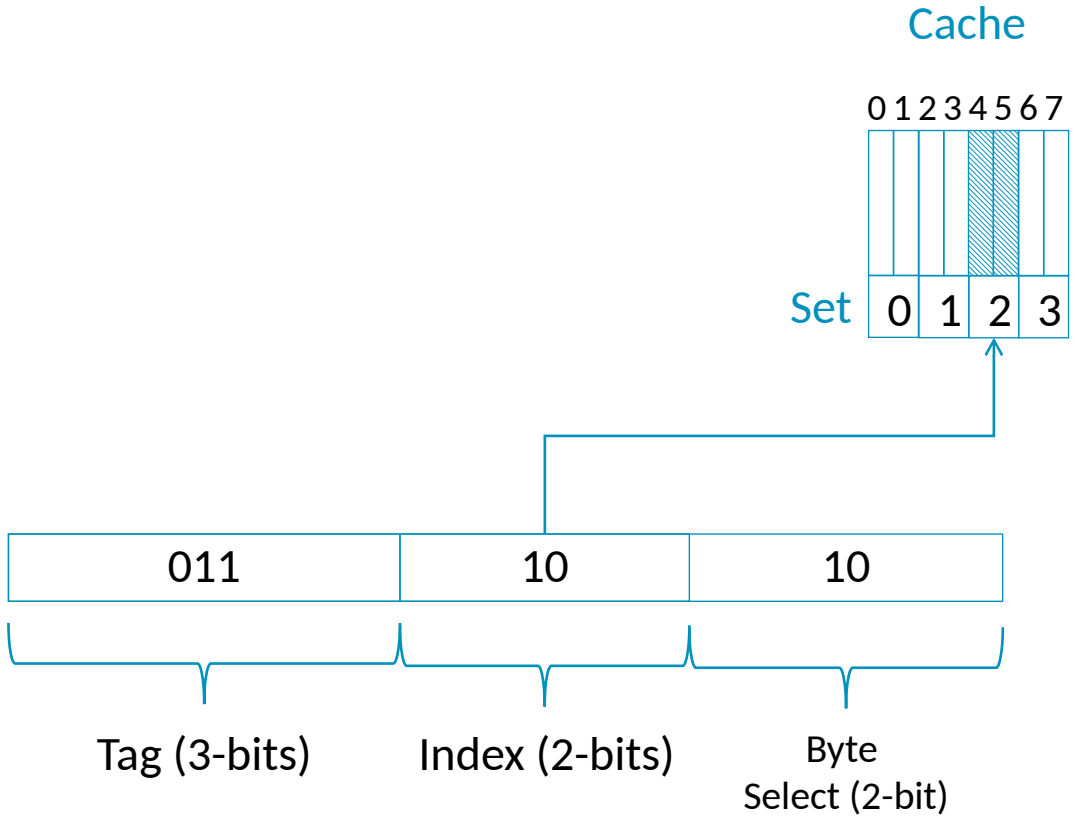
- A block can be placed in N slots in an N-Way Set Associative Cache.
- A block is first mapped to a set, and then can be put in any available slot within the set.
- This goes towards solving the problem with memory locations repeatedly mapping to the same slot.
- How it is placed within the set depends on the Replacement Policy (covered later).

- Pros:
  - The main advantage of the Set Associative cache is that is combines the speed of direct mapping with the flexibility of fully associative caches.
- Cons:
  - Finding blocks within a set requires more complex hardware – more expensive in terms of cost and time

N-Way Set Associative Cache

- Example: 2-Way Set; 32-block (5-bit) address space; 4-byte words; 8-block cache
- Access byte 2 of block 12 (01110).





- 2-bit index maps each block to a set that contains 2 slots.
- 8 memory blocks map to each cache slot, differentiated by 3-bit tag.

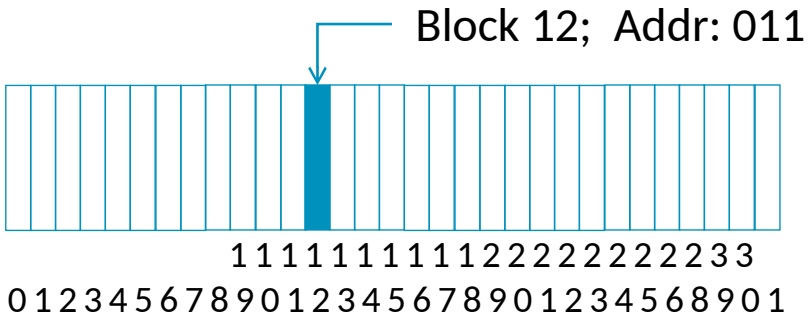
Fully Associative Cache

- In a Fully Associative Cache a block can be placed in any available slot.
- Makes the cache more flexible, increasing the hit rate
- More complex, as searching for a block involves comparing all slots.

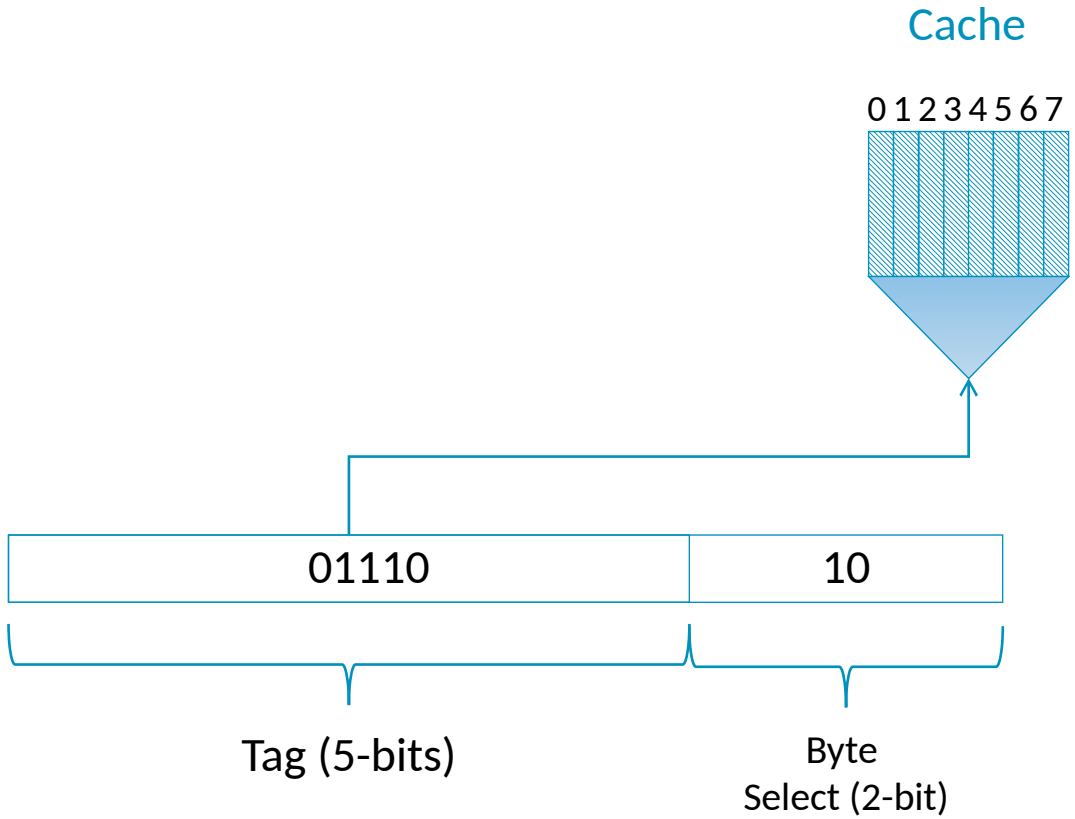
- Pros:
  - Good flexibility - greater hit rate
- Cons:
  - Requires large comparator to search for blocks within sets, which is expensive and slow

Fully Associative Cache

- Example: 32-block (5-bit) address space; 4-byte words; 8-block cache
- Access byte 2 of block 12 (01110).



- No index, block can be placed anywhere in the cache.
- Each block is identified in the cache by a 5-bit tag.



Associativity Trade-off

- The level of associativity is a trade-off when considering N-Way caches
- The more slots available (bigger the set) for a block requires more searching, which costs time and power – however more slots increases the flexibility and therefore the hit rate.

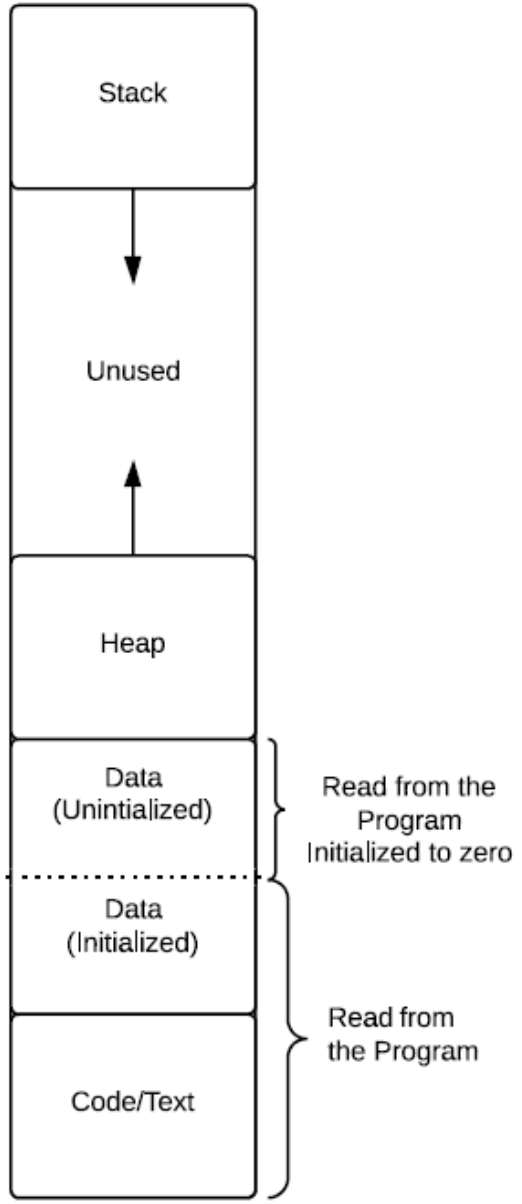
Block Replacement Policy

- In a Direct Mapped cache there is no choice as to where to put a block. However with associative caches there is more than one slot to choose from when a block is retrieved.
- Some common replacement policies:
  - Random – new block replaces randomly selected block, easiest to implement
  - Round Robin – first-in, first-out (FIFO); new block replaces oldest block – easy to implement
  - Least Recently Used – new block replaces least recently used block, requires extra complexity to track block usage

Memory Management

Processes Need Memory

- Code or Text
  - Binary instructions to be executed
  - A clone of the program
  - Usually read-only
  - Program counter (PC), points to the next instruction
- Static Data
  - Global/Constant/Static variables - shared between threads
  - If not initialized by program, will be zero or null pointer
- Heap
  - malloc/free
- Stack
  - Used for procedure calls and return
  - Stack Pointer(SP)
    - First In Last Out (FILO)



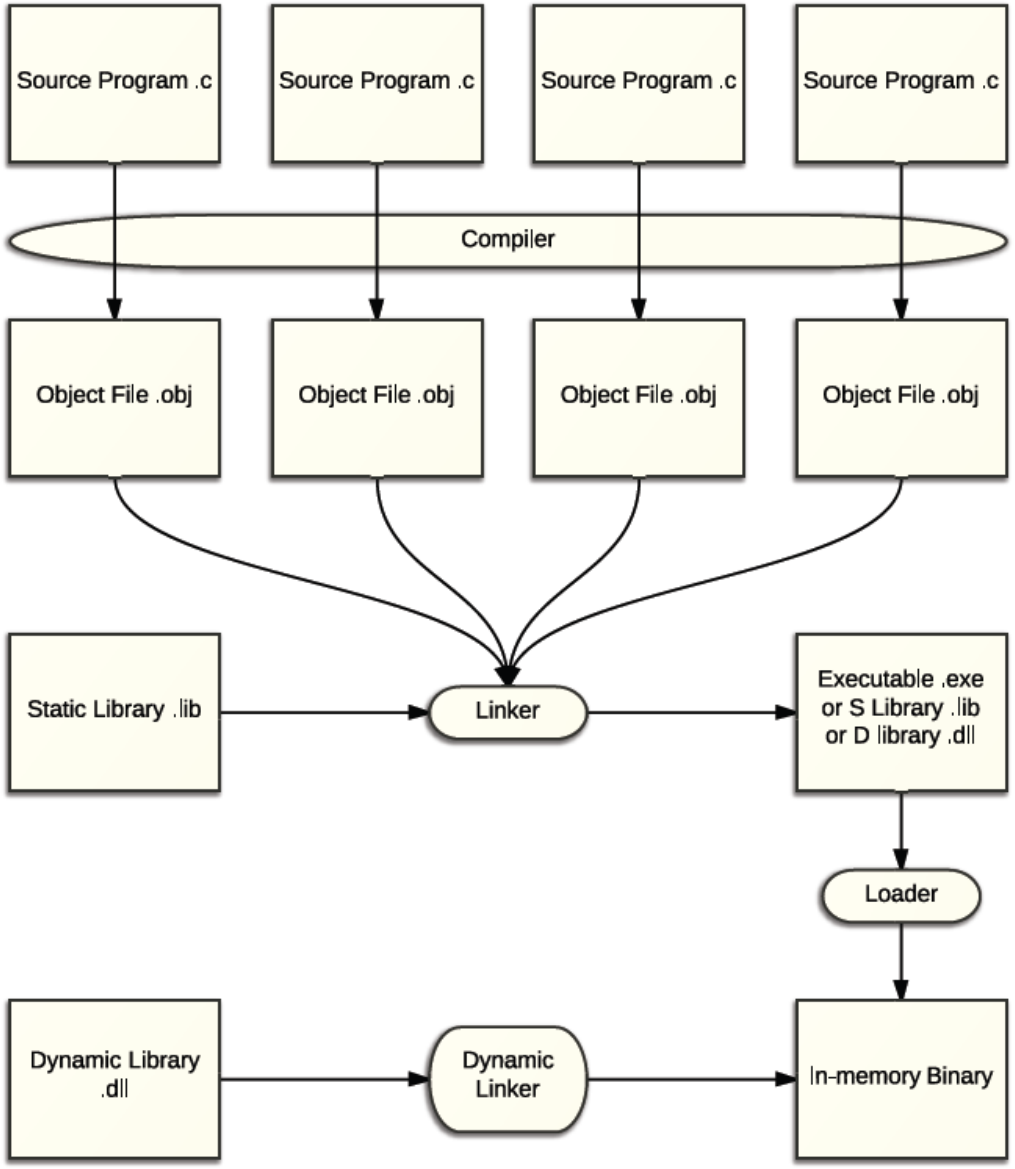
What Does Memory Management Do?

- Both the OS and user processes need memory

- Allocation/Partition – compact way of allocating
- Relocation – changing the memory space dynamically, ideally translation done by hardware
- Protection – illegal reference to other processes' memory should be detected and stopped at run time
- Sharing – several processes access common part of the memory

Address Binding

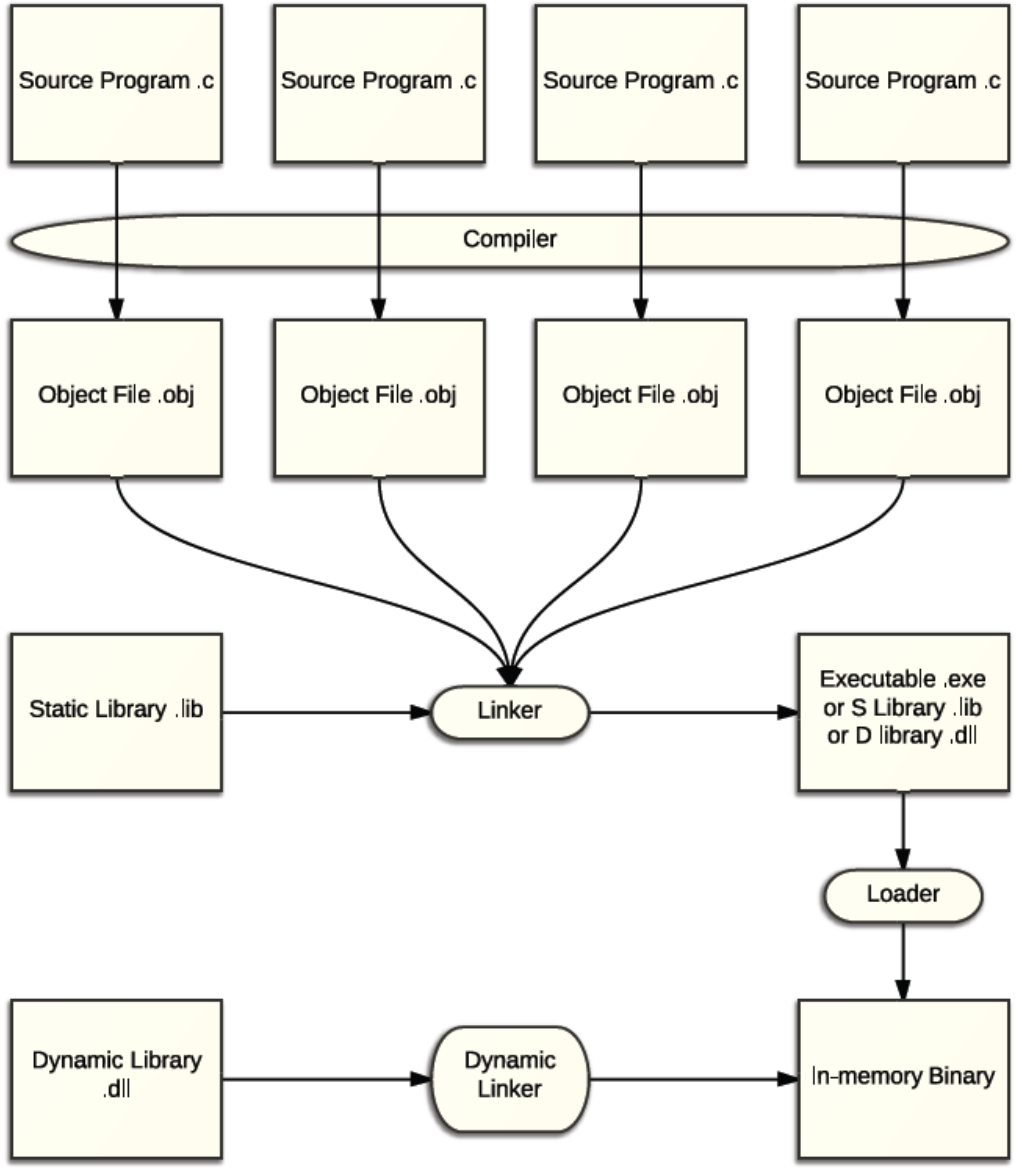
- Where is the variable in memory and where is the location that the program is branching to?
- Needs to map the program to main memory
  - Symbolic addresses in the source program
  - Instructions in the source program
- Steps to run your program:
  - Compilation – convert symbolic references into numerical values and turn everything into machine code. Also contains placeholders and offsets as the linker will solve them.
  - Linking – either static or dynamic. It is possible to load and link libraries at run-time.
  - Loading – load the program into the main memory and all other necessary preparation



arm

Address Binding

- Static Address Binding at
  - Programming
    - Programmers know the exact address they want
  - Compilation
    - Memory location known before execution
    - Absolute code and less flexible
  - Loading
    - Simple - relative to the beginning of the executable (offset)
    - Program starts at location L, then L is added to all memory addresses in the program
    - Not flexible enough – what if swapped out?
- Dynamic Address Binding at runtime
  - Flexible – can relocate processes
  - At the cost of complicated address translation



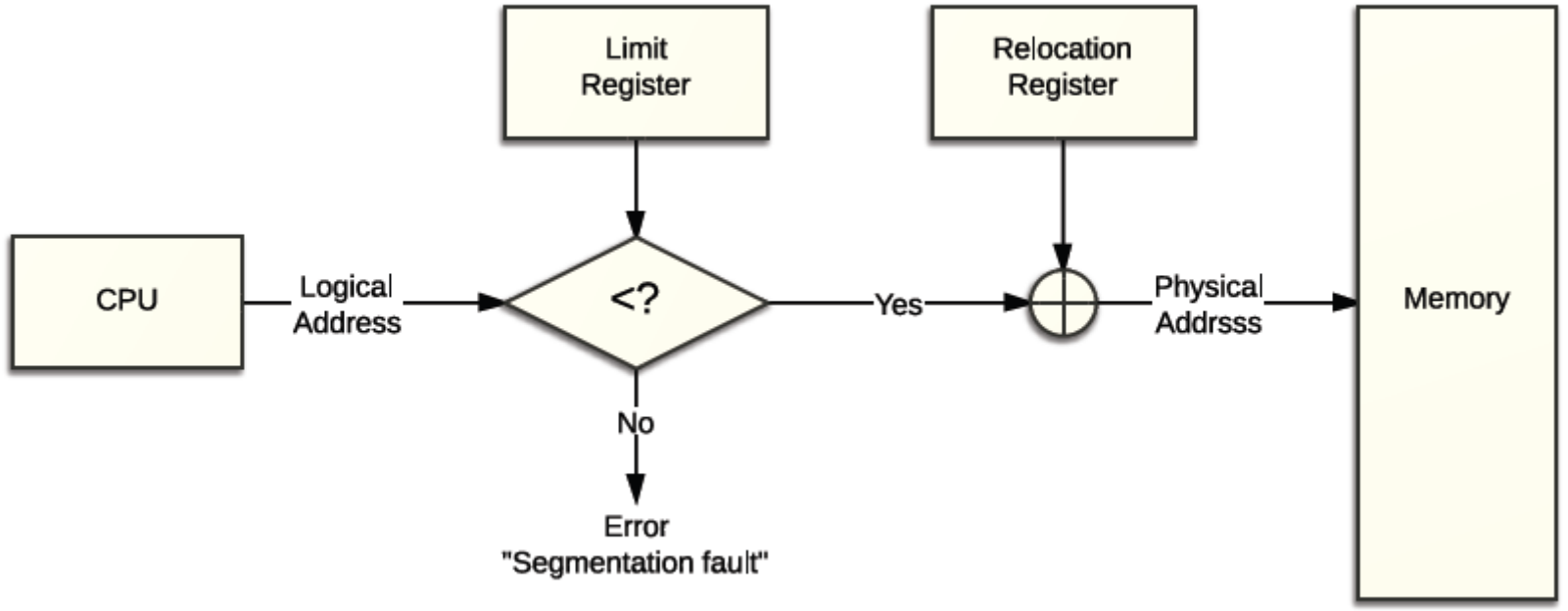
arm

Dynamic Address Binding

- Relocation is very frequent due to compaction and swapping
- If static binding, needs explicit programming for the relocation
- Alternatively, make OS handle it at runtime
- Bind a logical/virtual address to a physical address
  - Physical address – absolute location in the main memory
    - Logical address – a reference to the physical address, should be independent of the organization and structure of the physical memory
    - Conceptually, PhysicalAddress = address_translation (LogicalAddress)
    - All translation is done by the component called the memory management unit (MMU)
- How to implement the address_translation?
  - Relocation Register – relative address and continuous memory
  - Page/Multi-Segment – non–continuous memory

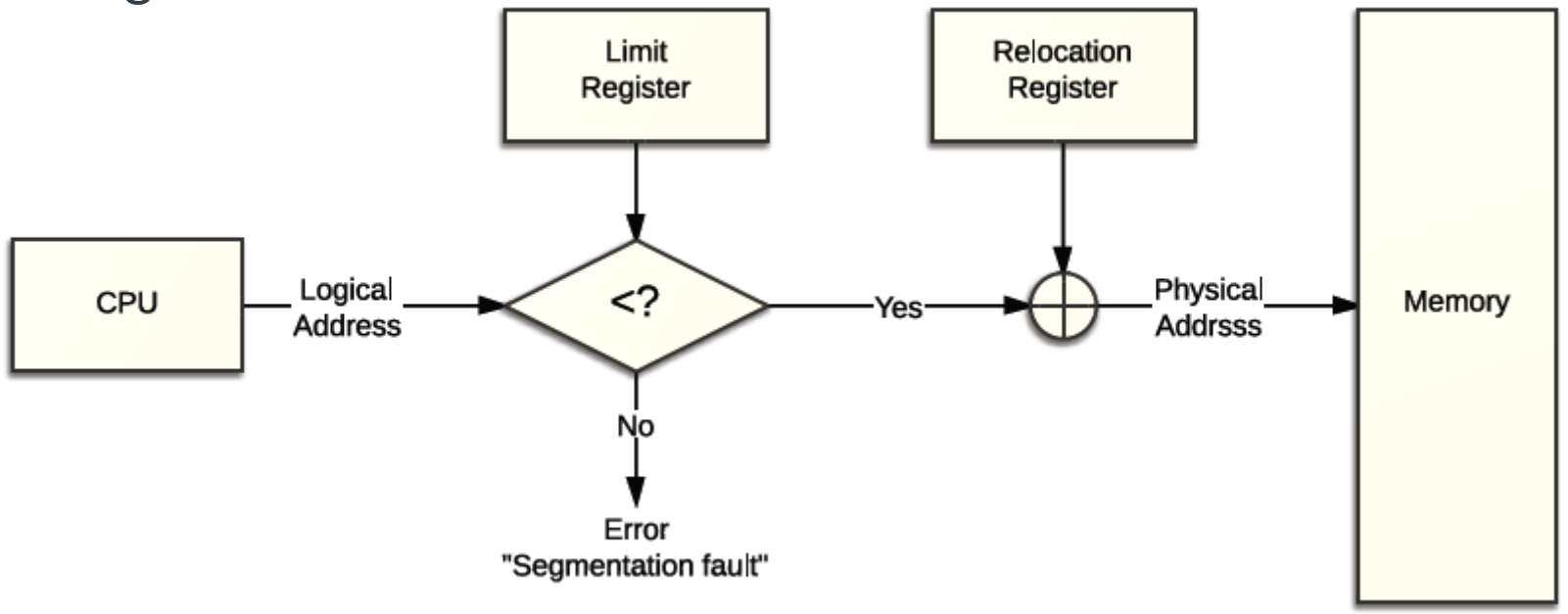
Relocation Register

- OS writes the base address *b* into the relocation register and a hardware adder will automatically translate the program referred address (relative address) *a* to *a+b*.
- For protection purpose, check if the relative address is out of range, using a comparator and the limit register (*a<l?*). Address fault if not and OS has to handle the error.
- So, the OS only needs to modify the relocation register for swaps/compactions
- Memory space from *b* to *b+l* can also be called a segment.



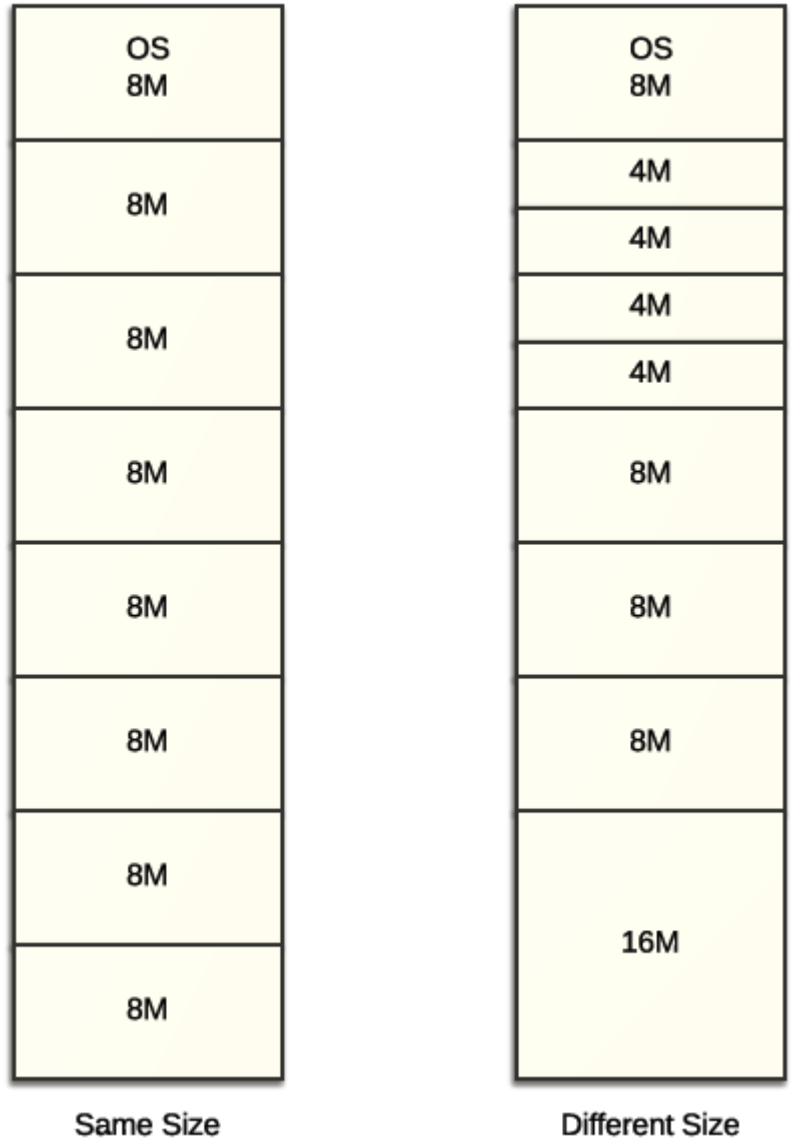
Relocation Register

- This can be extended to a multi-segment scheme
  - Different segments for different types of data e.g., text/stack/data
  - Logical address: (segment-id, offset), for different segments, we have different *b* and *l*
  - Each process possesses a unique segment table stores related translation information
- So the question is how can you determine
  - *b*
  - *l*
- Memory Partitioning
  - Two Schemes:
  - Fixed
  - Dynamic



Memory Partitioning

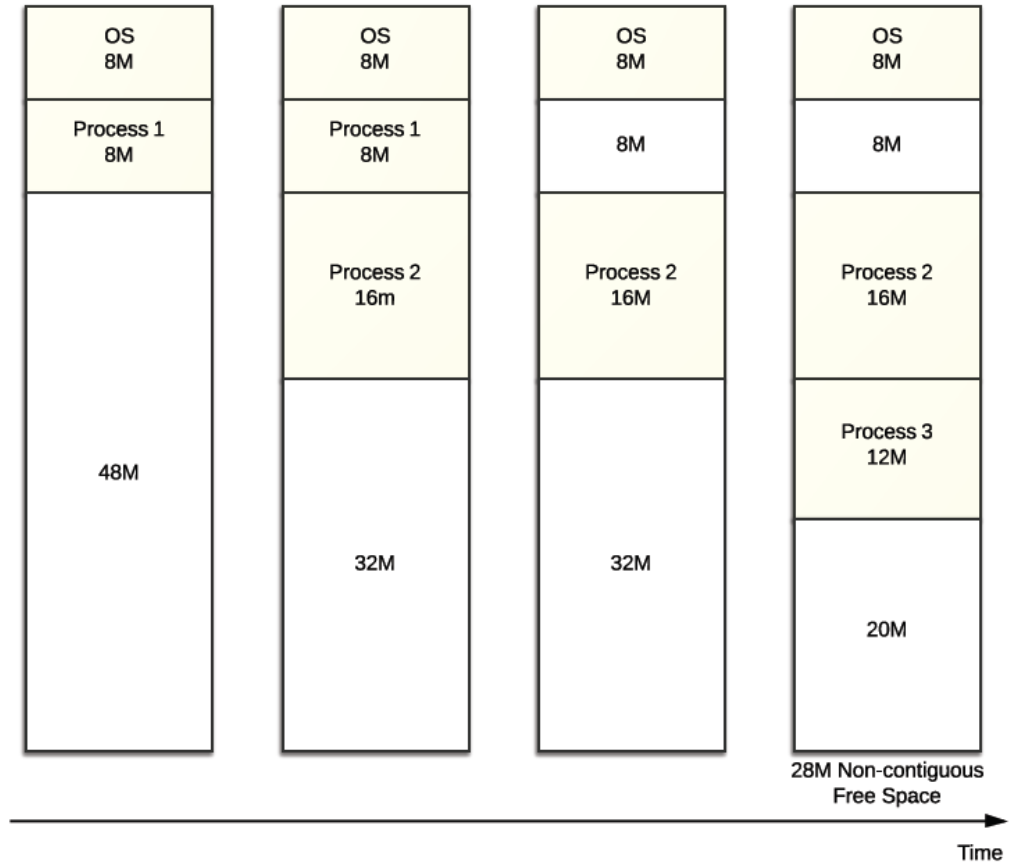
- Fixed *b*
  - *l* can either be fixed or not.
  - No overlapping and the boundary is fixed for all chunks.
- Does the memory space for a process grow/shrink?
- What if a process requires more than 8M?
  - Program should be explicit enough to handle this
  - Or a larger *l*?
- What if a process only needs tiny space?
  - Regardless of the size, a whole chunk is allocated
  - Internal fragmentation - waste of space
  - Different size chunks may help but not so thoroughly



arm

Memory Partitioning

- Variable *I*
  - Choose a suitable size, slightly larger than what is required
  - External fragmentation
- Variable *b*
  - Many options, which to go? To reduce fragmentation
- Very dynamic scheme and OS has to keep track of all related information
  - Responsive – find a hole quickly enough
  - Coalescing – merge adjacent free holes



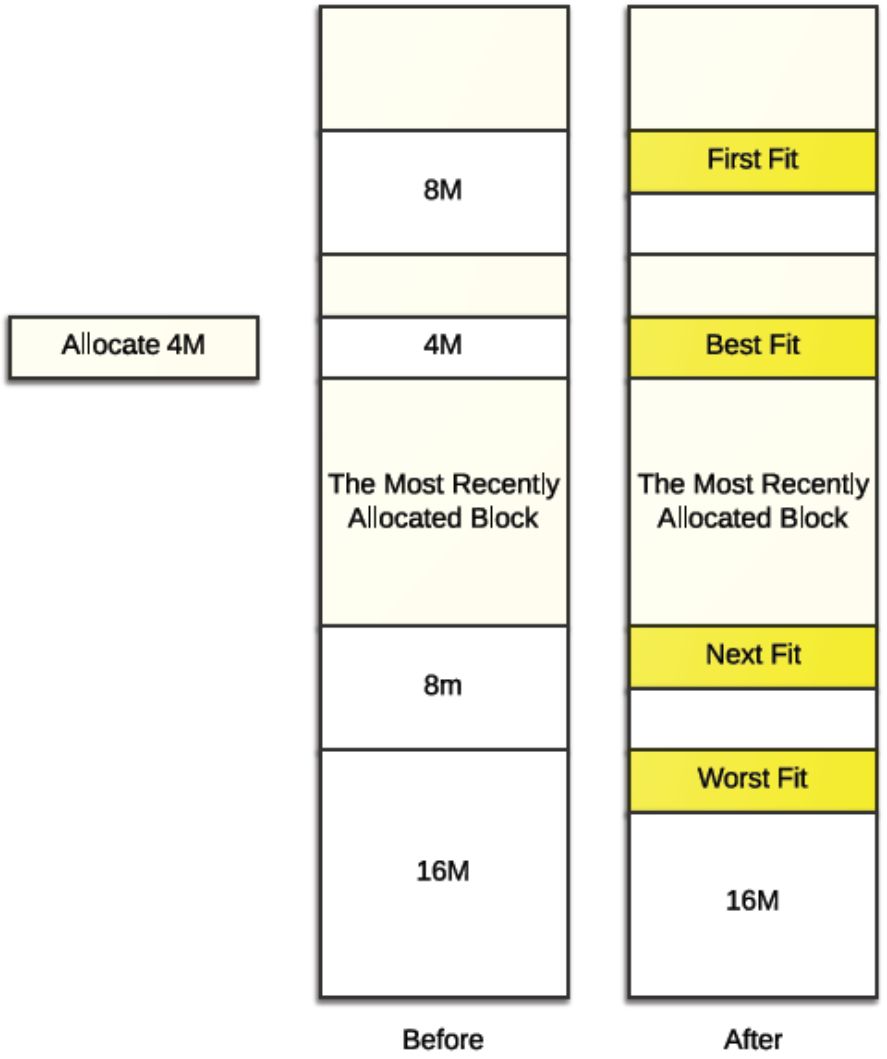
arm

Memory Partitioning

- Variable *b* – Some algorithms
  - First fit – first large enough hole
    - Next fit – first large enough hole from where last search ended
    - Best fit – smallest large enough hole
    - Worst fit – largest large enough hole

- First fit is generally the fastest and produces less fragmentation
- Next fit better than best fit
- Best fit better than worst fit

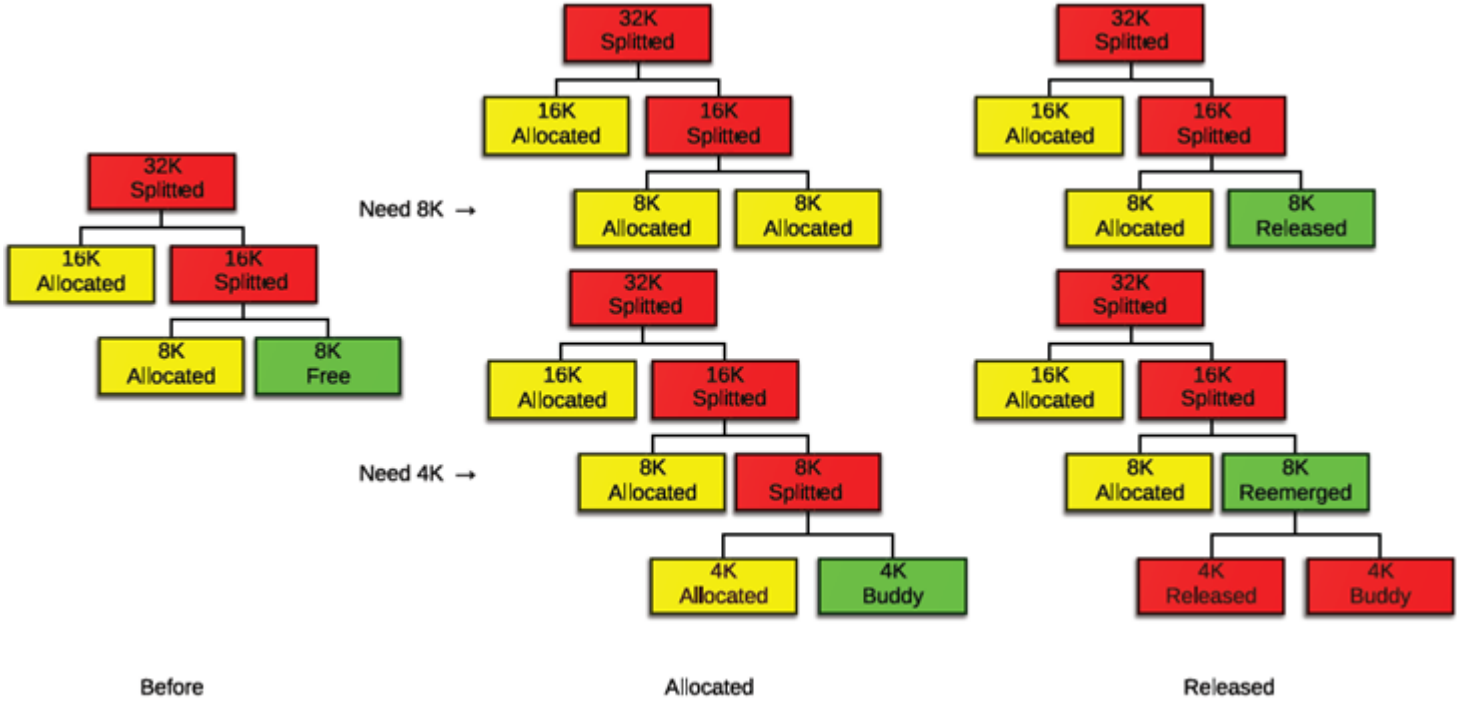
- Need compaction in the end



arm

Buddy Memory Allocation

- Less (external) fragmentation
- Binary buddies, the most common variant
  - Keeps a tree like structure with depth order *n*
  - Block size 2^n (or proportional to 2^n) –one block can be split into two blocks (two buddies) that are one order lower
    - On arrival of request of space *r*, $2^{i-1} < r \leq 2^i$, find a free block from all current blocks with the order *i*. If not found, find a free block from all blocks with the order *i+1* and split that block into two to make a free block and repeat this recursively until successful allocation
    - On freeing a block, coalesce the other buddy if it is free to make a block that are one order higher



*Knowlton, Kenneth C. "A fast storage allocator." *Communications of the ACM* 8.10 (1965): 623-624.

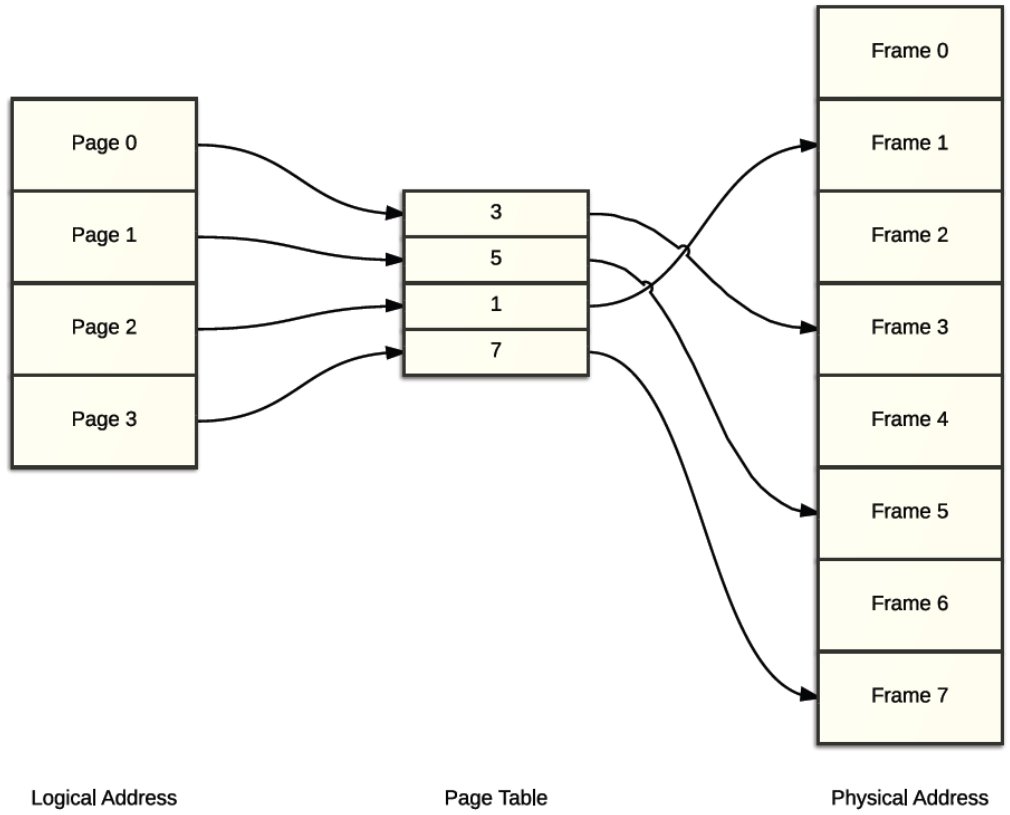
Non-Contiguous Memory Allocation

- Internal/External Fragmentation is common and another way to get rid of them is via non-contiguous memory allocation
  - PhysicalAddress = address_translation (LogicalAddress)
- Although the logical address can be contiguous, the physical address could actually be separated into several chunks in different physical memory location
- Although it may not match the users' (programmers') view of memory:
  - It can further reduce fragmentation
  - Protecting only part of the memory of the processes is possible
  - Sharing chunks of data will be easier
  - Better support for swapping and compaction
  - Virtual memory

Paging

- Logical address – fixed size page
- Mapped to physical address – same size frame
- The translation is done through the page table

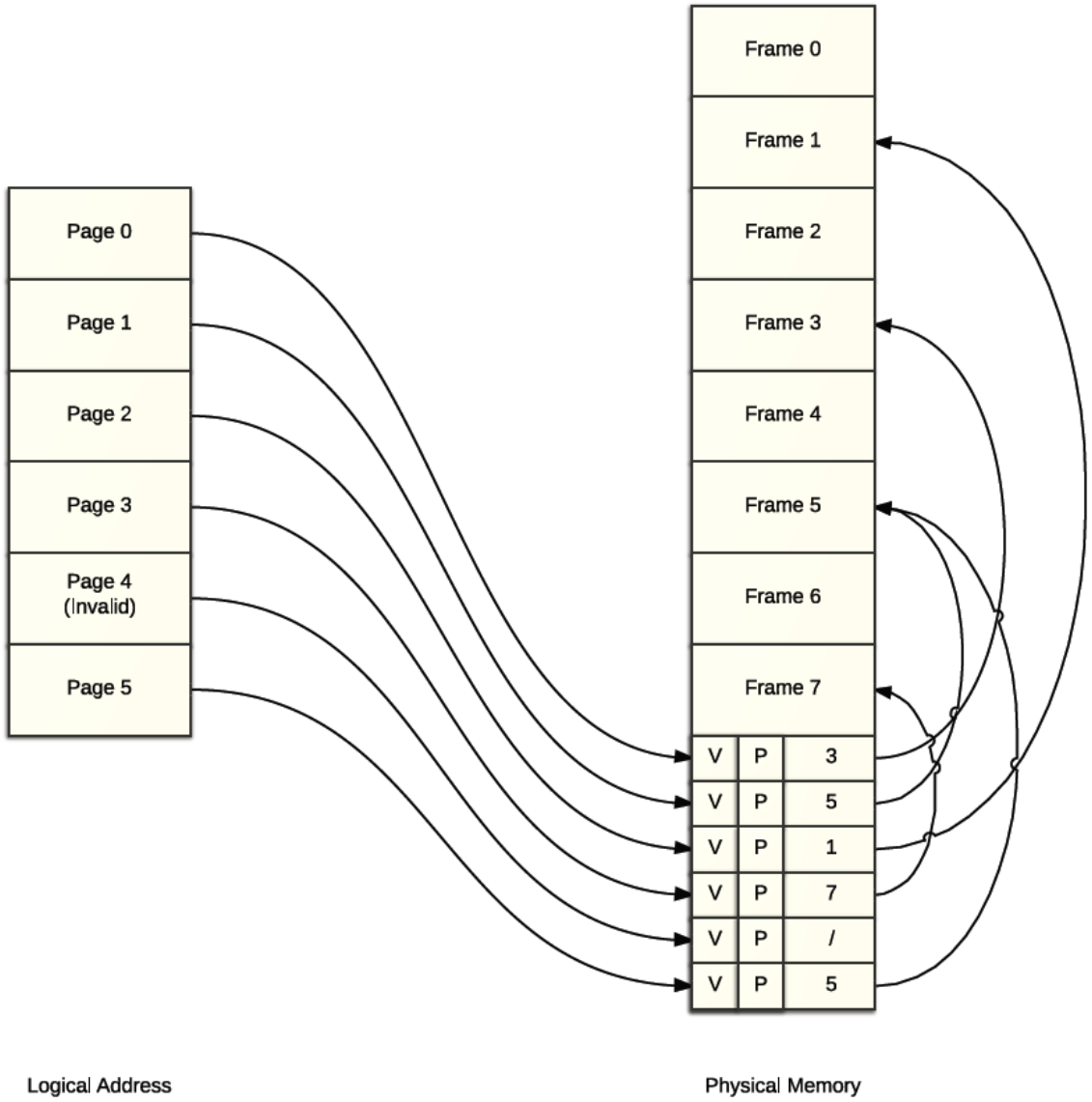
- PhysicalAddress = PageSizepage_table(LogicalAddress/PageSize) +LogicalAddress%PageSize
- Page size, usually power of 2, 4KB for example
- Assume byte addressing, 32 bit address: PhysicalAddress[31:12] = page_table(LogicalAddress[31:12]); PhysicalAddress[11:0] = LogicalAddress[11:0]



arm

Paging

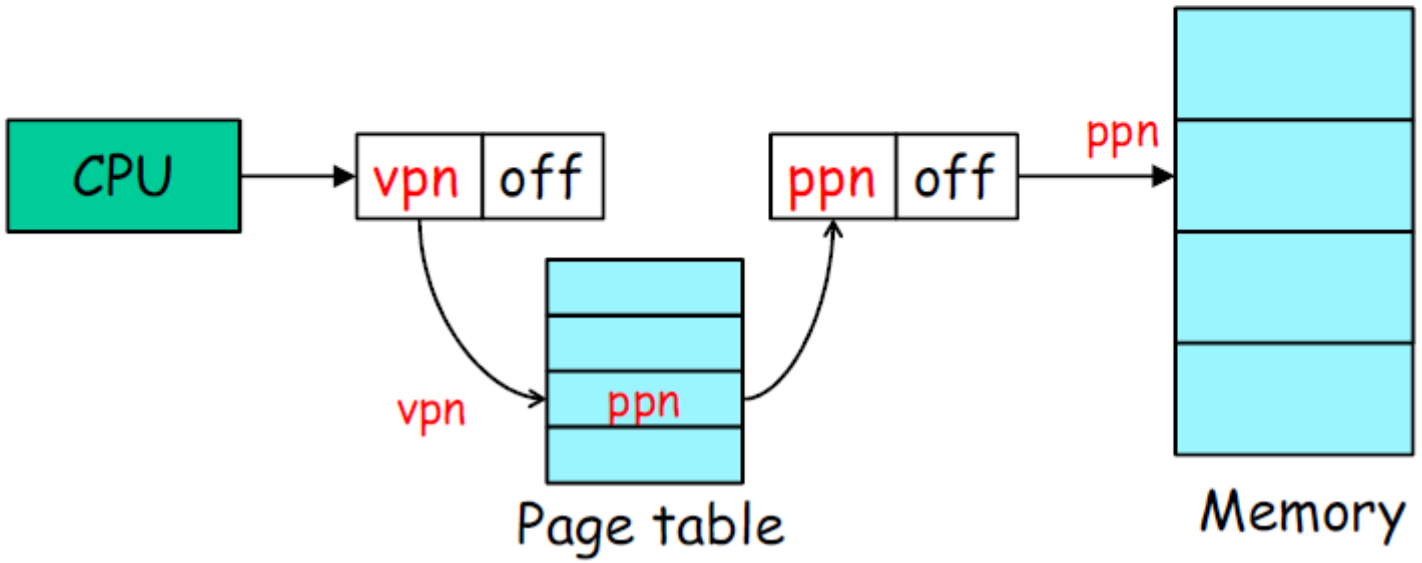
- PT entries can also include
  - Valid bit
  - Protection/Mode bits
  - Other control bits
- PT makes sharing a specific chunk of memory simple (Page 1 and Page 5)
- PT has to be fast. Where does it reside ?
  - Register? – fast but limited space
  - More commonly, kept in the main memory
  - Two access to memory for every data/instruction access



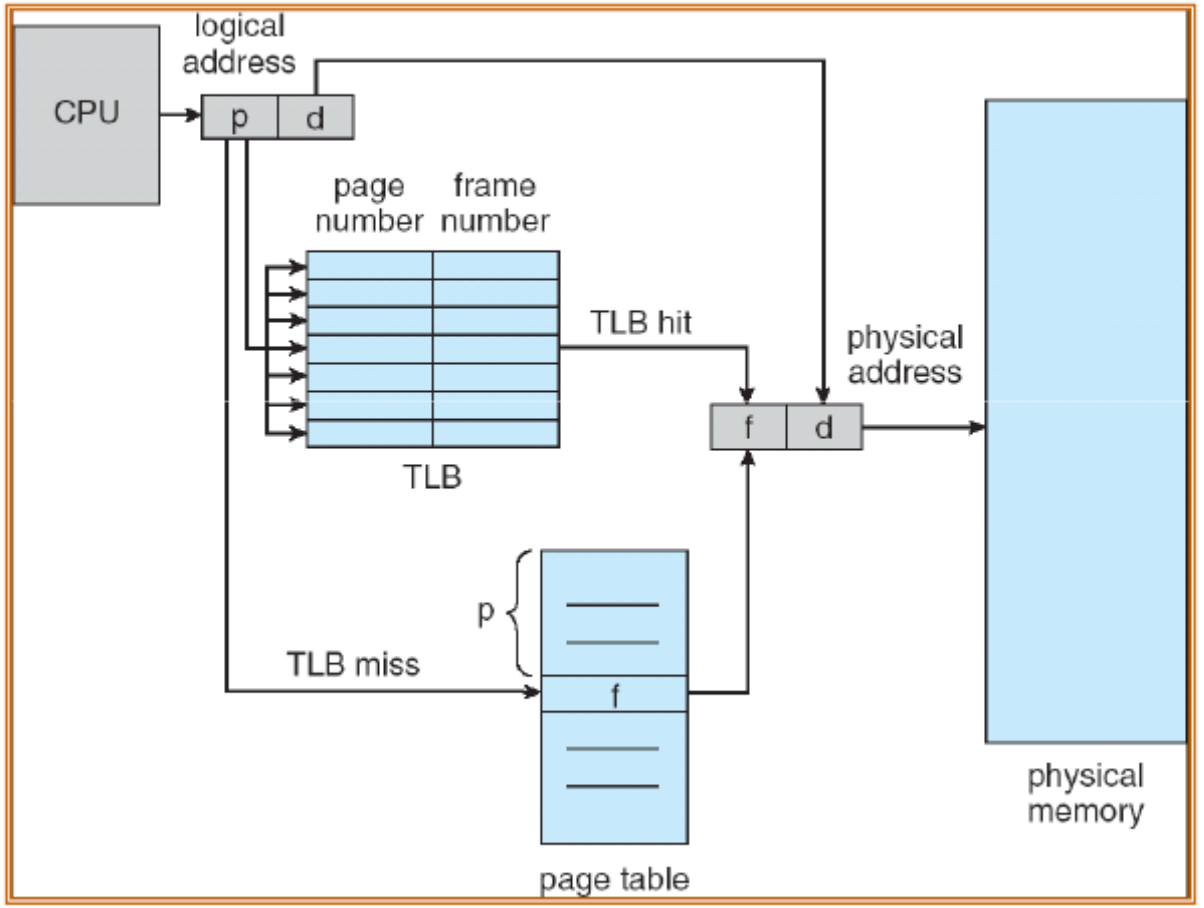
Page translation

- Address bits = page number + page offset
- Translate virtual page number (vpn) to physical page number (ppn) using page table

  pa = page_table[va/pg_sz] + va%pg_sz



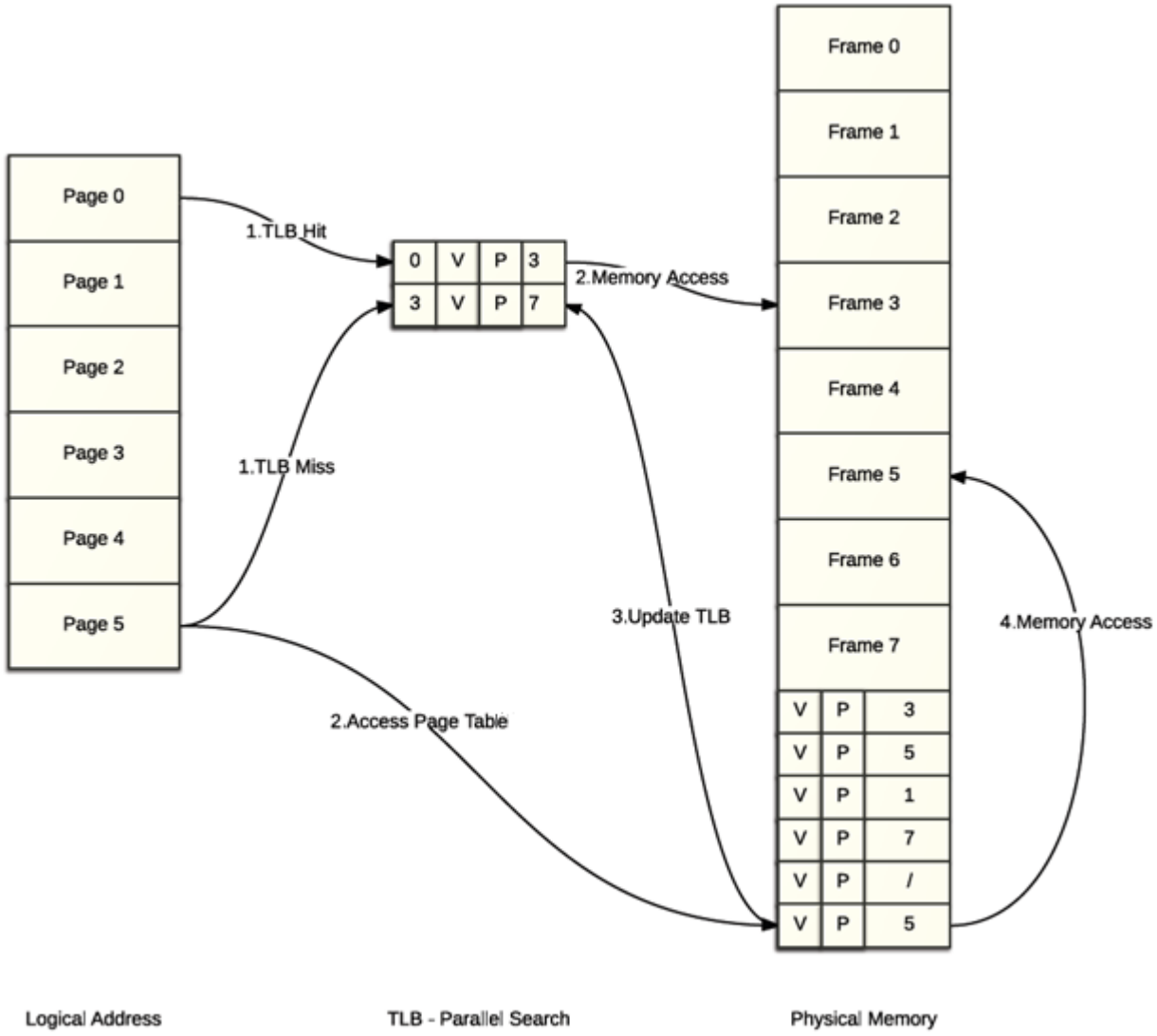
Paging hardware with TLB



Translation Lookaside Buffer

- Cache the PT in a hardware lookup cache
- Search for entries parallel in the TLB
- If a page is present, then it's a hit, get the physical address directly
- Best case:
  PhysicalAddress[31:12] =
  TLB(LogicalAddress[31:12]);
- Otherwise, usual sequence – page table then memory
- Also needs to update the TLB

- Have to be careful about the coherence:
  - On context switch, flush the TLB
  - Manage other bits and coordinate with PT

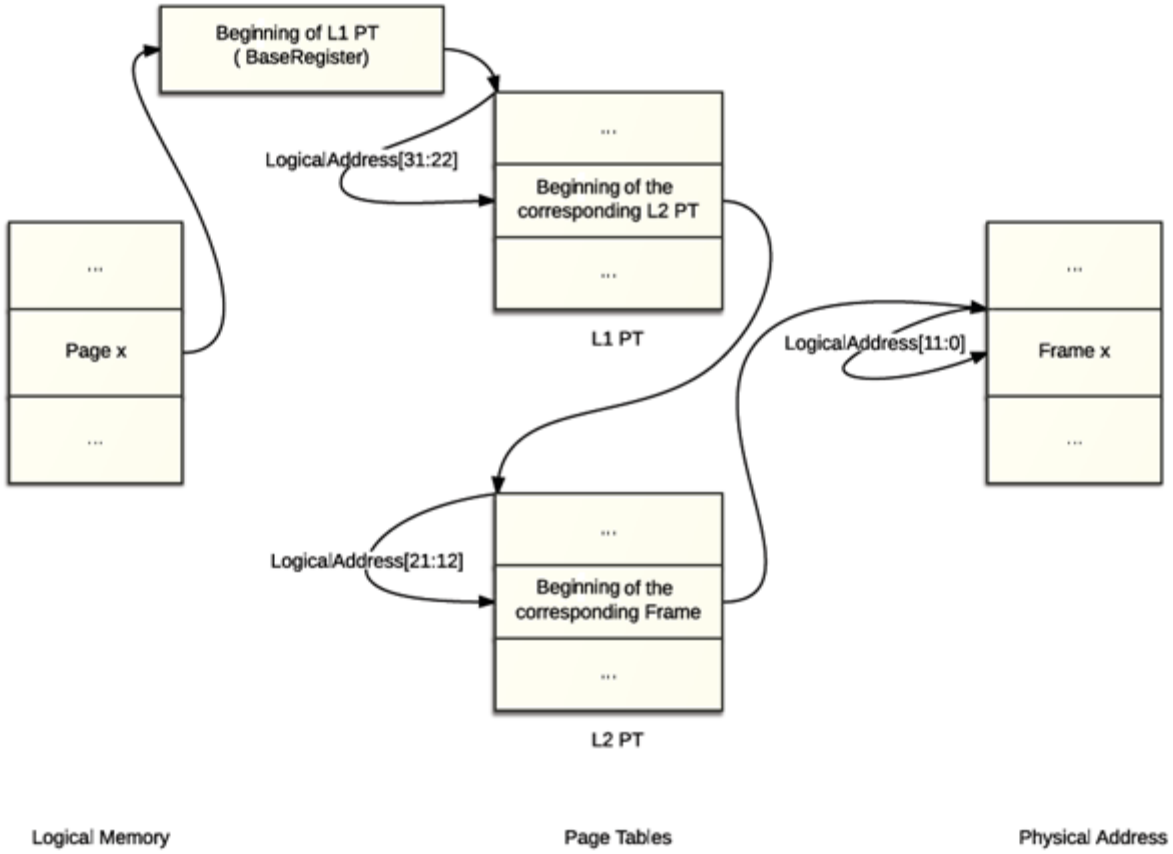


arm

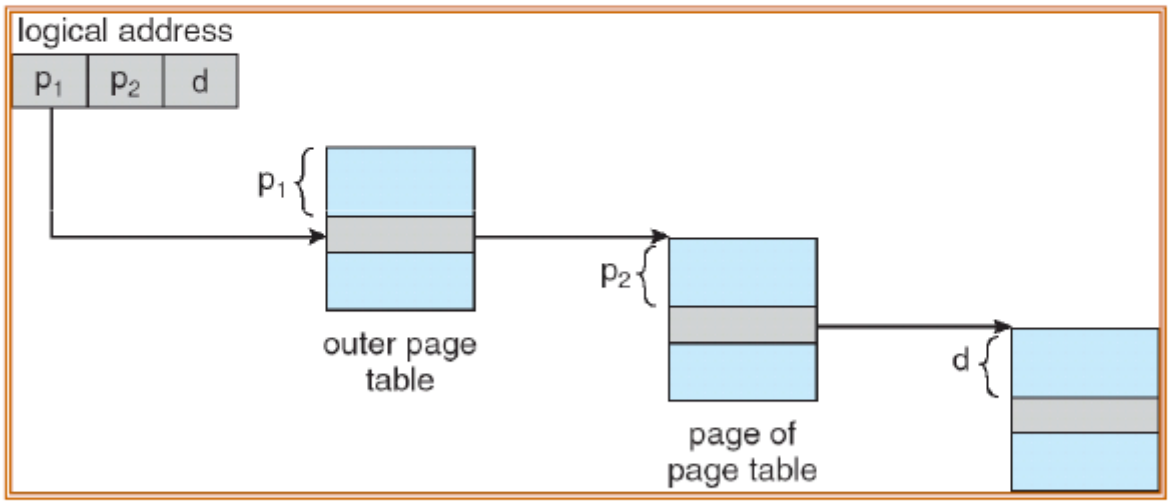
Multilevel Page Table

- PT needs 524288 entries for 32-bit address 4kB pages!
- Repeat the paging “trick”
- PhysicalAddress[31:12] = page_table_L2(page_table_L1(LogicalAddress[31:22]),LogicalAddress[21:12]); PhysicalAddress[11:0] = LogicalAddress[11:0]

- You can also keep the TLB!



Address-translation scheme



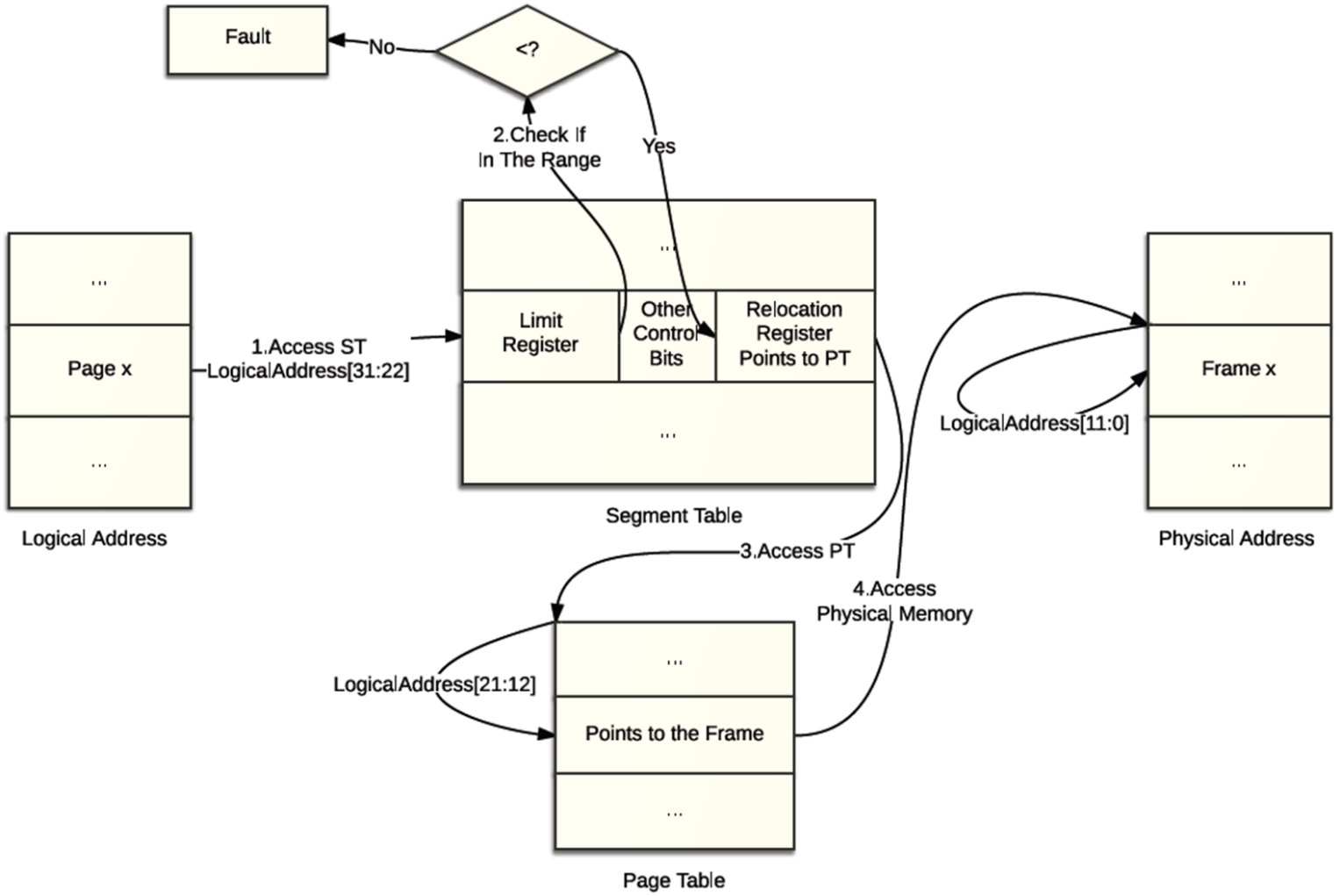
Segmentation With Paging

- Although every segment is independent and can grow, shrink, be protected or be shared without affecting each other, managing (multi)segments, particularly with many variables, can be tricky.
- Paging better solves the fragmentation (no external fragmentation for paging) and allocation/de-allocation is quick.
- Paging however poses extra (rather serious) overhead in terms of address translation and context switching

- Modern OSs can combine both
  - Similar to 2-level paging
  - Protection/Access mode can be decided on a segment basis
  - Page table can be more flexible and swapped out

Segmentation With Paging

- LogicalAddress:(segment-id, page-id,offset)



Next

- Virtual Memory