Operating Systems

Operating Systems

Professor Sina Meraji

U of T

Recap

- Last time we looked at a number of possible scheduling policies
    - First-Come-First-Serve (FCFS)
    - Shortest-Job-First (SJF)
    - Round-robin (RR)
    - Priority scheduling

What do real systems do?

- Combination of
  - Multi-level queue scheduling
    - Typically with RR and priorities
  - Feedback scheduling

New topic:

- Memory management!

Memory Management

- Every active process needs memory
- CPU scheduling allows processes to share (multiplex) the processor
- Must figure out how to share main memory as well
- What should our goals be?
  - Support enough active processes to keep CPU busy
  - Use memory efficiently (minimize wasted memory)
  - Keep memory management overhead small
  - … while satisfying basic requirements

Requirements

- Relocation dont really care about the exact location of data stored in memory…
  - Programmers don’t know what physical memory will be available when their programs run
  - Scheduler may swap processes in/out of memory, need to be able to bring it back in to a different region of memory
  - This implies we will need some type of **address translation**
- Logical Organization
  - Machine accesses memory addresses as a one-dimensional array of bytes
  - Programmers organize code in modules
  - Need to map between these views

More requirements

- Protection
  - A process's memory should be protected from unwanted access by other processes, both intentional and accidental
  - Requires hardware support
- Sharing
  - In some instances, processes need to be able to access the same memory
  - Need ways to specify and control what sharing is allowed
- Physical Organization
  - Memory and Disk form a two-level hierarchy, flow of information between levels must be managed
  - CPU can only access data in registers or memory, not disk

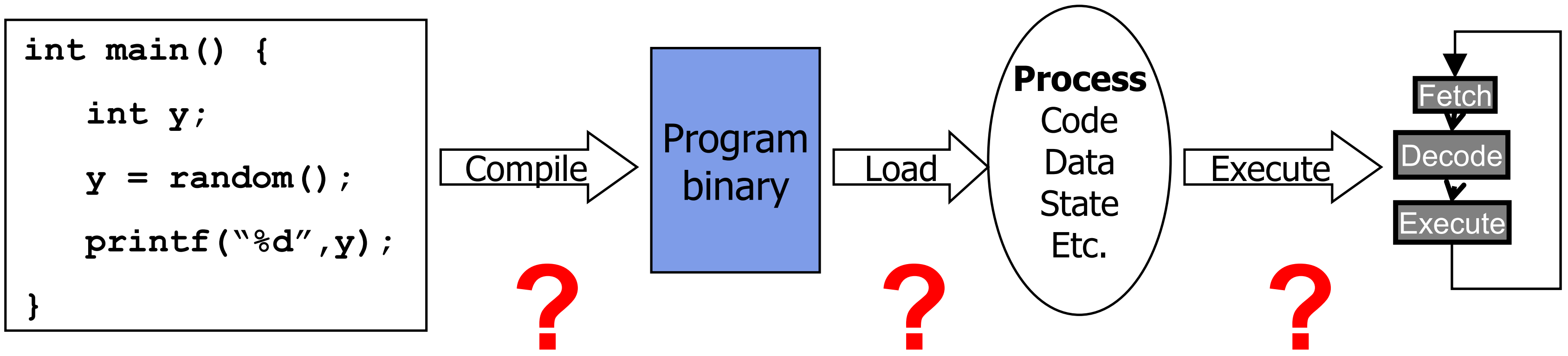
a common interface for software when underlying hardware varies

Meeting the requirements

- Modern systems use *virtual memory*
  - Complicated technique requiring hardware & software support
- We'll build up to virtual memory by looking at some simpler schemes first
  - Fixed partitioning
  - Dynamic partitioning
  - Paging
  - Segmentation
- We'll begin with loading and address translation

Address Binding

- Programs must be in memory to execute
  - Program binary is *loaded* into a process
    - Needs memory for code (instructions) & data
  - Addresses in program must be *translated* to real addresses
    - Programmers use *symbolic* addresses (i.e., variable names) to refer to memory locations
    - CPU fetches from, and stores to, real memory addresses



```
int main() {
   int y;
   y = random();
   printf("%d",y);
}
```

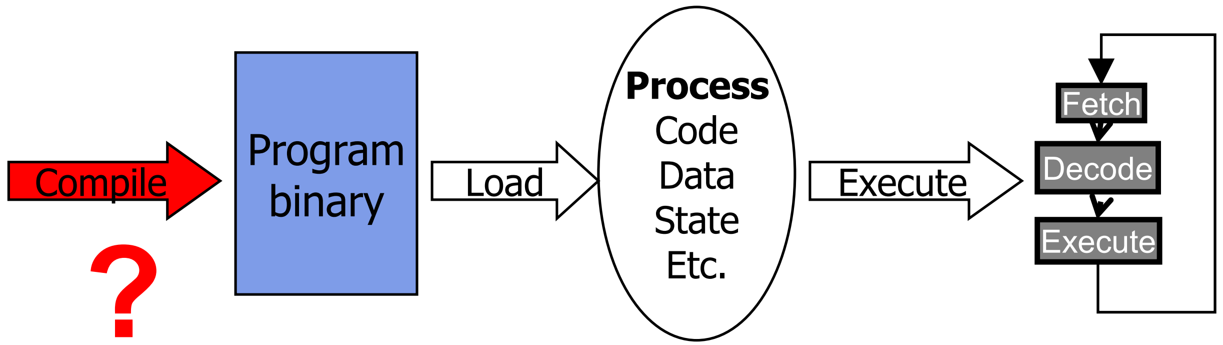
When are addresses bound?

When are addresses bound?

- Compile time
  - Called *absolute code* since binary contains real addresses
  - Disadvantage?
    - Must know what memory process will use during compilation
    - No relocation is possible

need to explicitly specify the exact hardware memory address.
No relocation means no other process cannot use the memory even if the this program is not running

```
int main() {
   int y;
   y = random();
   printf("%d",y);
}
```

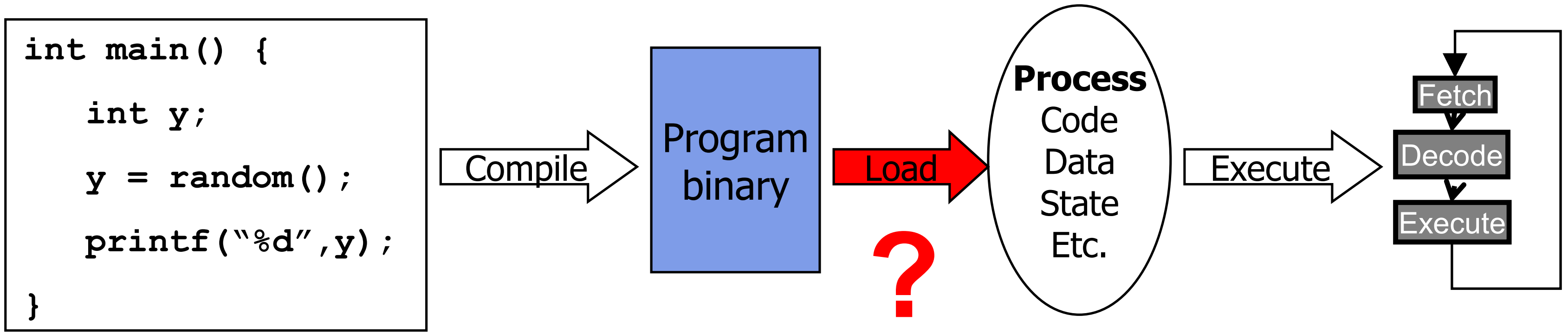


When are addresses bound?

- Load time
  - Compiler translates (binds) symbolic addresses to ***logical, relocatable*** addresses within compilation unit (source file)
  - Linker takes collection of object files and translates addresses to ***logical, absolute*** addresses within executable
    - Resolves references to symbols defined in other files/modules
  - Loader translates logical absolute addresses to ***physical*** addresses when program is loaded into memory
  - Disadvantage?

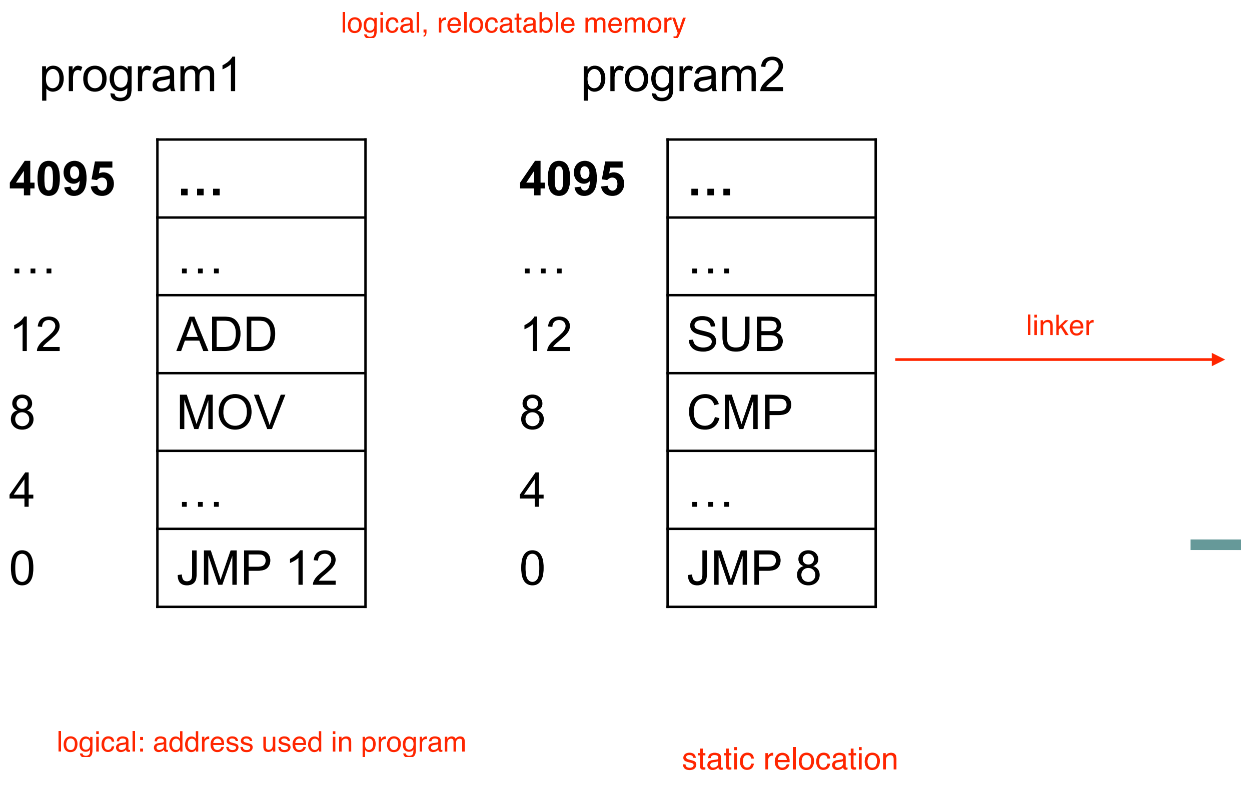
logical absolute address -> physical address when program first loaded into memory

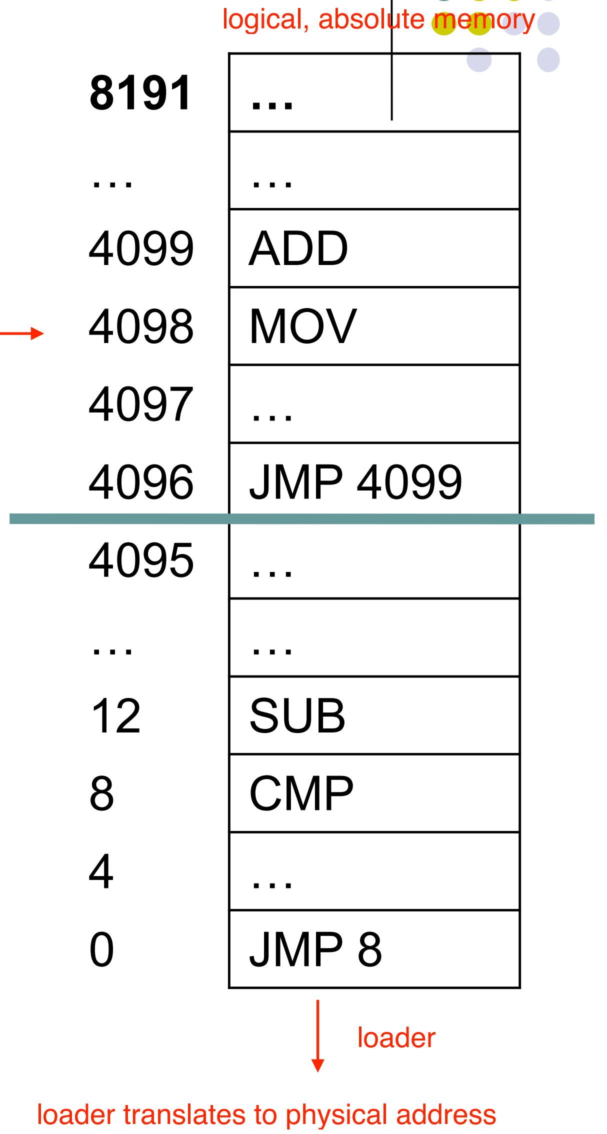
    - Programs can be loaded to different address when they start, but cannot be relocated later



```
int main() {
   int y;
   y = random();
   printf("%d",y);
}
```

Load-Time Binding Example



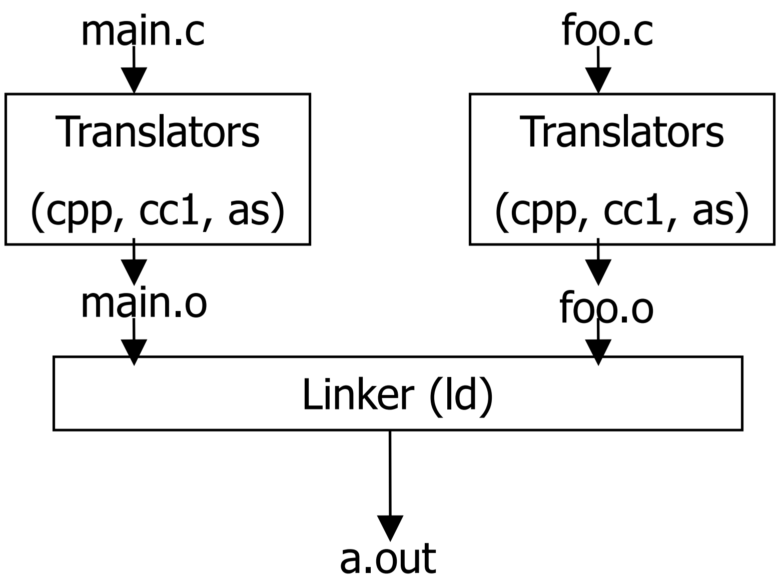


- Programs can be loaded to different address when they start, but cannot be relocated later at runtime

Another point: require contiguous memory (a large memory request may not be satisfied)

A better plan

- Bind addresses at execution time



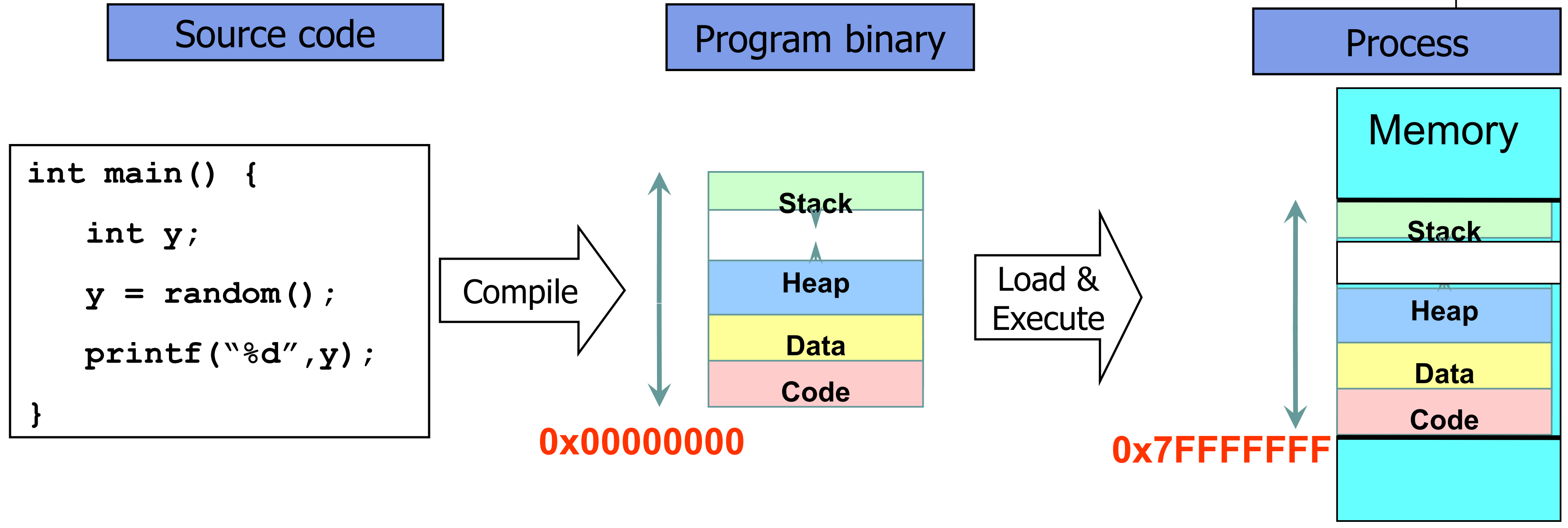
- Executable object file, a.out, contains logical addresses for entire program
  - translated to a real, physical address during execution
  - Flexible, but requires special hardware (as we will see)

Memory management

- Two key problems:
  - How do you map **logical to physical** addresses?
  - How do you allocate physical memory for a process?

at runtime

Address translation: Logical and physical addresses



```
int main() {
    int y;
    y = random();
    printf("%d",y);
}
```

Uses ***symbolic addresses*** (variable names)

Uses ***logical addresses*** (relative to start of stack frame)

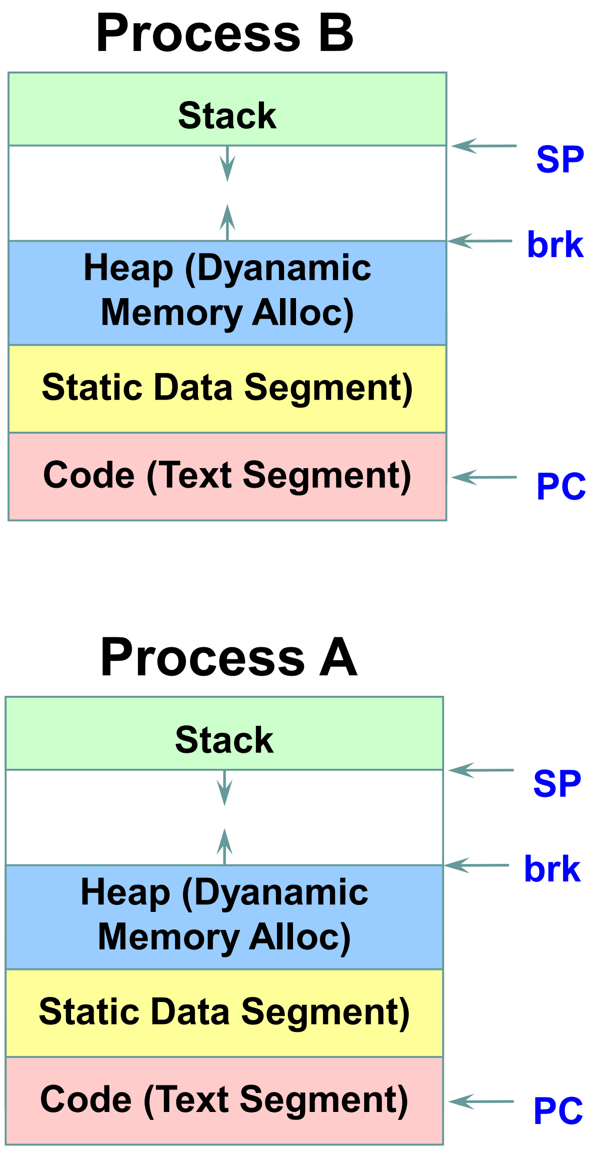
Logical addresses need to be translated to ***physical addresses***

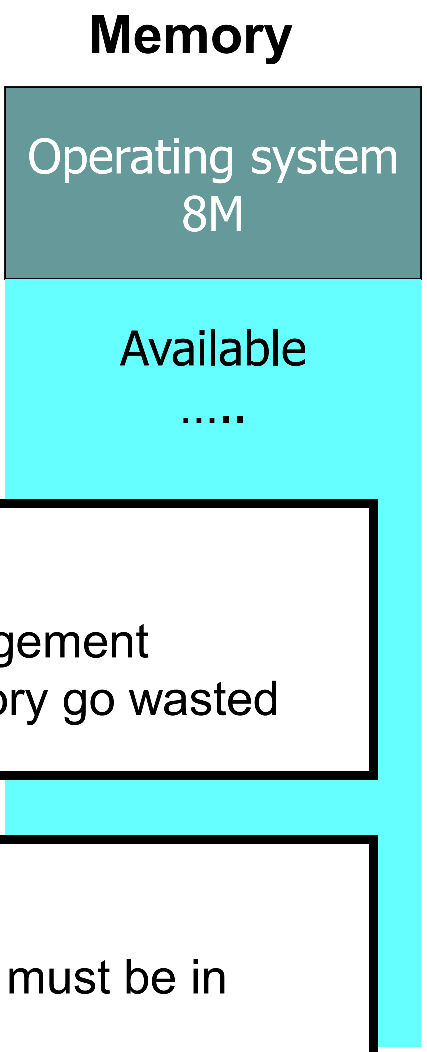
Assume for now:

-Entire address space of process must be in main memory

-A process uses a contiguous chunk of main memory

How to allocate physical memory?





Goals:
- Efficient management
- Don't let memory go wasted

Assumptions:
- Entire process must be in memory to run

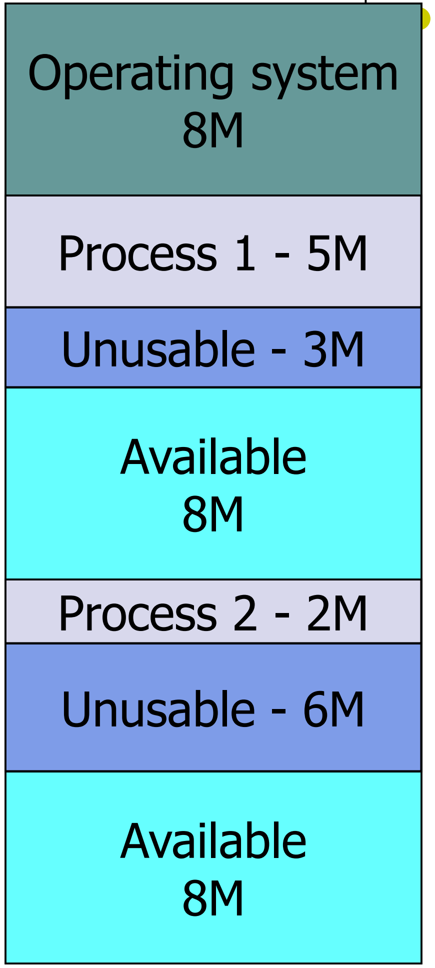
Fixed Partitioning

- Divide memory into regions with fixed boundaries
  - Can be equal-size or unequal-size
- A single process can be loaded into each remaining partition
- Example: 2 processes with 5M and 2M, respectively.

easy to implement

Disadvantages?

- Memory is wasted if process is smaller than partition (*internal fragmentation*)
- Programmer must deal with programs that are larger than partition (*overlays*)

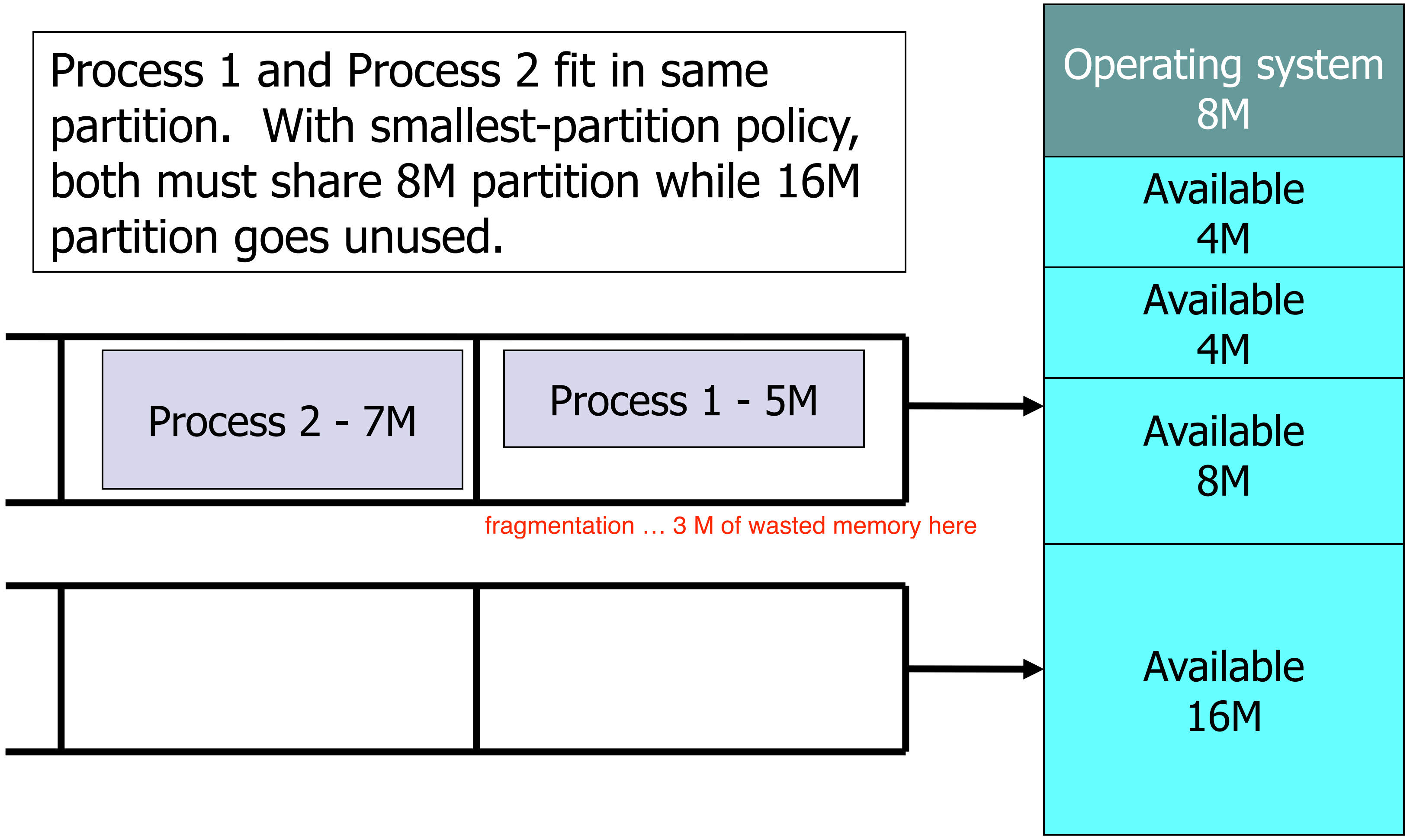


Placement w/ Fixed Partitions

- Number of partitions determines number of active processes
- If all partitions are occupied by waiting processes, swap some out, bring others in
- Equal-sized partitions:
  - Process can be loaded into any available partition
- Unequal-sized partitions:
  - Queue-per-partition, assign process to smallest partition in which it will fit
    - A process always runs in the same size of partition
  - *Single queue*, assign process to smallest *available* partition

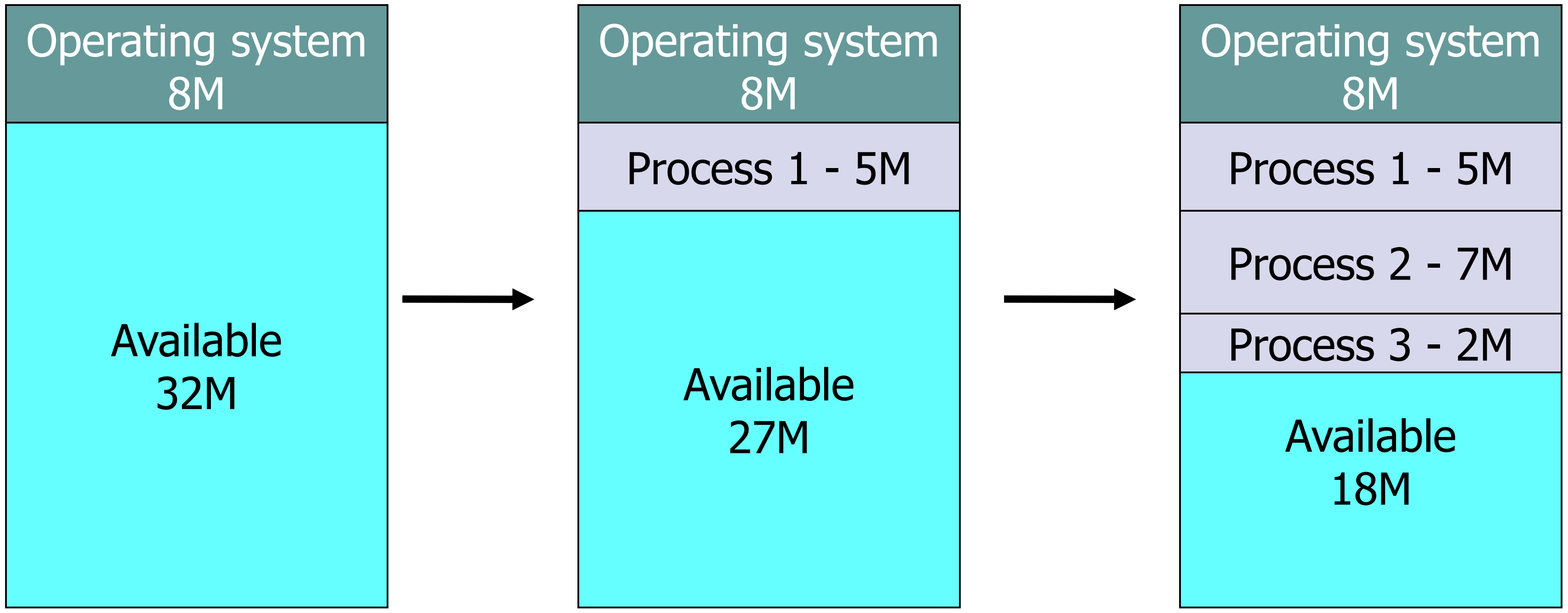
Placement Example (Queue per partition)

Process 1 and Process 2 fit in same partition. With smallest-partition policy, both must share 8M partition while 16M partition goes unused.



Dynamic Partitioning

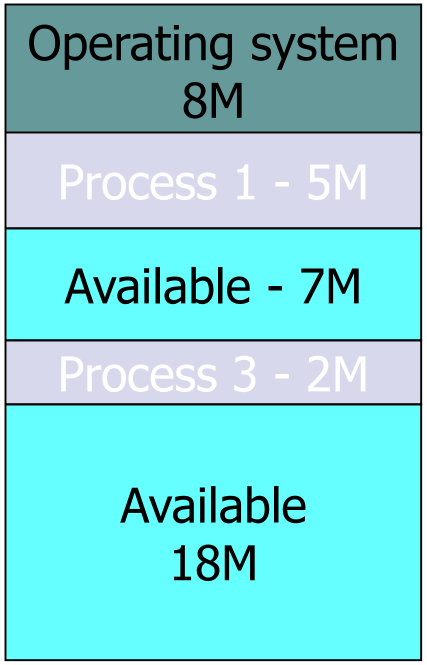
- Partitions vary in length and number over time
- When a process is brought in to memory, a partition of exactly the right size is created to hold it



Disadvantages?

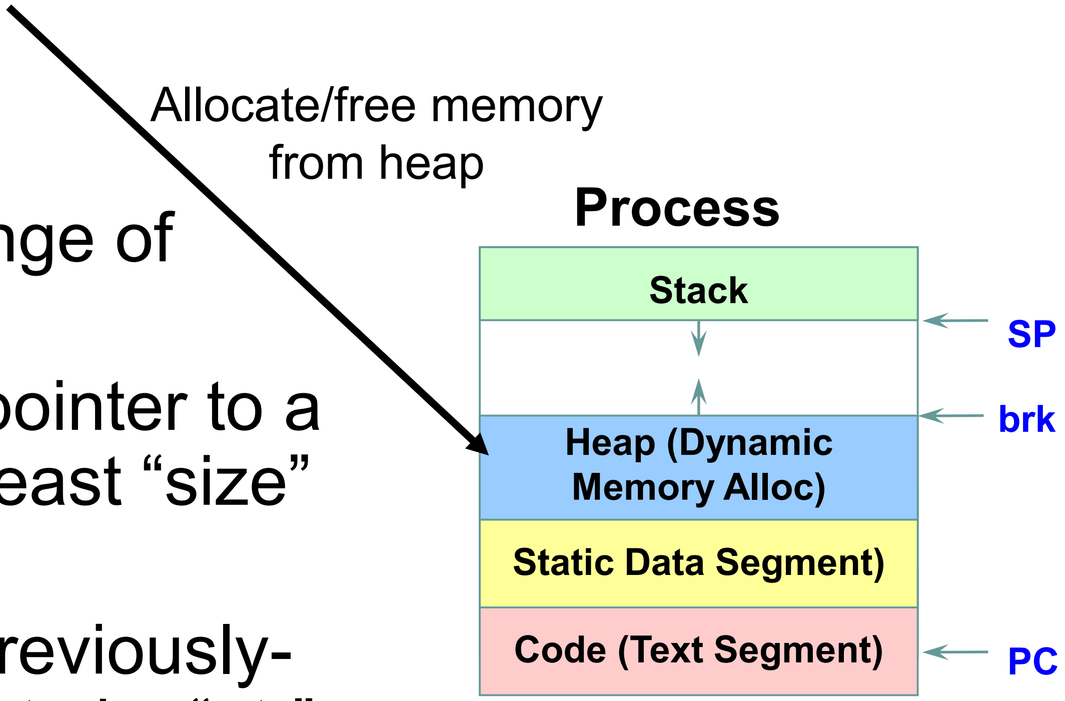
More Dynamic Partitioning

- As processes come and go, "holes" are created
  - Some blocks may be too small for any process
  - This is called *external fragmentation*
- OS may move processes around to create larger chunks of free space
  - E.g. if Process 3 were allocated immediately following Process 1, we would have a 25M free partition
  - This is called *compaction* require updating addresses,
  - Requires processes to be relocatable



Heap Management

- How are malloc() / free() implemented?



- Manage contiguous range of logical addresses
- malloc(size) returns a pointer to a block of memory of at least “size” bytes, or NULL
- free(ptr) releases the previously-allocated block pointed to by “ptr”
- Dynamic partitioning system

Tracking Memory Allocation

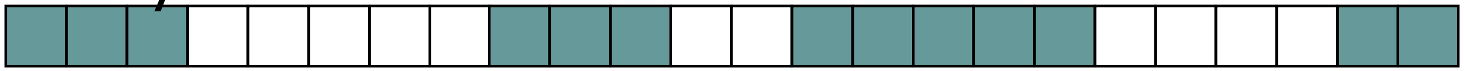
read bitmaps to check for `free` memory, and update the bitmaps

bitmap has to be in memory

- Bitmaps
  - 1 bit per allocation unit
  - “0” == free, “1” == allocated
  - Advantage/Disadvantages?
    - Allocating a N-unit chunk requires scanning bitmap for sequence of N zero’s
    - Slow

linear time to number of bits, traversal is slow

Memory:



Bitmap:

111000001110011111000011

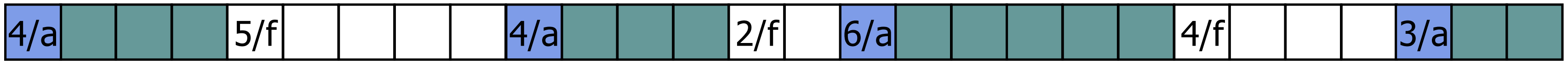
Tracking Allocation (2)

- Free lists
  - Maintain linked list of allocated and free segments
  - List needs memory too. Where do we store it?

list is stored in free memory (RAM). No separate structure for storing pointers

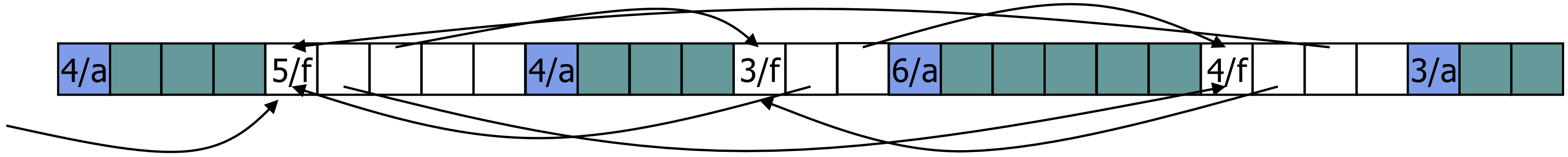
- Implicit list
  - Each block has header that records size and status (allocated or free)
  - Searching for free block is linear in total number of blocks

f: free; a: allocated



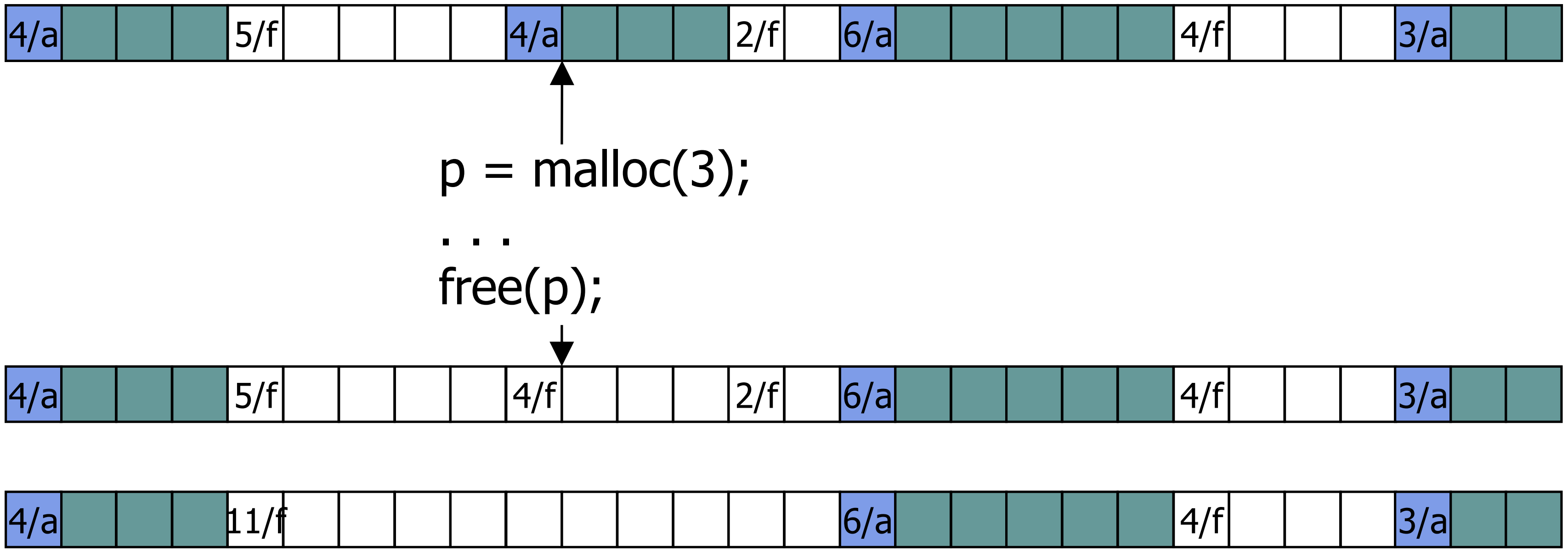
- Explicit list
  - Store pointers in free blocks to create doubly-linked list

track free blocks only, search is linear to total number of free blocks



Freeing Blocks

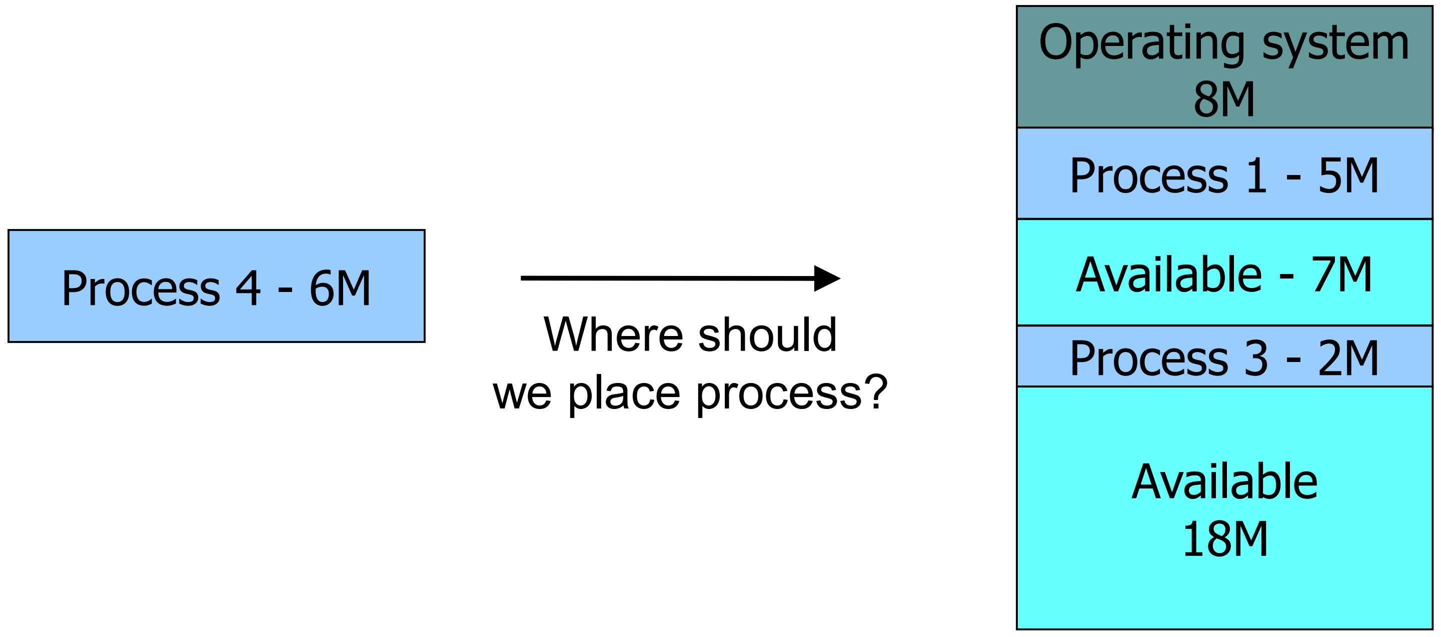
- Adjacent free blocks can be *coalesced* compaction



- Easier if all blocks end with a footer with size/status info (called *boundary tag)*

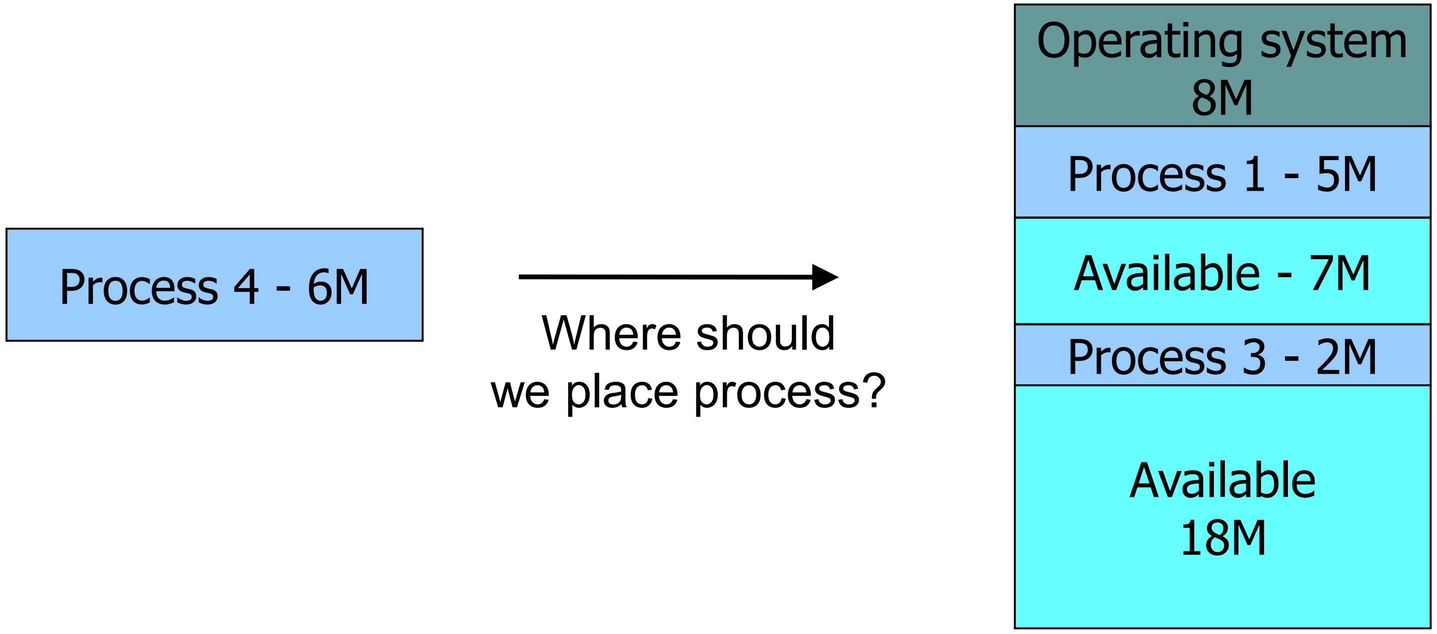
Placement Algorithms

- Compaction is time-consuming and not always possible
- We can reduce the need for it by being careful about how memory is allocated to processes over time
- Given multiple blocks of free memory of sufficient size, how should we choose which one to use?



Placement Algorithms

- **First-fit -** choose first block that is large enough; search can start at beginning, or where previous search ended (called next-fit)
- **Best-fit -** choose the block that is closest in size to the request
- **Worst-fit** – choose the largest block
- **Quick-fit** – keep multiple free lists for common block sizes



Comparing Placement Algs.

- First-fit
  - Simplest, and often fastest and most efficient
  - May leave many small fragments near start of memory that must be searched repeatedly increase search time over time
- Best-fit
  - left-over fragments tend to be small (unusable)
  - In practice, similar storage utilization to first-fit
- Worst-fit
  - Not as good as best-fit or first-fit in practice
- Quick-fit
  - Great for fast allocation, generally harder to coalesce

Problems with Partitioning

- With fixed partitioning, internal fragmentation and need for overlays are big problems
  - Scheme is too inflexible
- With dynamic partitioning, external fragmentation and management of space are major problems
- Basic problem is that processes must be allocated to contiguous blocks of physical memory
  - Hard to figure out how to size these blocks given that processes are not all the same
- We'll look now at *paging* as a solution

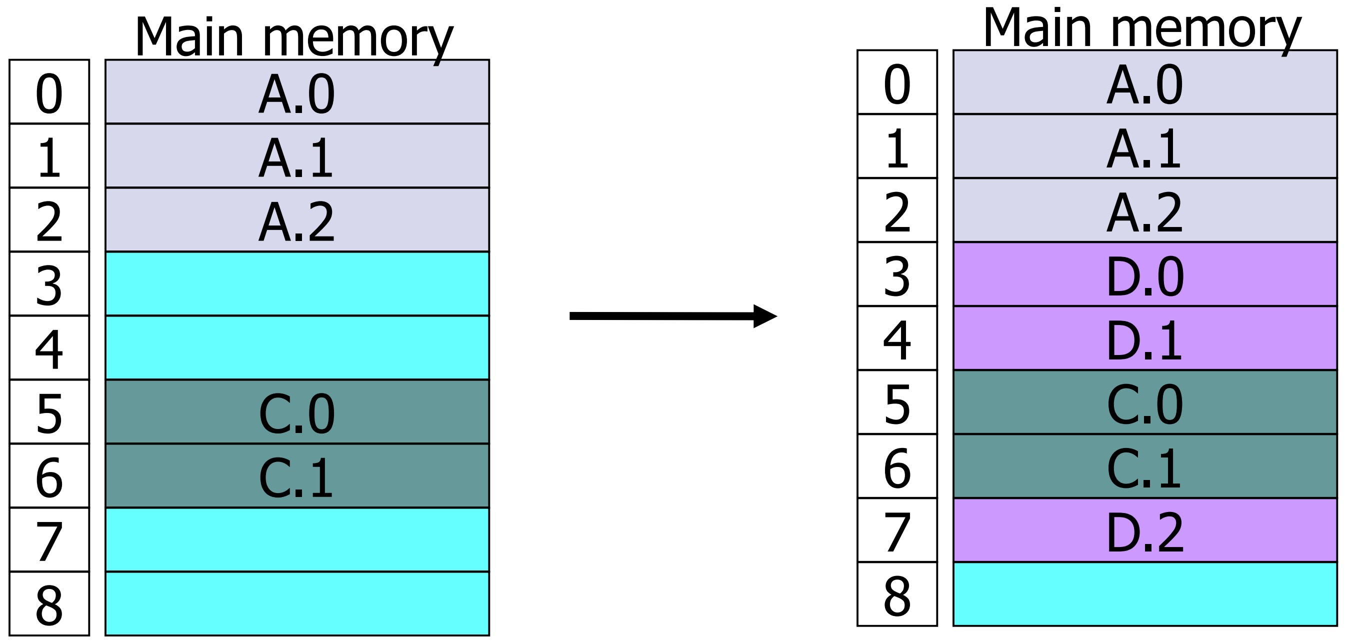
Paging

frame and page has same size

- Partition memory into equal, fixed-size chunks
  - These are called *page frames* or simply *frames*
- Divide processes' memory into chunks of the same size
  - These are called *pages*
- Possible page frame sizes are restricted to powers of 2 to simplify translation

Example of Paging

Suppose a new process, D, arrives needing 3 frames of memory

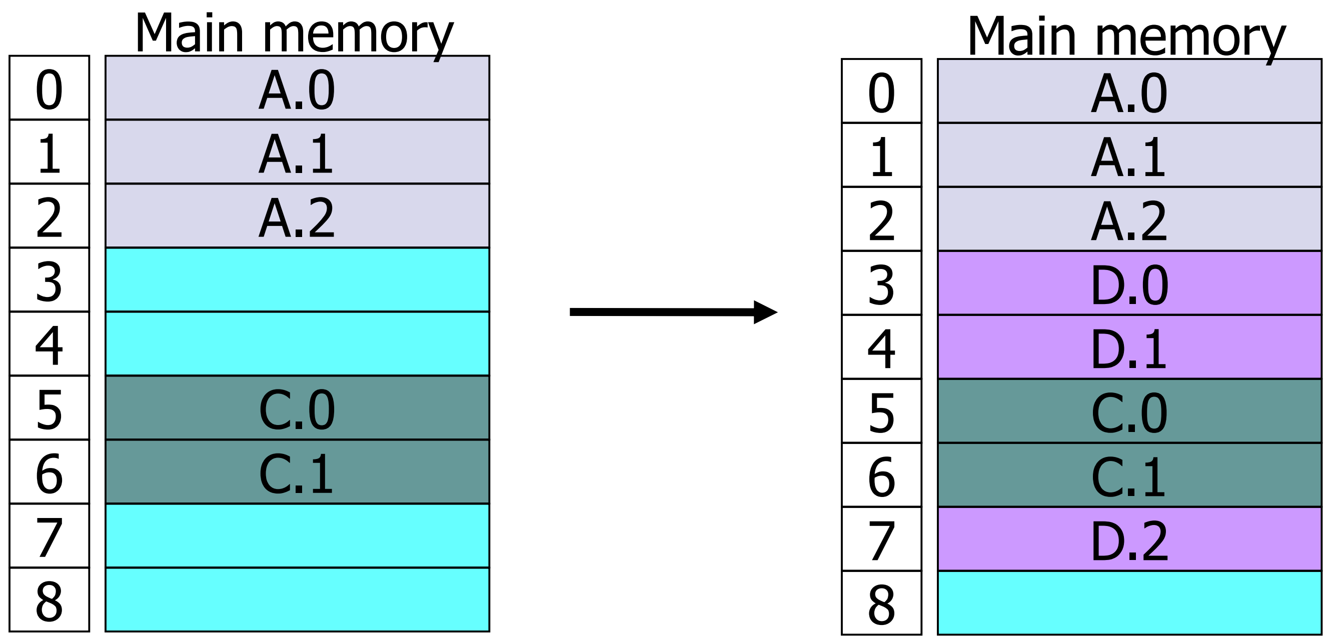


For each process, requires hash table for mapping
page -> frame

- We can fit Process D into memory, even though we don't have 3 contiguous frames available!

Example of Paging

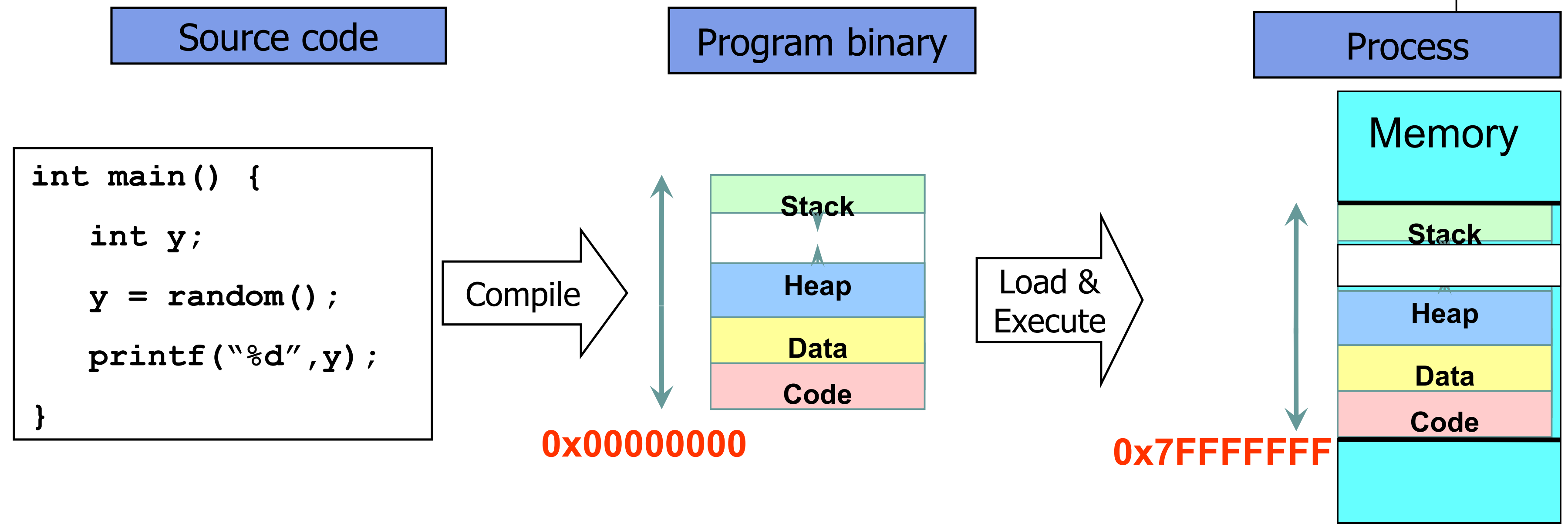
Suppose a new process, D, arrives needing 3 frames of memory



- Is there fragmentation with paging?
  - External fragmentation is eliminated
  - Internal fragmentation is at most a part of one page per process

in last page of each process

Address translation



```
int main() {
   int y;
   y = random();
   printf("%d",y);
}
```

Uses ***symbolic addresses*** (variable names)

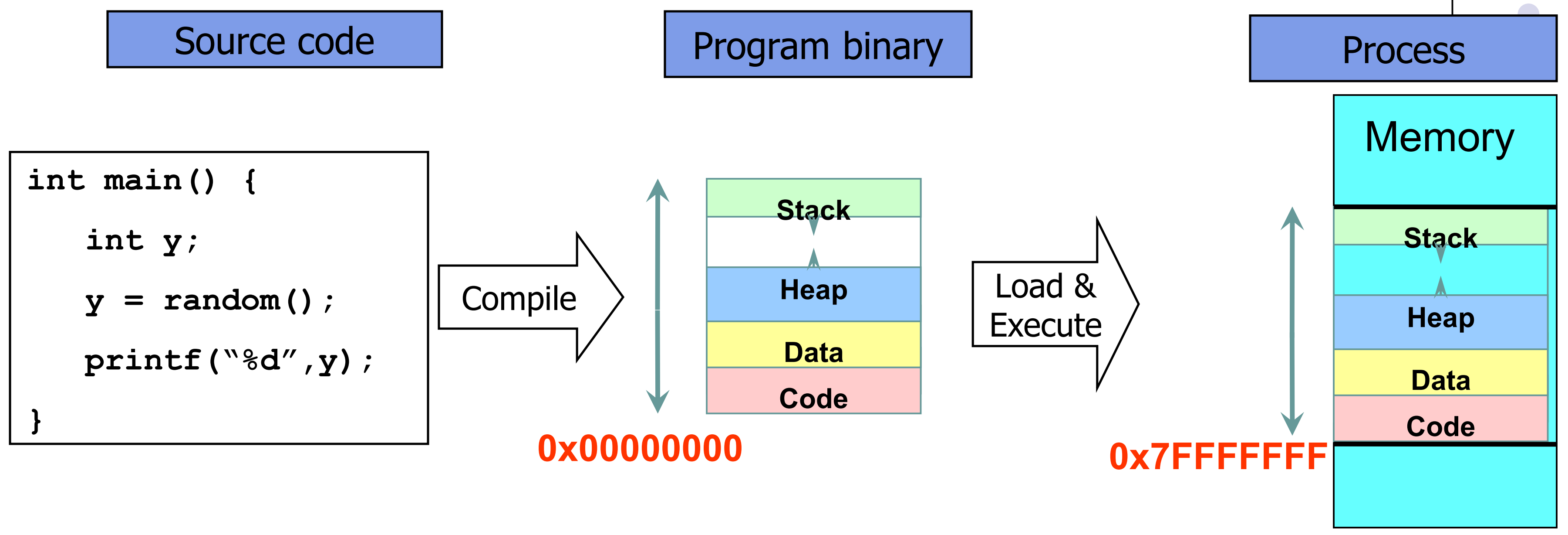
Uses ***logical addresses*** (relative to start of stack frame)

Logical addresses need to be translated to ***physical addresses***

Address translation

- Swapping and compaction require a way to change the physical memory addresses a process refers to
- Really, need dynamic relocation (aka execution-time binding of addresses)
  - process refers to *relative* addresses, hardware translates to physical address as instruction is executed

Address translation



```
int main() {
   int y;
   y = random();
   printf("%d",y);
}
```

Uses ***symbolic addresses*** (variable names)

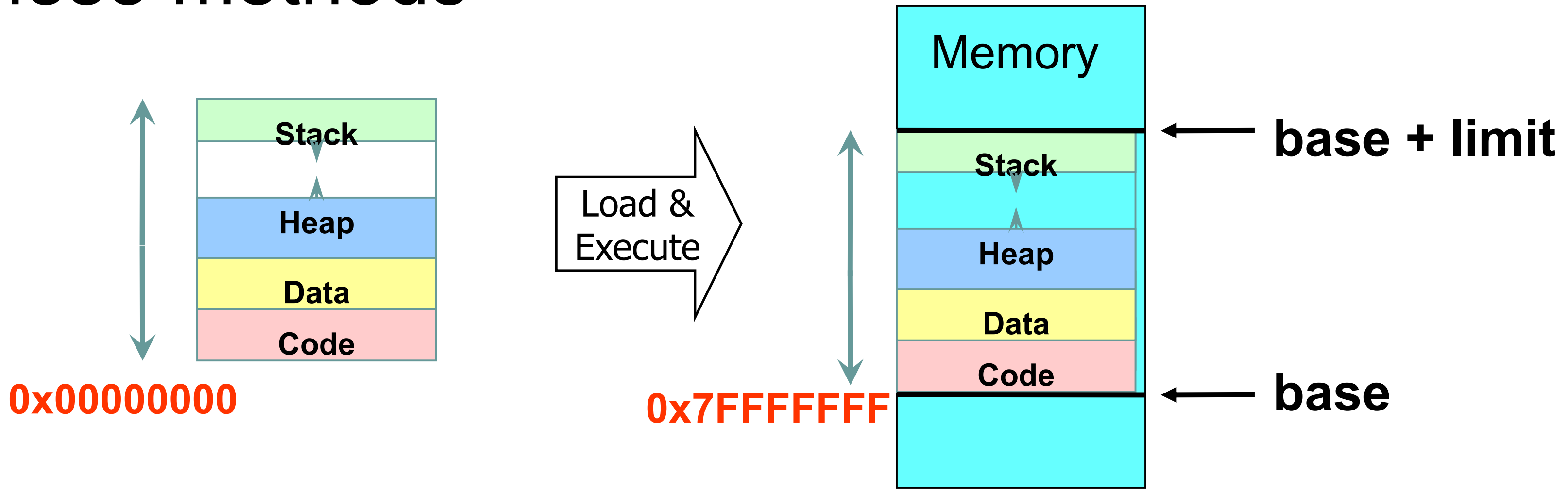
Uses ***logical addresses*** (relative to start of stack frame)

Logical addresses need to be translated to ***physical addresses***

- How does address translation work for
  - Static/dynamic partitioning?
  - Paging?

Address translation: Partitioning schemes

- All memory used by process is *contiguous* in these methods



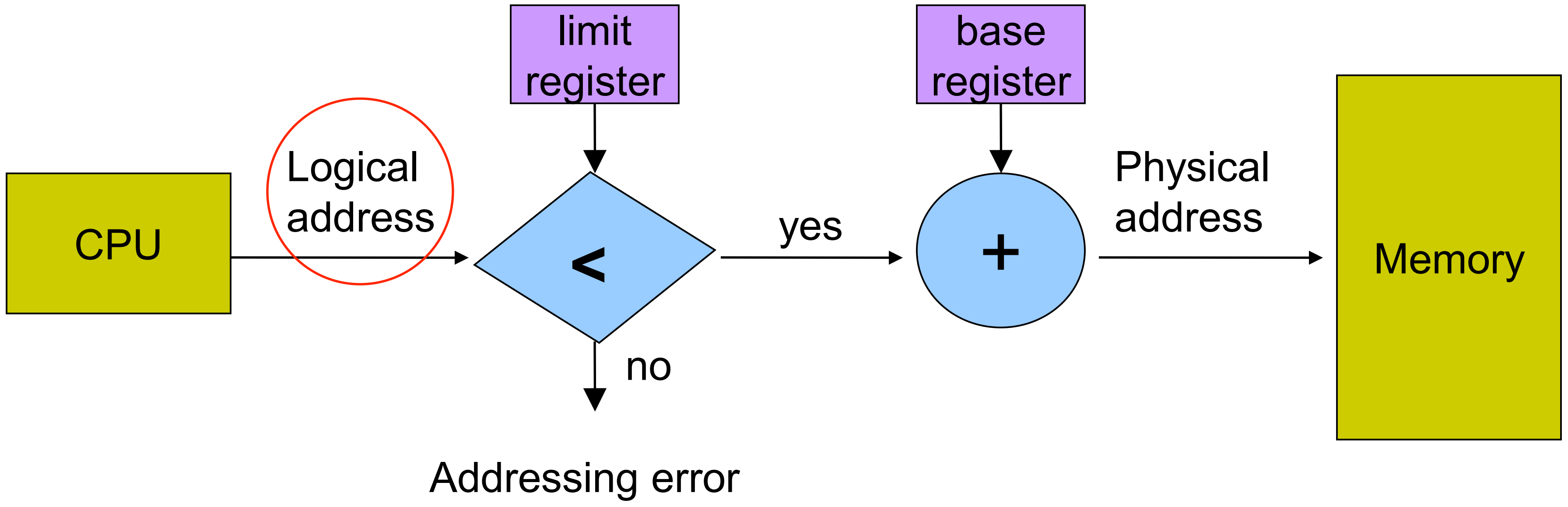
- Basic idea: add relativ
(base) address to form
- 2 registers, “base” and

Why do we need to keep track of limit?

Hardware for Relocation

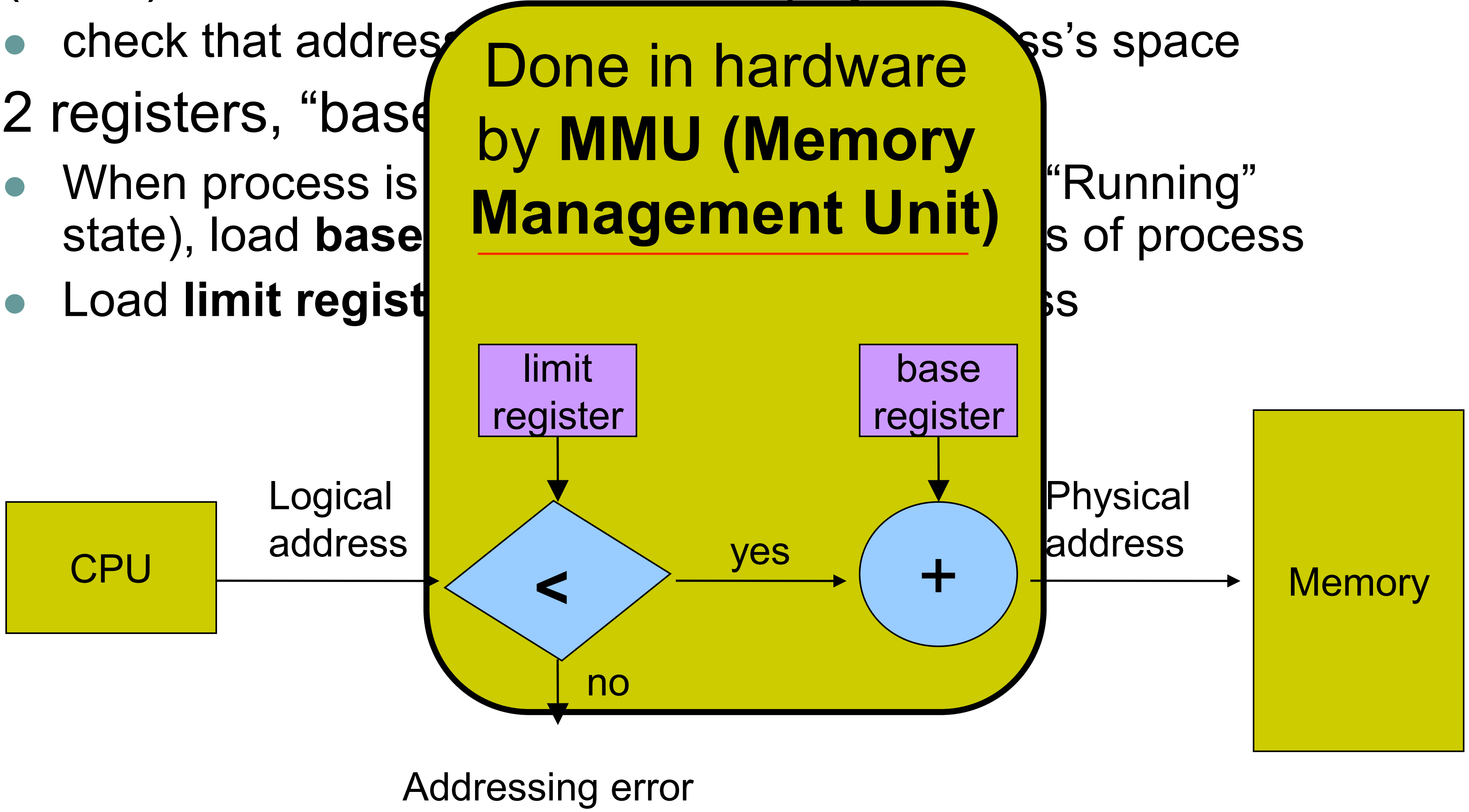
base, limit register specific to memory

- Basic idea: add relative address to process starting (base) address to form real, or physical, address
  - check that address generated is within process's space
- 2 registers, "base" and "limit"
  - When process is assigned to CPU (i.e., set to "Running" state), load **base register** with starting address of process
  - Load **limit register** with last address of process

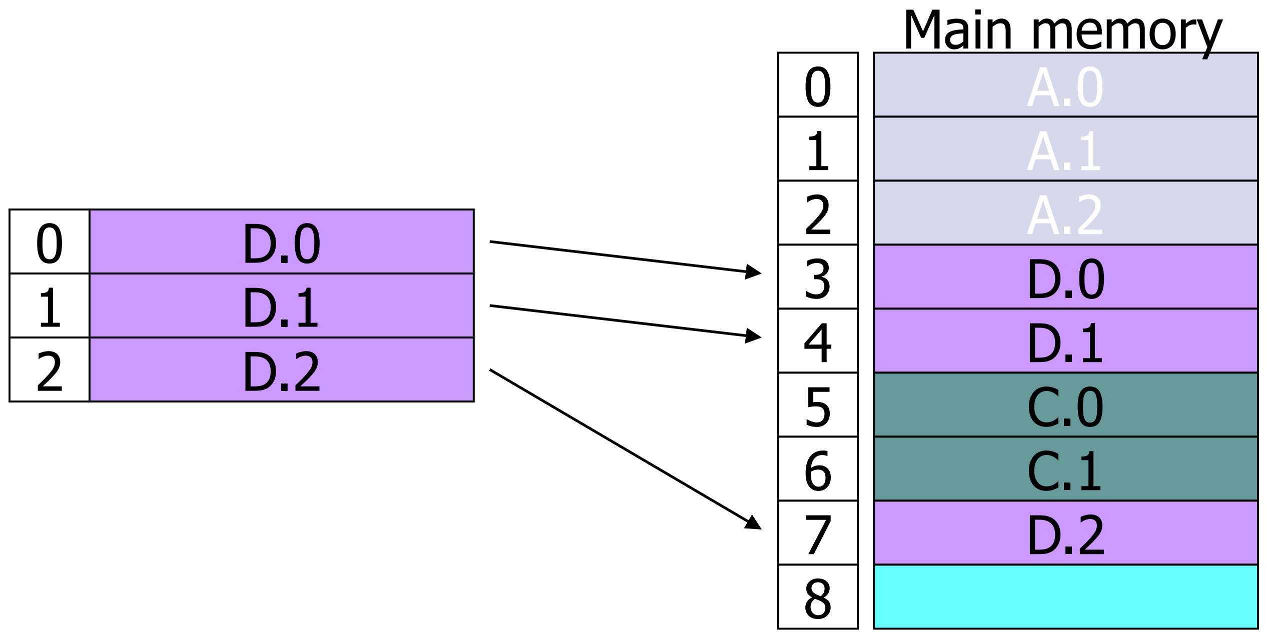


Hardware for Relocation

- Basic idea: add relative address to process starting (base) address to form real, or physical, address
  - check that addres ss's space
- 2 registers, "base
  - When process is "Running" state), load base s of process
  - Load limit regist ss



Address translation for Paging



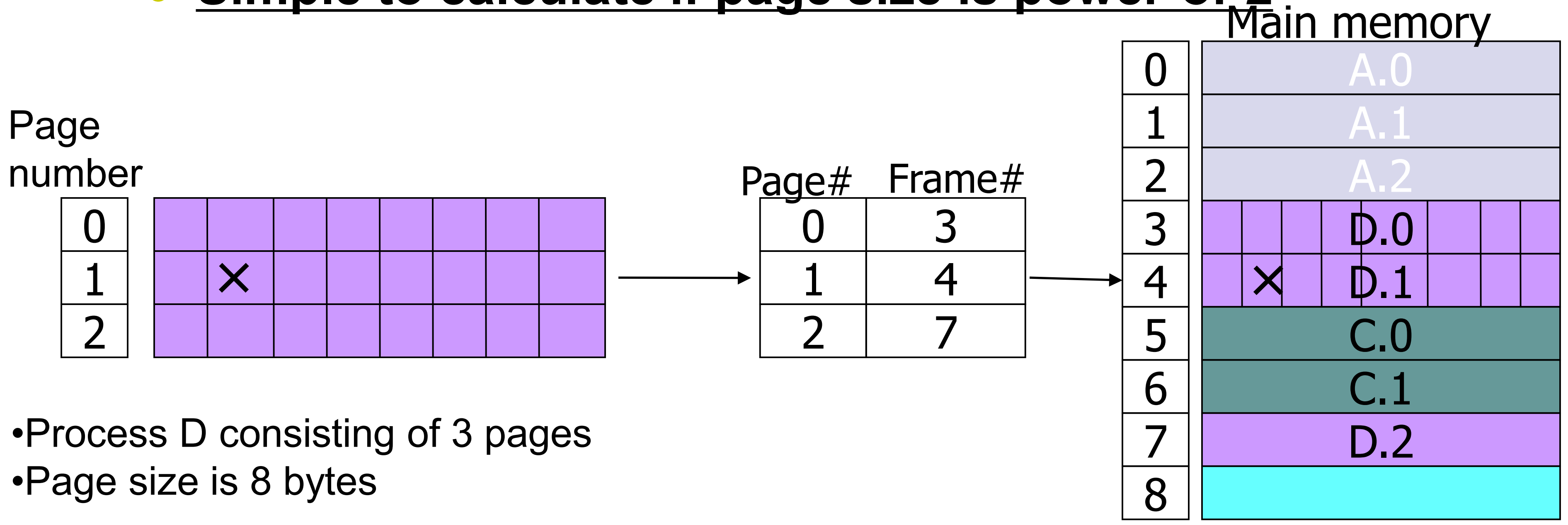
- Need more than base & limit registers now
- Operating system maintains a *page table* for each process

What does an address specify now?

Support for Paging

- Operating system maintains *page table* for each process
  - Page table records which physical frame holds each page
  - virtual addresses are now *page number + page offset*
    - *page number = ?*
      - *=vaddr / page_size*
    - *page offset = ?*
      - *vaddr % page_size*
    - **Simple to calculate if page size is power-of-2**

no need for base, limit anymore,
virtual memory in page can be mapped to
physical memory with
1. page —page table—> frame number
2. offset
phyMem = frameNum * sizeofFrame + offset



•Process D consisting of 3 pages
•Page size is 8 bytes

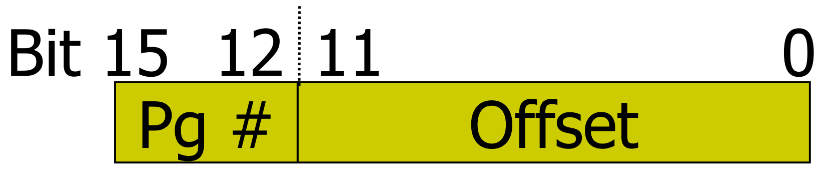
Support for Paging

- Operating system maintains *page table* for each process
  - Page table records which physical frame holds each page
  - virtual addresses are now *page number + page offset*
    - *page number = vaddr / page_size*
    - *page offset = vaddr % page_size*
    - **Simple to calculate if page size is power-of-2**

  - On each memory reference, processor translates page number to frame number and adds offset to generate a physical address
  - Keep a "page table base register" to quickly locate the page table for the running process

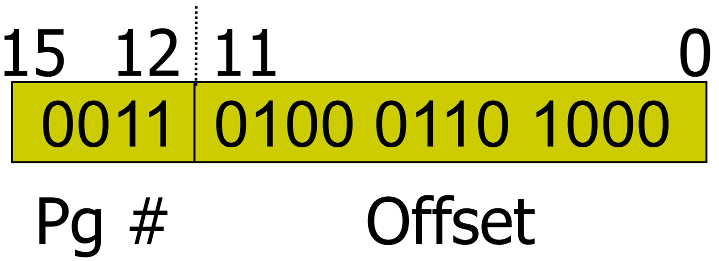
Example Address Translation

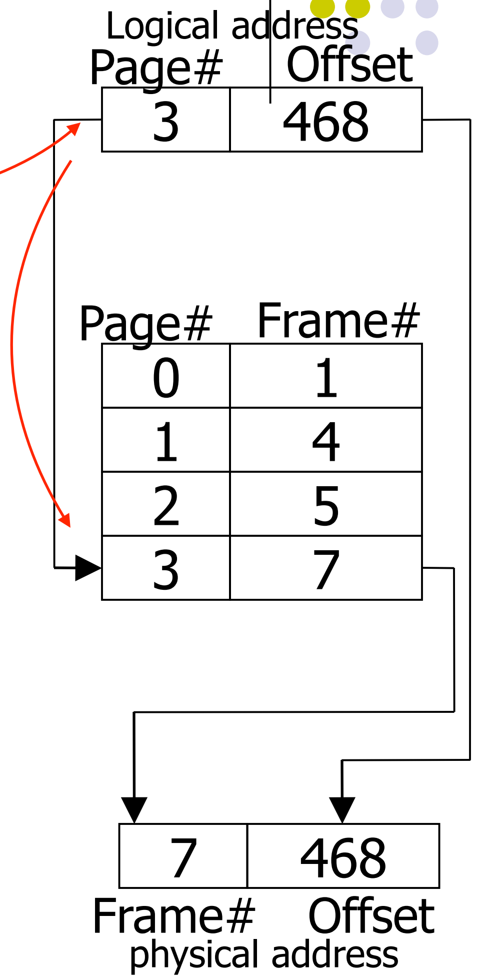
- Suppose addresses are 16 bits, pages are 4K (4096 bytes)
  - How many bits of the address do we need for offset?
    - 12 bits (2^12 = 4096)   offset varies from 0 -> 2^12 since page is 4K
  - What is the maximum number of pages for a process?
    - 2^4



Example Address Translation

- To translate virtual address: 0x3468
  - Extract page number (high-order 4 bits) -> page = vaddr >> 12 == 3
  - Get frame number from page table
  - Combine frame number with page offset
    - offset = vaddr % 4096
    - paddr = frame * 4096 + offset
      - paddr = (frame << 12) | offset





page #: specified by index of PTE in page table

Page Table Entries (PTE)

2^6 pages?

32-bit

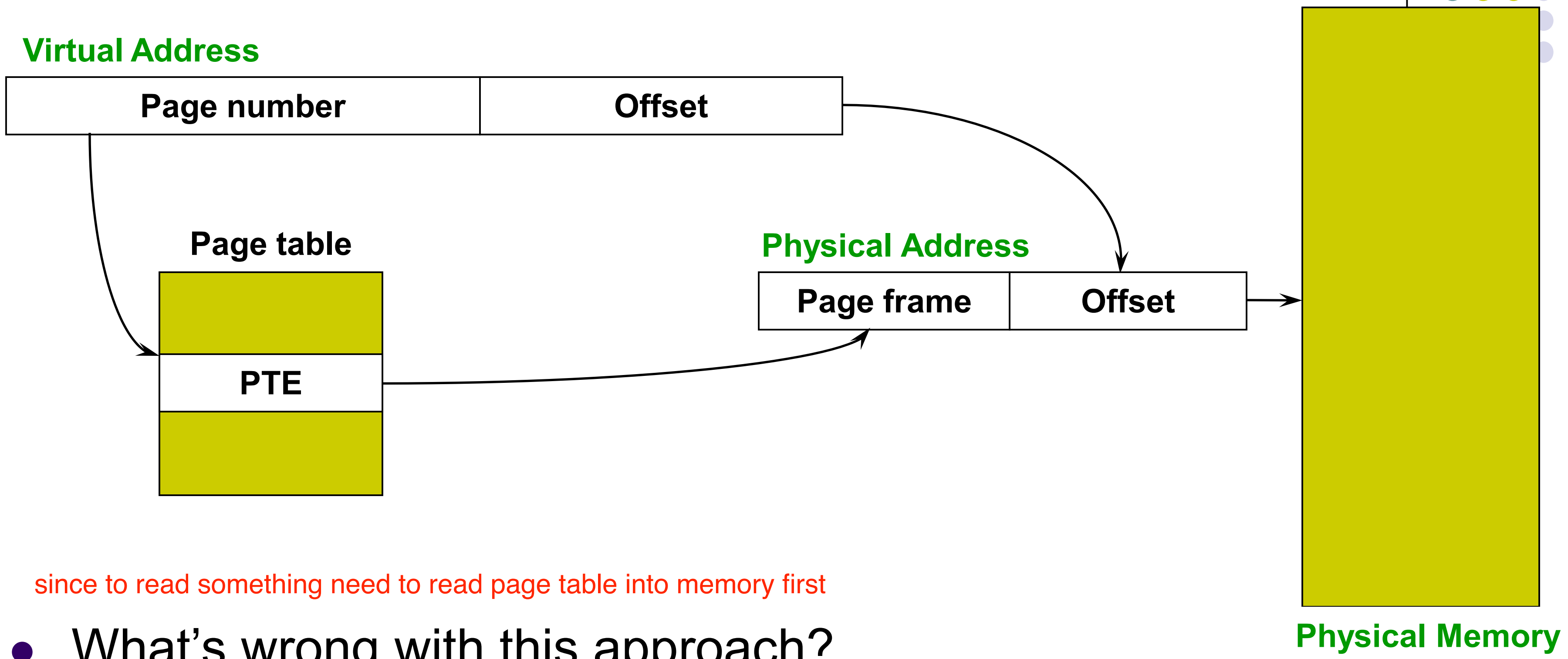
1	1	1	3	26
M	R	V	Prot	Page Frame Number

- Page table entries (PTEs) control mapping
  - Modify bit (M) says whether or not page has been written
    - Set when a write to a page occurs
  - Reference bit (R) says whether page has been accessed
    - Set when a read or write to the page occurs
  - Valid bit (V) says whether PTE can be used

    see if the current page is used, since only part of memory is in memory in real implementation
    - Checked on each use of virtual address

      if V invalid, implies page not in memory, so page fault
  - Protection bits specify what operations are allowed on page
    - Read/write/execute
  - Page frame number (PFN) determines physical page
  - Not all bits are provided by all architectures

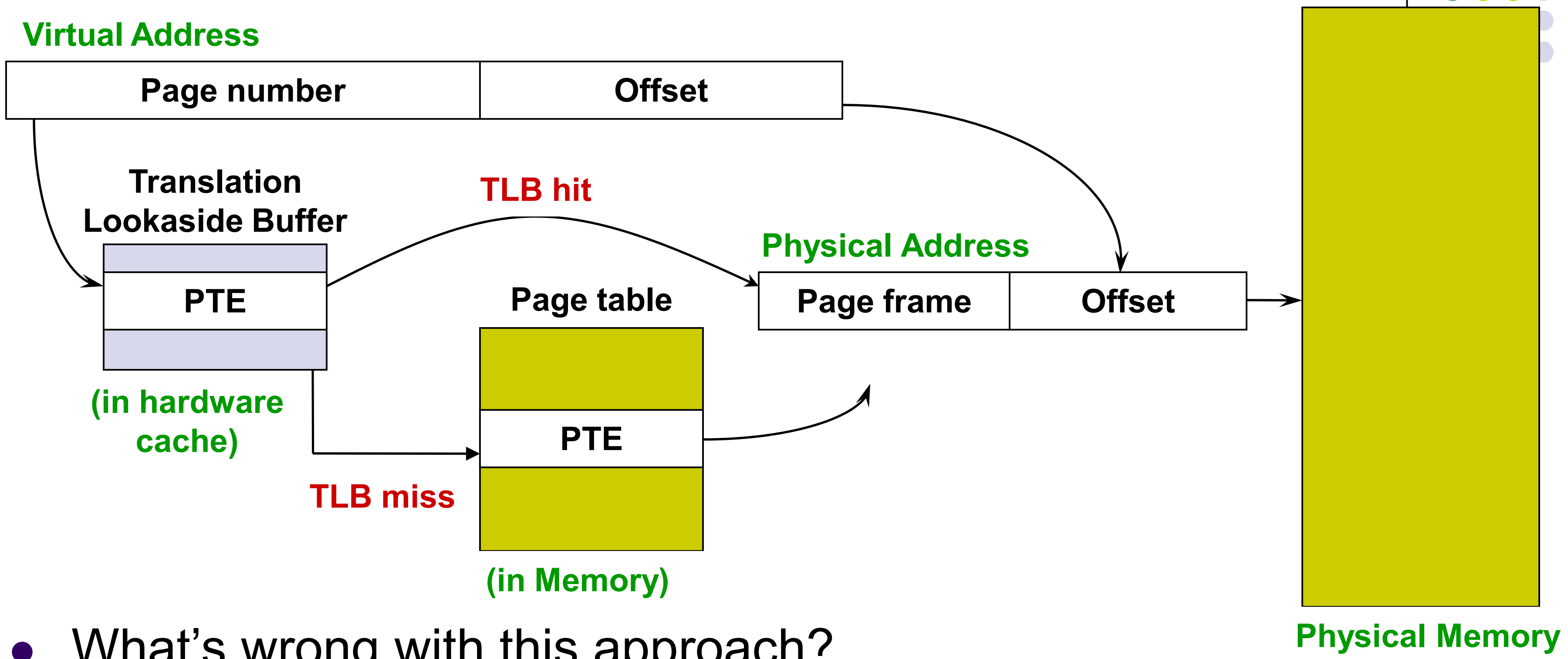
Page Lookups Overview



since to read something need to read page table into memory first

- What's wrong with this approach?
  - Need 2 references for address lookup (first page table, then actual memory)
- Idea: Use hardware cache of page table entries
  - Translation Lookaside Buffer (TLB) portion of page table for different active processes
  - Small hardware cache of recently used translations

Page Lookups Overview



- What's wrong with this approach?
  - Need 2 references for address lookup (first page table, then actual memory)
- Idea: Use hardware cache of page table entries
  - Translation Lookaside Buffer (TLB)
  - Small, fully-associative hardware cache of recently used translations

TLBs

Translate virtual page #s into PTEs (not physical addrs)

- Can be done in a single machine cycle

- TLBs implemented in hardware
  - Fully associative cache (all entries looked up in parallel)
  - Cache tags are virtual page numbers
  - Cache values are PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address

TLBs

- TLBs are small (64 – 1024 entries)
- Still, address translations for most instructions are handled using the TLB
  - **>99% of translations**, but there are misses (TLB miss)…

- TLBs exploit **locality**
  - Processes only use a handful of pages at a time
    - 16-48 entries/pages (64-192K)
    - Only need those pages to be “mapped”
  - Hit rates are therefore very important

Managing TLBs

- Who places translations into the TLB (loads the TLB)?
  - Hardware (Memory Management Unit)
  - Software loaded TLB (OS)

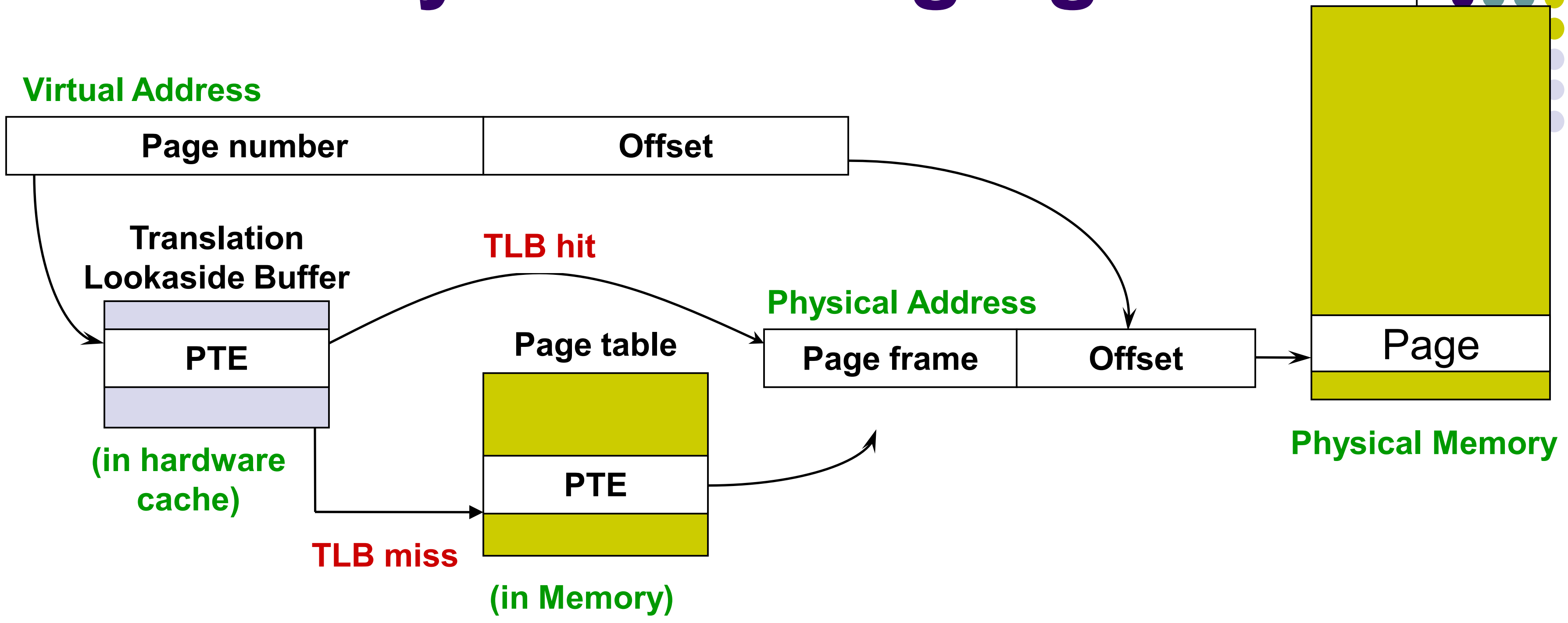
Managing TLBs

- Who places translations into the TLB (loads the TLB)?
  - Hardware (Memory Management Unit)
    - Knows where page tables are in main memory
    - OS maintains tables, HW accesses them directly
    - Tables have to be in HW-defined format (inflexible)
  - Software loaded TLB (OS)
    - TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
    - Must be fast (but still 20-200 cycles)
    - CPU ISA has instructions for manipulating TLB
    - Tables can be in any format convenient for OS (flexible)

Managing TLBs (2)

- OS ensures that TLB and page tables are consistent
  - When it changes the protection bits of a PTE, it needs to invalidate the PTE if it is in the TLB
- Reload TLB on a process context switch
  - Invalidate all entries
  - Why?
- When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  - Choosing PTE to evict is called the TLB replacement policy
  - Implemented in hardware, often simple

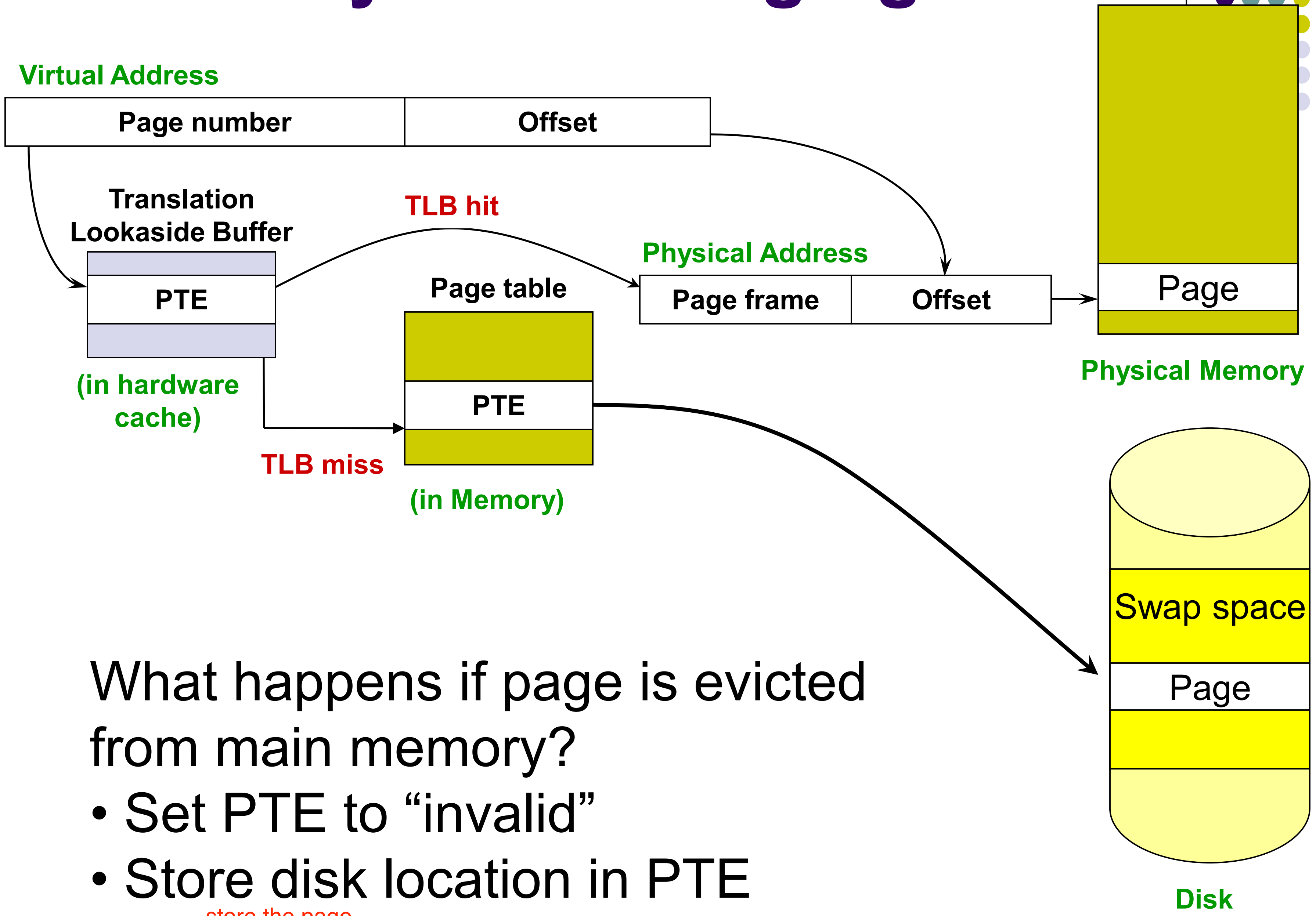
Summary so far: Paging



What happens if not all pages of all processes fit into physical memory?

i.e. valid bit is invalid.. implies page not in memory so have to swap in from disk

Summary so far: Paging



What happens if page is evicted from main memory?

- Set PTE to “invalid”
- Store disk location in PTE

store the page

How much space does a page table take up?

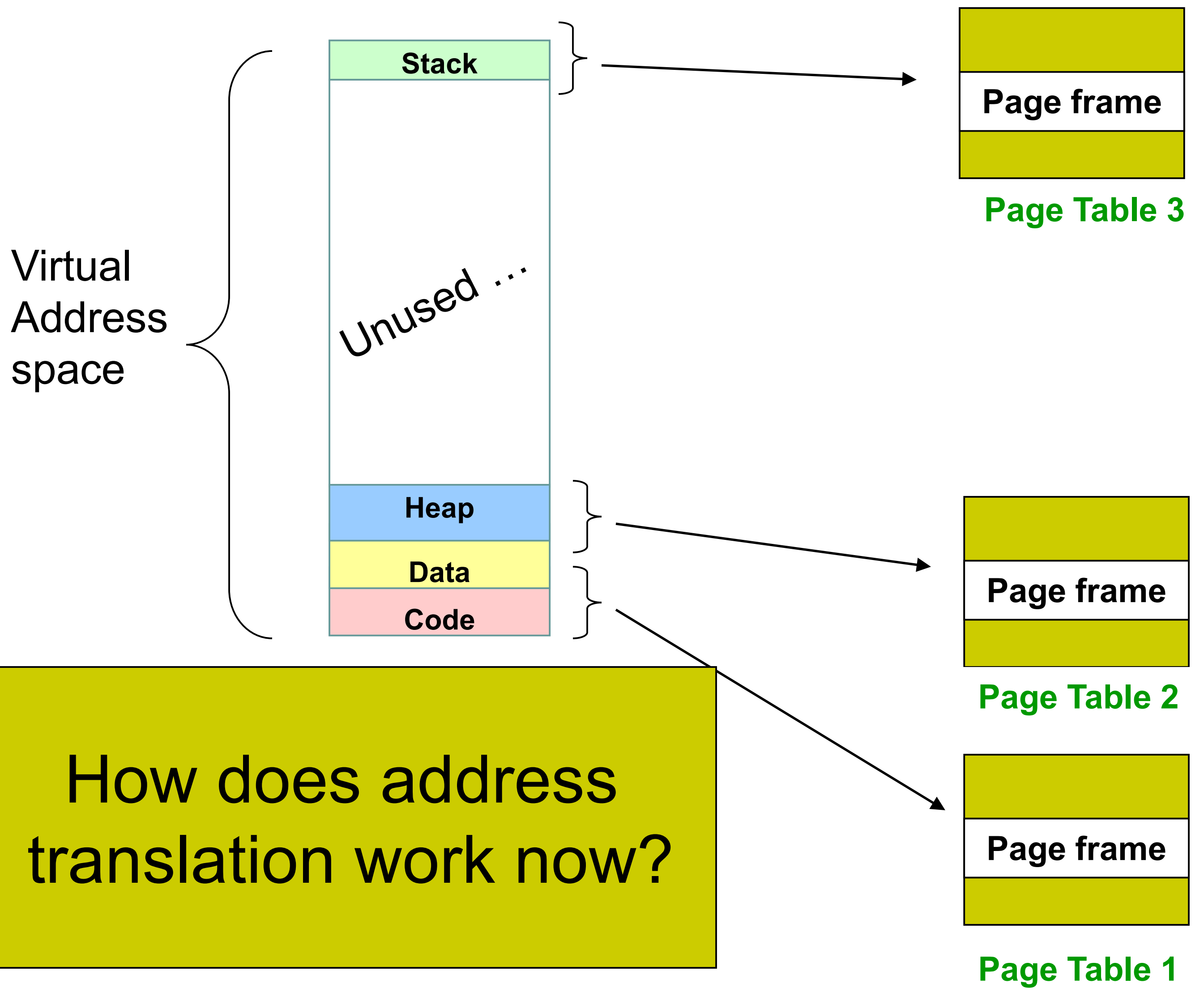
- Need one PTE per page
- 32 bit virtual address space w/ 4K pages
  - = 2^{20} PTEs
- 4 bytes/PTE = 4MB/page table
- 25 processes = 100MB just for page tables!
  - And modern processors have 64-bit address spaces -> 16 petabytes for page table!

- Solutions
  - Hierarchical (multi-level) page tables
  - Hashed page tables
  - Inverted page tables

Managing Page Tables

- How can we reduce space overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)
- How do we only map what is being used?
  - Can dynamically extend page table…
  - Does not work if addr space is sparse (internal fragmentation)
- Use another level of indirection: two-level page tables (or multi-level page tables)

Motivation: two-level page tables

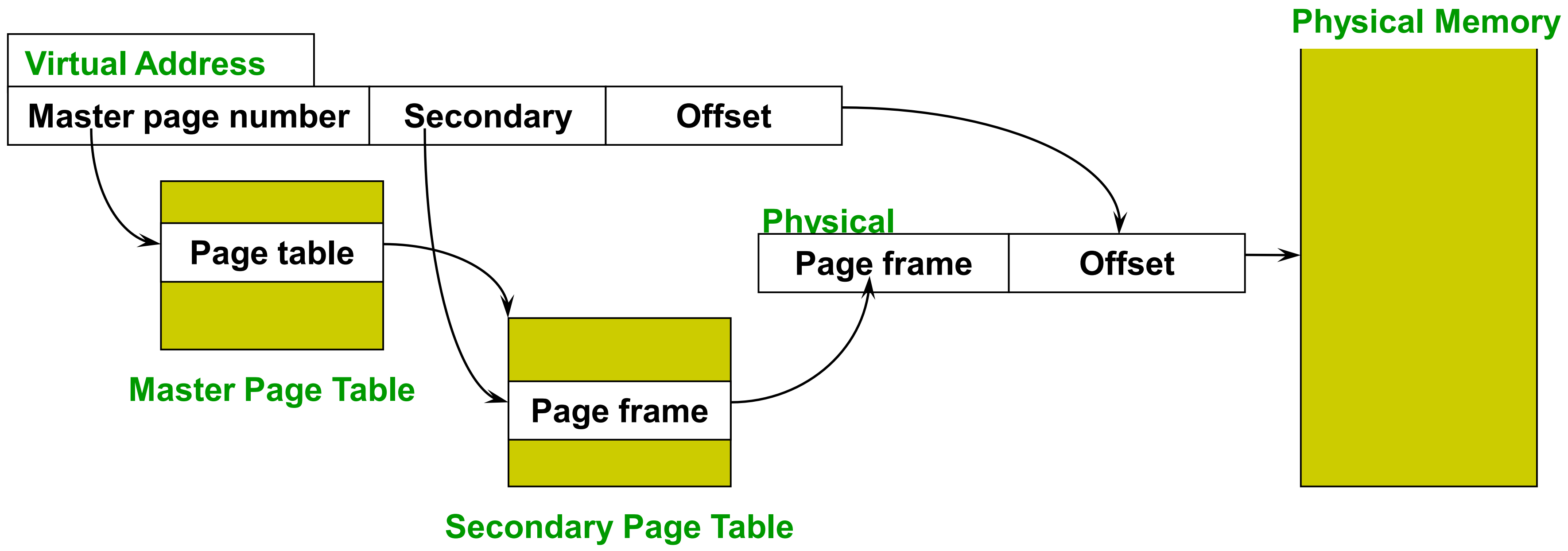


How does address translation work now?

Two-Level Page Tables

Virtual addresses (VAs) have three parts:

- Master page number, secondary page number, and offset
- Master page table maps VAs to secondary page table
- Secondary page table maps page number to physical frame
- Offset selects address within physical frame



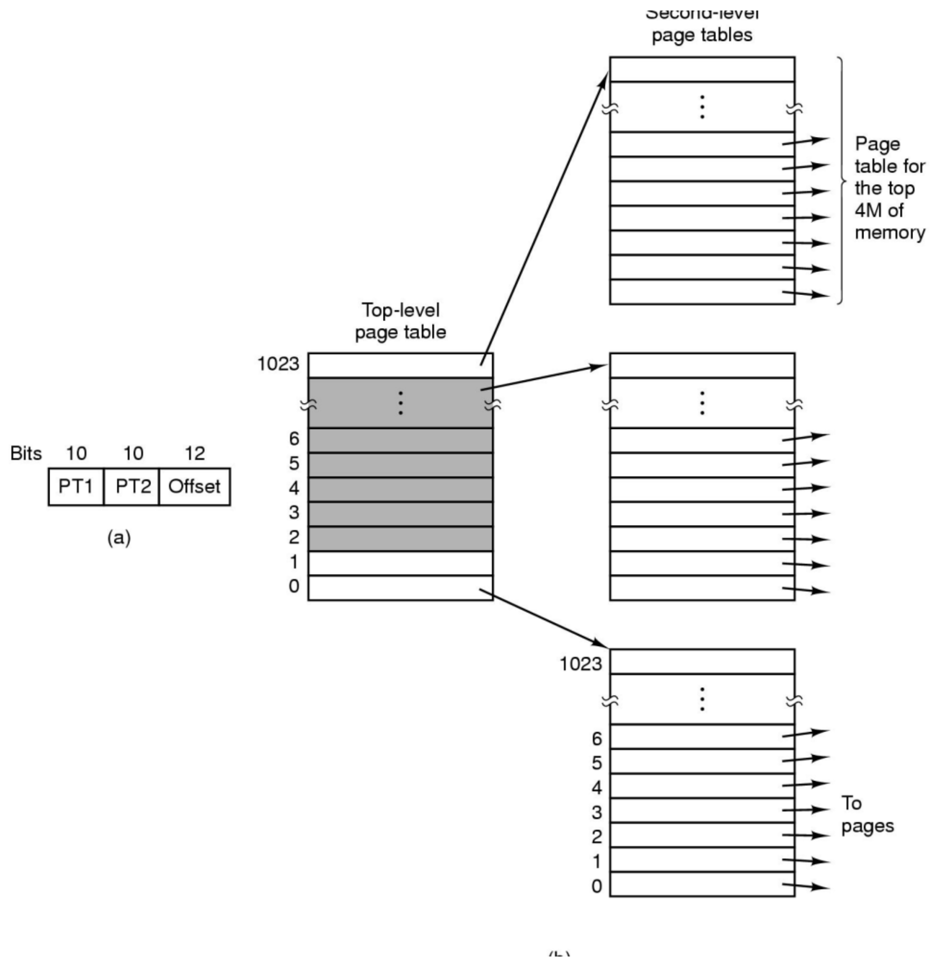
2-Level Paging Example

Virtual Address

Master page number	Secondary	Offset
10 bits	10 bits	12 bits

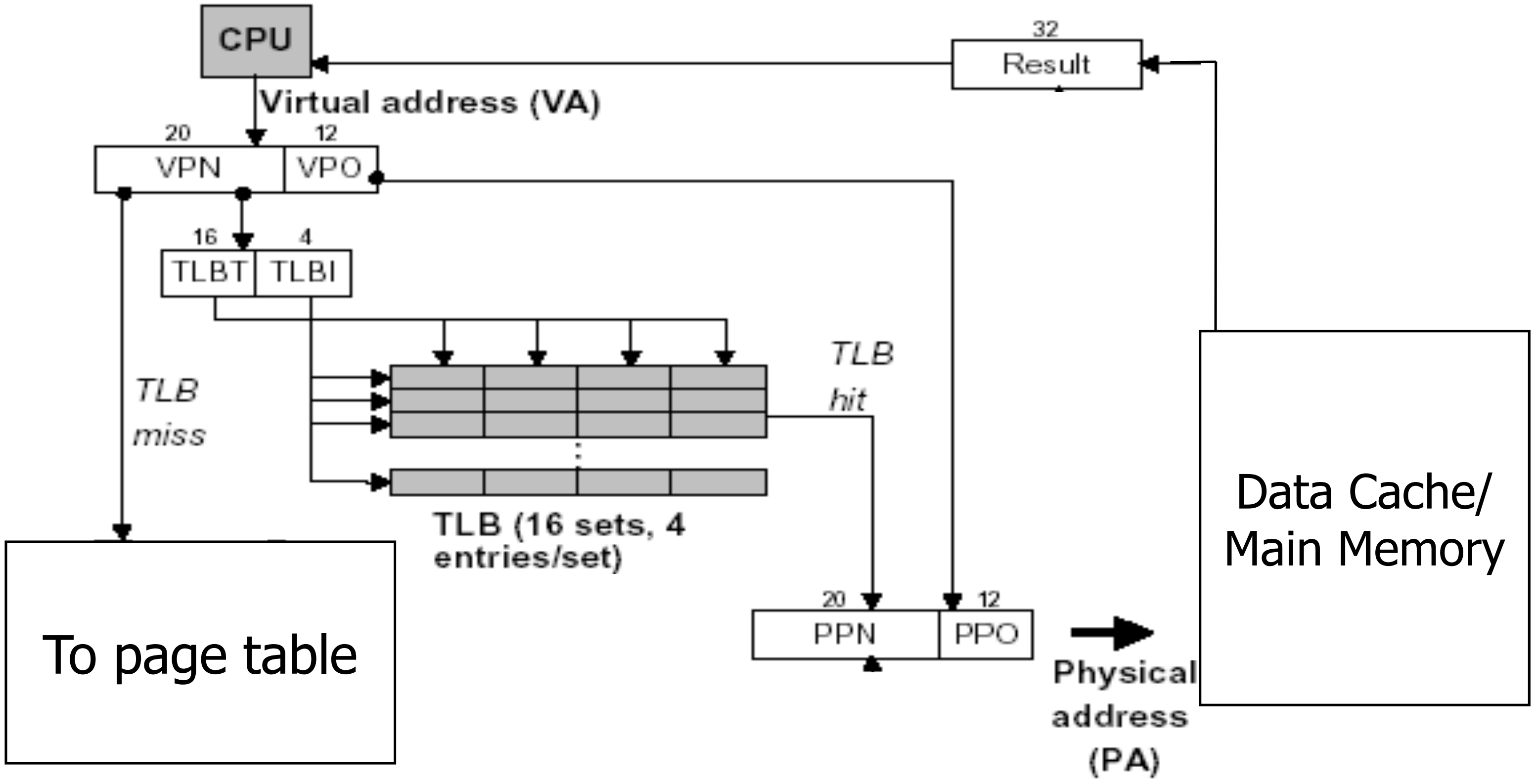
- 32-bit virtual address space
  - 4K pages, 4 bytes/PTE
  - How many bits in offset?
    - 4K = 12 bits, leaves 20 bits
  - Want master/secondary page tables in 1 page each:
    - 4K/4 bytes = 1K entries.
      - How many bits to address 1K entries?
      - 10 bits
    - master = 10 bits
    - offset = 12 bits
    - secondary = 32 – 10 – 12 = 10 bits
  - This is why 4K is common page size!

Multilevel Page Tables

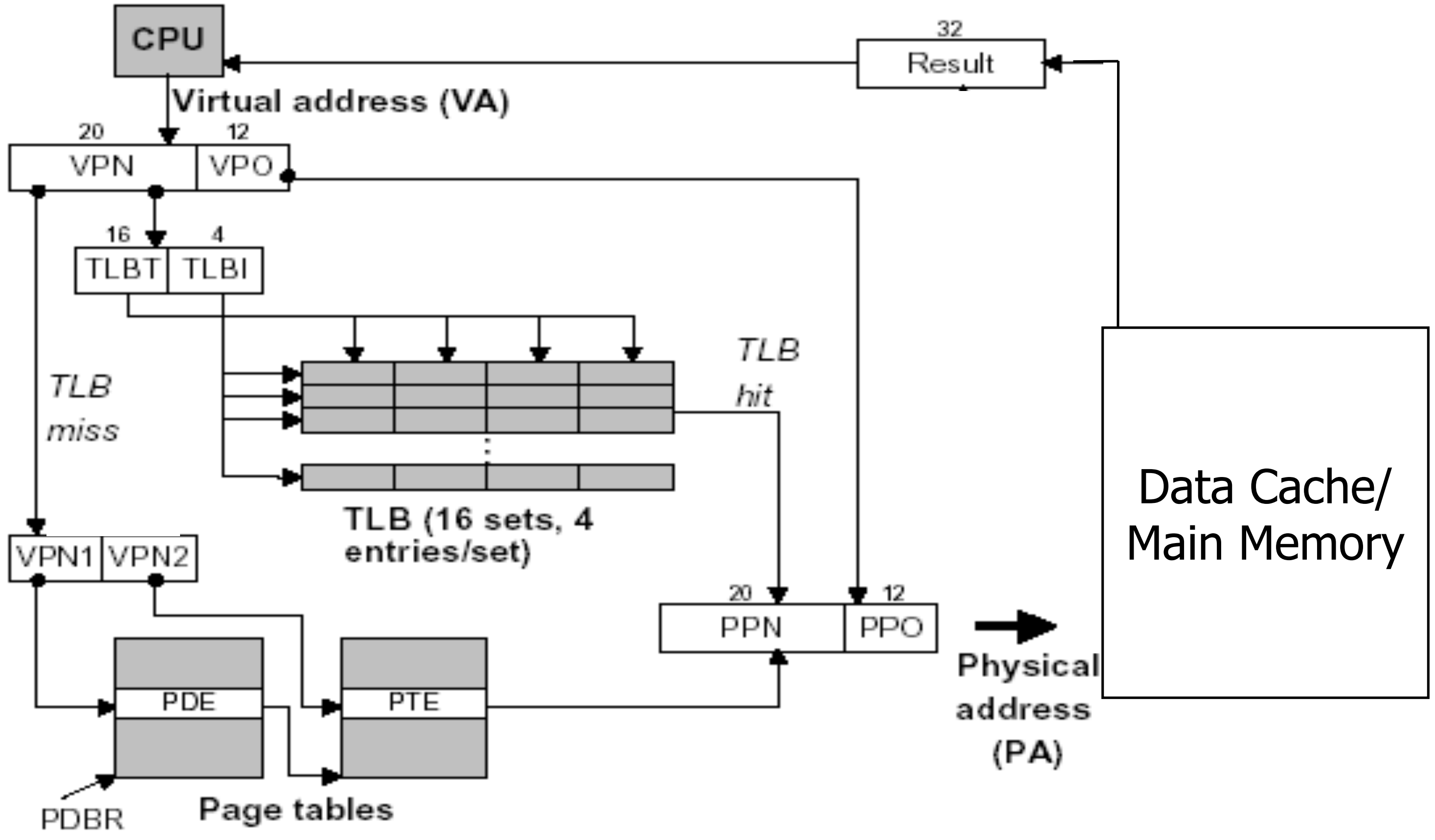


(a) A 32-bit address with two page table fields.
(b) Two-level page tables.

Pentium Address Translation



Pentium Address Translation



64-bit Address Spaces

- Suppose we just extended the hierarchical page tables with more levels
  - 4K pages → 52 bits for page numbers
  - Maximum 1024 entries per level → 6 levels
    - Too much overhead
  - 16K pages → 48 bits for page numbers
  - Maximum 4096 entries per level -> 4 levels
    - Better, but still a lot

Inverted Page Tables

- Keep one table with an entry for each physical page frame
- Entries record which virtual page # is stored in that frame
  - Need to record process id as well
- Less space, but lookups are slower
  - References use virtual addresses, table is indexed by physical addresses
  - Use hashing to reduce the search time

Efficient Translations

- Our original page table scheme already doubled the cost of doing memory lookups
  - One lookup into the page table, another to fetch the data
- Two-level page tables triple the cost!
  - Two lookups into the page tables, a third to fetch the data
  - And this assumes the page table is in memory
- <u>TLB's hide the cost for frequently-used pages</u>