Mechanisms: Segmentation & Paging

2023 Fall COMP3230A

Contents

- Segmentation
  - Address Translation
  - Protection and Sharing of Memory
  - External Fragmentation
  - Strategies in Managing Free Space
    - Best-fit, Worst-fit, First-fit, & Next-fit
- Paging
  - Virtual Page and Page Frame
  - Address Translation using Page Table

Related Learning Outcomes

- ILO 2b - describe the principles and techniques used by OS in effectively virtualizing memory resources.

- ILO 3 [Performance] - analyze and evaluate the algorithms of . . . and explain the major performance issues . . .

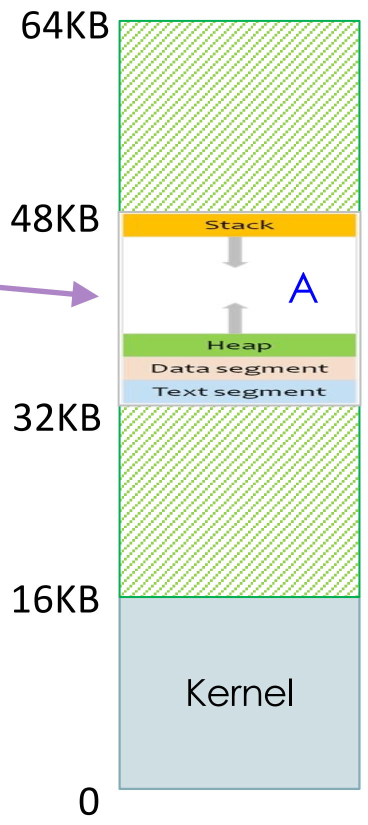
Readings & References

- Required Readings
  - Chapter 16 – Segmentation
    - http://pages.cs.wisc.edu/~remzi/OSTEP/vm-segmentation.pdf
  - Section 17.1 & 17.3 of Chapter 17 Free-Space Management
    - http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf
  - Chapter 18 – Paging: Introduction
    - http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf

REVIEW

A Flaw: Internal Fragmentation

- Example: Process A is in the memory block 32KB to 48KB. A lot of space within this block in physical memory is not used
  - Space between the stack and heap is wasted
- **Fragmentation**
  - The phenomenon wherein **the system cannot use** certain areas of “available” memory
- **Internal** Fragmentation
  - Process is allocated a partition, but it may not take up entire allocated partition
  - Memory **internal** to a partition is not used, but cannot be used by other processes
  - **Waste of memory**



Segmentation

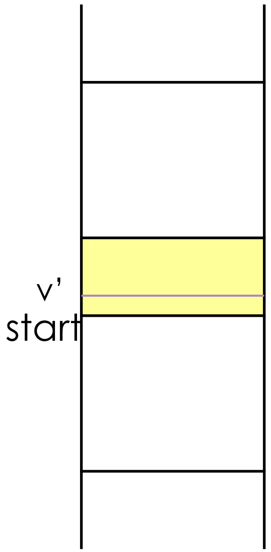
- A segment is a **block of contiguous virtual addresses** of a particular length within a process's address space, which is for a specific purpose/usage
  - e.g., text segment, data segment, heap segment, and stack segment
- OS can place each segment of a process in **different parts** of physical memory
- Only allocate physical memory to memory segments that are in use; thus, **reduce** internal fragmentation → avoid waste
- **Assumption 2** (in Lecture 8 slide # 14) is no longer valid and **Assumption 3** is not relevant anymore

Hardware Support

- To support segmentation, MMU needs to know how to translate **multiple logical segments** within a process's address space
  - **Multiple base-and-bounds register pairs**
  - **Each** base-and-bounds pair for the **address translation of one logical segment**

- Address Translation of a virtual address v'

  offset = virtual address v' – starting virtual address of **the segment**
  physical address = segment_base + offset



- Protection

  if offset > segment_bounds then generate **segmentation fault**

Operational Issue

- Given a virtual address v', which segment is this address in?
  - We need to know **which segment** this virtual address is in so as to find the corresponding base-and-bounds pair

- Each virtual address v' is in one segment only. Can we make use of information within a virtual address to determine which segment it belongs to?
  - A virtual address v' can be **divided** into two parts
  - **Top few bits** are for determining which segment this virtual address v' is in
  - Remaining bits are **the offset**, which points to the memory location **relative to** the segment base address

Address Translation

Assume 16-bit virtual address 41414, the system uses top 4 bits to select the segment

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	1	0	0	0	1	1	0

segment id

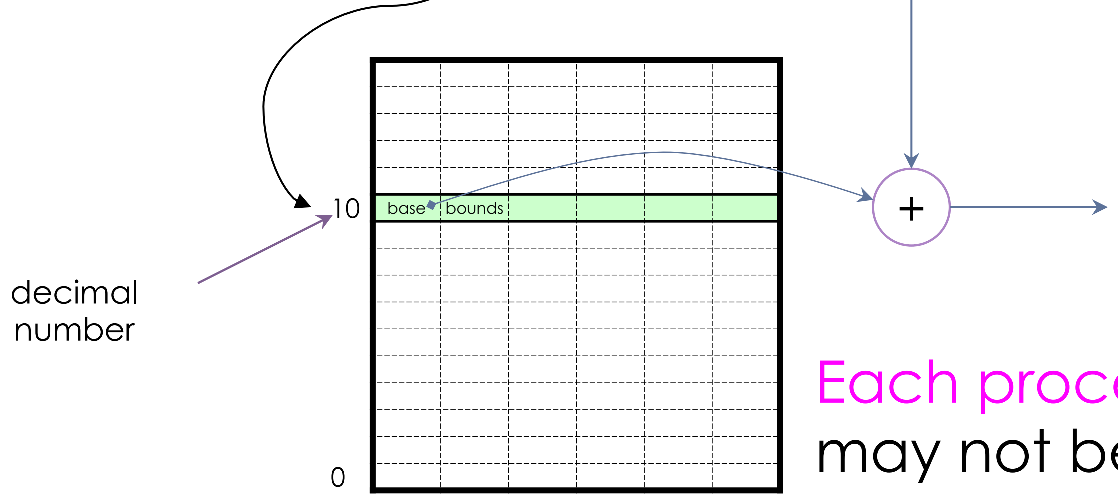
offset

No. of segments

= ??

Max. size of each segment

= ??

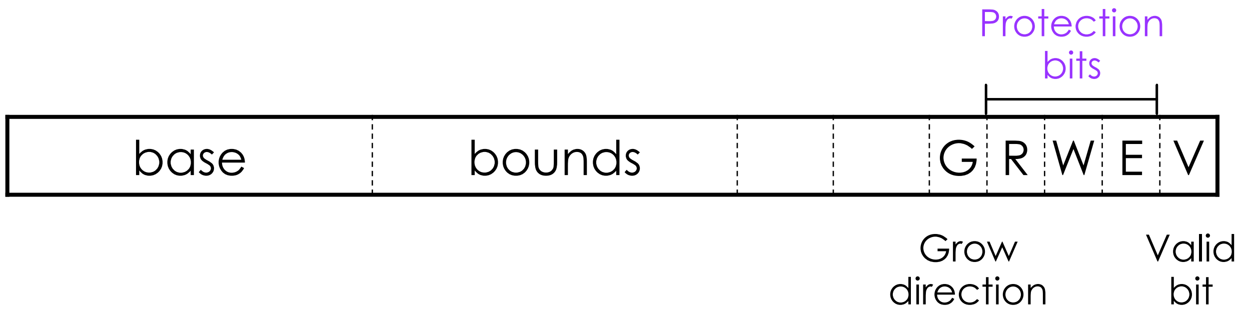


Segment table

physical address = base + offset

Each process has a segment table (may or may not be built in MMU) that contains the address translation information of **all segments** in the process.

Table Entry of Segment Table



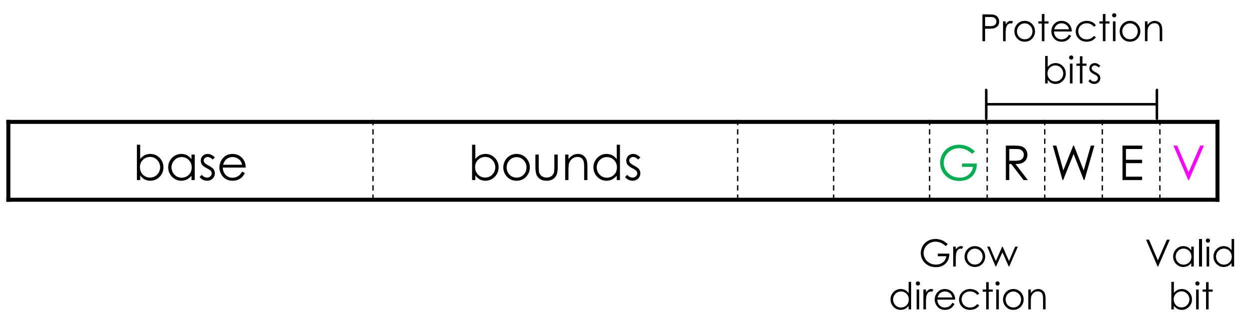
- Protection bits
  - Indicate whether or not a program can **R**ead or **W**rite to a segment, or even **E**xecute code in the segment
  - If process tries to write to a read-only segment, CPU would raise an exception to alert OS to deal with the offense

- Sharing of memory segment between processes
  - e.g., code sharing
  - By appropriate setting of protection bits, OS maps the **same** physical segment into **different** logical segments of **different** process address spaces.

More on Sharing of memory

- Shared code segments, e.g., libraries
- Copy-on-write

Table Entry of Segment Table

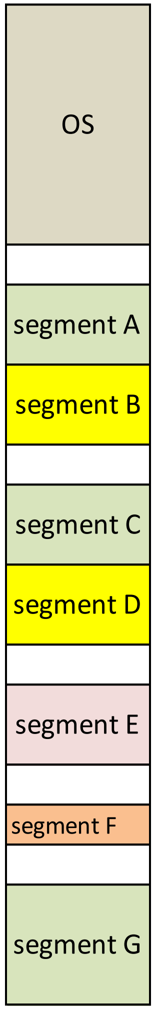


- Grow bit
  - Indicate which direction the segment grows
  - e.g., heap grows upwards, stack grows downwards

- Valid bit
  - When the segment is not in use, valid bit should set to false
  - If a process tries to access such memory, an exception will be generated

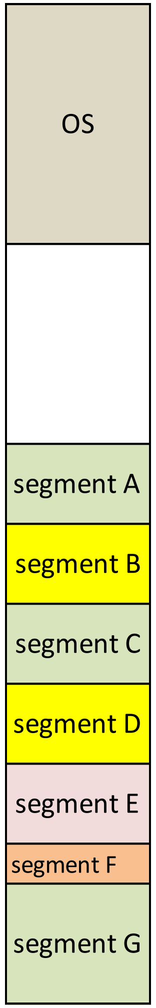
External Fragmentation

- An issue arises from allocating memory to segments of **different sizes**
  - OS needs to find suitable space (block) in physical memory and allocates to new segments or to grow existing segments
  - Processes have multiple segments; when the process terminated, all physical memory used by its segments would be released
  - During normal runtime, many processes arrive, use up the physical memory, terminate and release the physical memory
  - Physical memory quickly becomes **full of little holes of free space**, which are **not large enough to hold a segment**
  - However, the sum of the holes might large enough to accommodate some more segments



External Fragmentation

- Possible ways to combat external fragmentation
  - Coalescing
    - Maintain a list of free blocks
    - Combine adjacent free blocks into one large block
  - Compaction
    - Relocates all occupied areas of memory to one end
      - Forms a single contiguous block free space at the other end
    - Significant overhead
      - Processes have to be suspended during relocation and change their segment register values before resuming them
      - Extensive memory copy involved



Managing Free Space

- OS needs to allocate physical memory to individual segment; thus, OS needs to know where to find free memory
- A straightforward approach is to use **a free-list**
  - Contains references to all of the free chunks of space in physical memory
  - OS just needs to search through the list to locate a suitable free block and allocates to a new request
- The Crux
  - What strategy can be used that is both fast and minimizes external fragmentation?

Allocation Strategy – Best Fit

- Best-fit
  - Search free list and place the segment in the **smallest** possible memory block in which it will fit
    - Example: Freelist →[16KB]→[5KB]→[30KB]→[14KB]

    A new request for placing a segment of size 13KB

    After allocation, the freelist becomes
    Freelist →[16KB]→[5KB]→[30KB]→[1KB]
  - By allocating a "just fit" block, we try to reduce wasted space
  - However, still leave small amount of unused space
    - On long run, it too suffers with problem of numerous unusable small free areas
  - Quite significant performance overhead
    - An **exhaustive search** for the best suit free block

Allocation Strategy – Worst Fit

- Worst-fit
  - Place the segment in the **largest** available memory block in which it will fit
    - Example:

      Freelist → $\boxed{\text{16KB]}}$→[5KB]→[30KB]→[14KB]

      After allocation, the freelist becomes
      Freelist →[16KB]→[5KB]→[17KB]→[14KB]
  - Leaves another "large" hole, making it more likely that another segment can fit in the hole
  - Quite significant performance overhead
    - A full search of the list is required

Allocation Strategy – First Fit

- First-fit
  - Place the segment in the **first memory** block (relative to the list head) in the free list in which it will fit
    - Example:

    Freelist →$\boxed{[16KB]}$→[5KB]→[30KB]→[14KB]

    After allocation, the freelist becomes
    Freelist →[3KB]→[5KB]→[30KB]→[14KB]
  - Simple, **low execution-time overhead**
  - This technique may create many small holes at the beginning of the free list

Allocation Strategy – Next Fit

- Next-fit
  - Similar to first-fit
  - Instead, it searches the free list **at the point where the last search** was stopped at
    - Example: Freelist →[16KB]→[5KB]→[30KB]→[14KB]

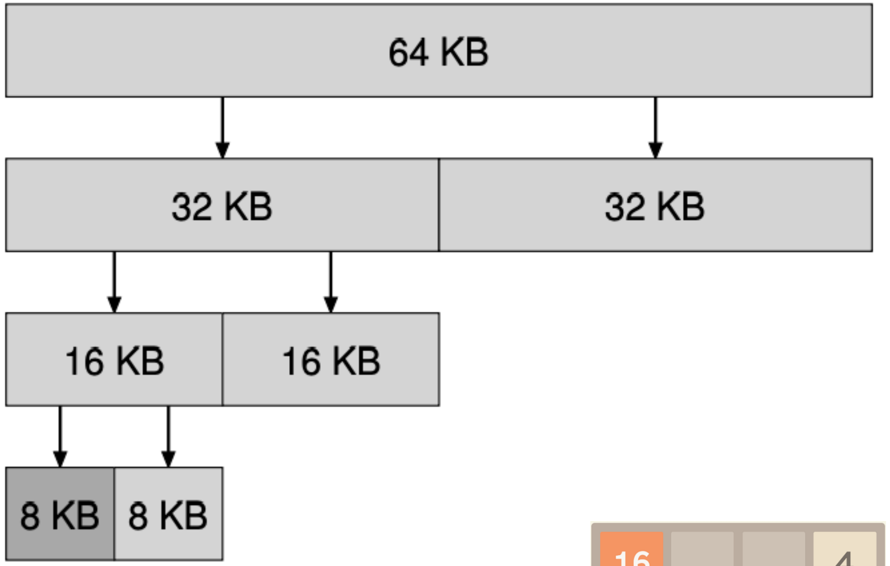
    After allocation, the freelist becomes
    Freelist →[16KB]→[5KB]→[17KB]→[14KB]

- Somewhat surprisingly, best fit results in more wasted memory than first fit and next fit because it tends to fill up memory with tiny, useless holes. First fit generates larger holes on the average.

NOT FOR EXAMINATION

Allocation Strategy – Buddy Allocation

- Binary Buddy Allocator
  - The search for free space recursively divides free space by 2 until a block that is big enough to accommodate the request is found (and a further split into two would result in a space that is too small).
  - When a block is freed, a recursive coalescing process continues up the tree, either restoring the entire free space or stopping when a buddy is found to be in use.



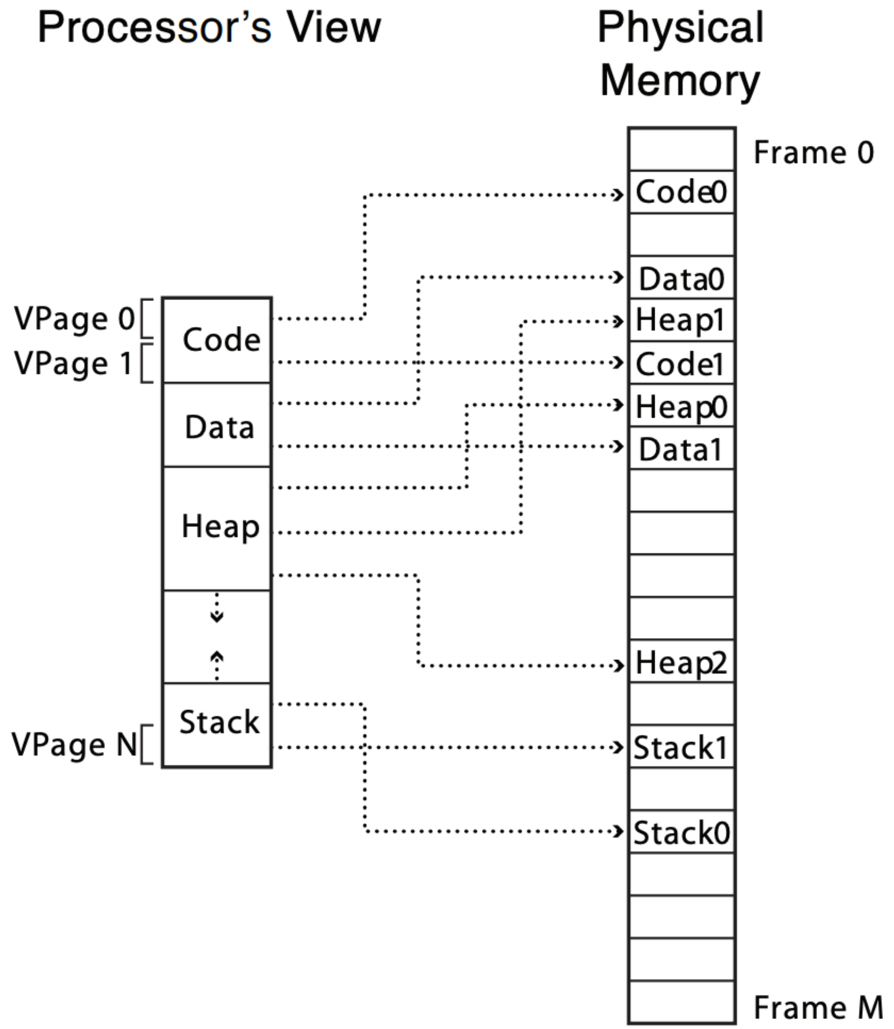
Fixed-Size Blocks

- The only real solution to combat external fragmentation is to avoid the problem altogether, which is by never allocating physical memory in variable-size blocks

- Paging
  - The whole process address space is divided into fixed-size memory block, called (virtual) **page**
  - Physical memory is also divided into memory blocks of the same size of a virtual page – we call it a page frame or simply a **frame**

Paging

- Advantages of Paging
  - Simple, because all pages/frames are of the same size
    - Completely avoid external fragmentation as each block is of the same size
      - Any virtual page of any process can be placed in any free page frame
    - This also simplifies the free-space management



Address Translation – Page Table

- Also known as page map table
- **Each process** has its page table
  - Contains the address translation information of **every virtual page** of a process
- Let's have the simple calculation
  - With 32-bit x86 processor, a process address space is of size 2^{32} bytes = 4294967296 bytes = 4 **GiB**
  - The default size of a virtual page (as well as page frame) is 2^{12} bytes = 4096 bytes = 4 **KiB**
  - How many virtual pages are there in one 32-bit address space?
    - $2^{32}/2^{12} = 2^{20}$ virtual pages
  - Thus, **a page table** needs to store 2^{20} = 1048576 entries of address translation information for a 32-bit process

Address Translation

- Given a virtual address ***v***, how to look up the translation information in the process's page table?
  - We need to **find out which virtual page** this virtual address is in

- Address Translation
  - Given a **virtual address *v***, **split** it into two parts
  - **The top left *p* bits** , we call it the *virtual page number*, are for determining which virtual page this virtual address is in
  - Remaining bits on the right are **the offset *d***, which points to the memory location ***relative to the starting address*** of a page/frame
  - How many virtual pages are there? And what is the size of a page?
    - No. of virtual pages = 2^p Size of a page = 2^d bytes

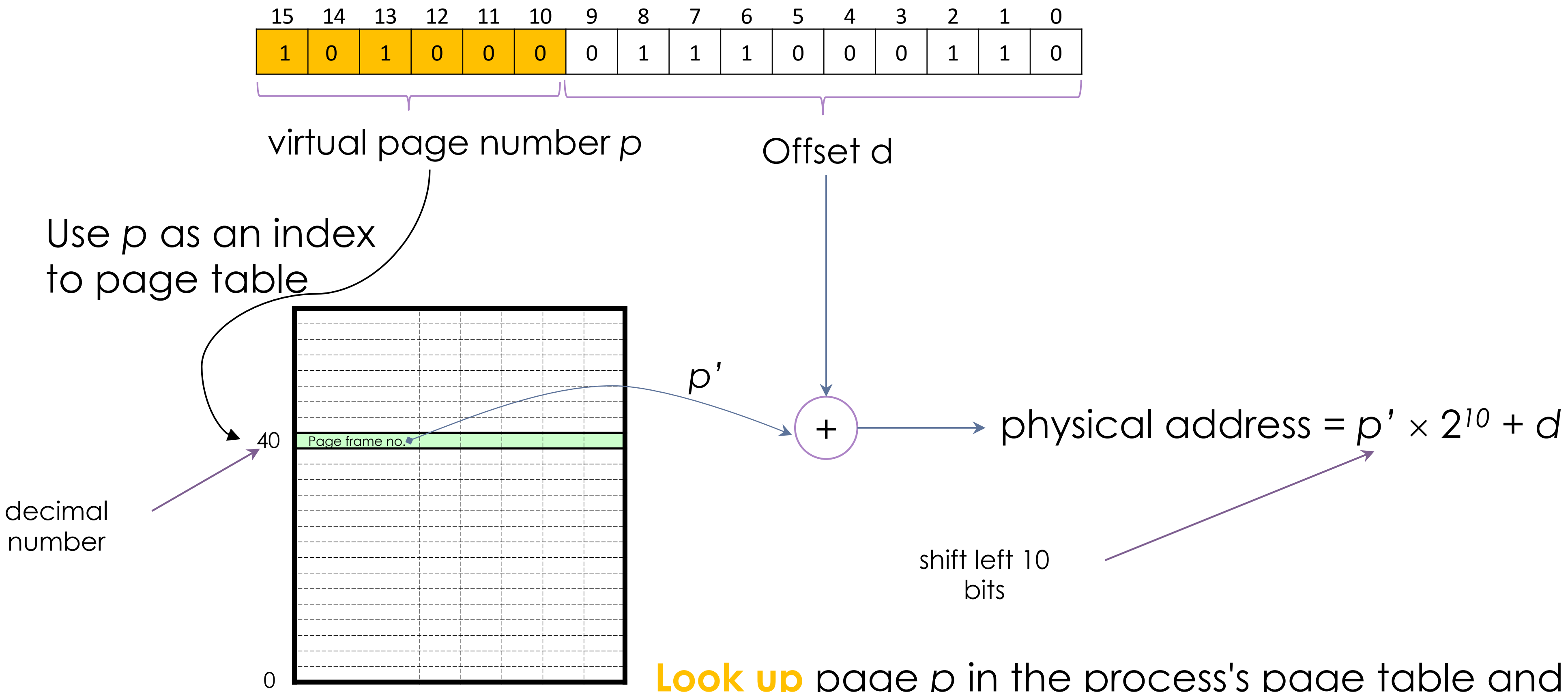
Address Translation

- Assume 16-bit addressing and the page size is 1024 bytes = 2^{10}
  - No. of virtual pages = $\boxed{??}$
- Given a virtual address 41414, we can determine its virtual page number *p* and the offset *d* in this virtual page

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	1	1	1	0	0	0	1	1	0

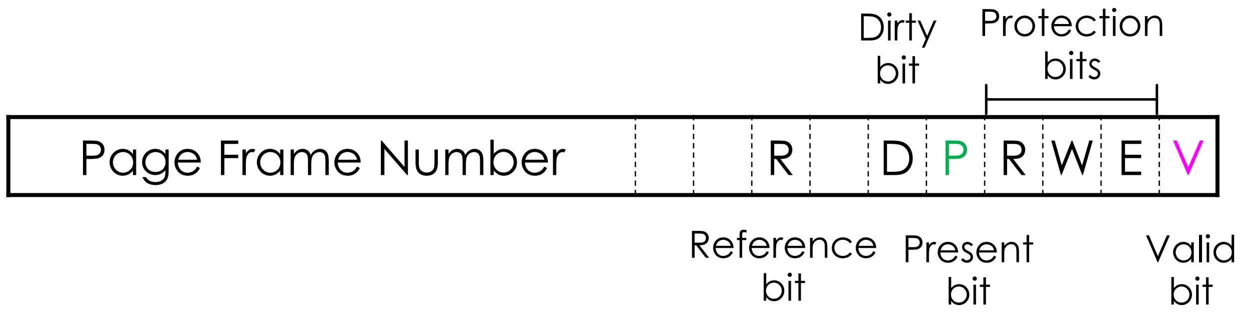
virtual page number *p* (bits 15–10) Offset d (bits 9–0)

Address Translation



Look up page *p* in the process's page table and find that page *p* is in page frame number *p*'

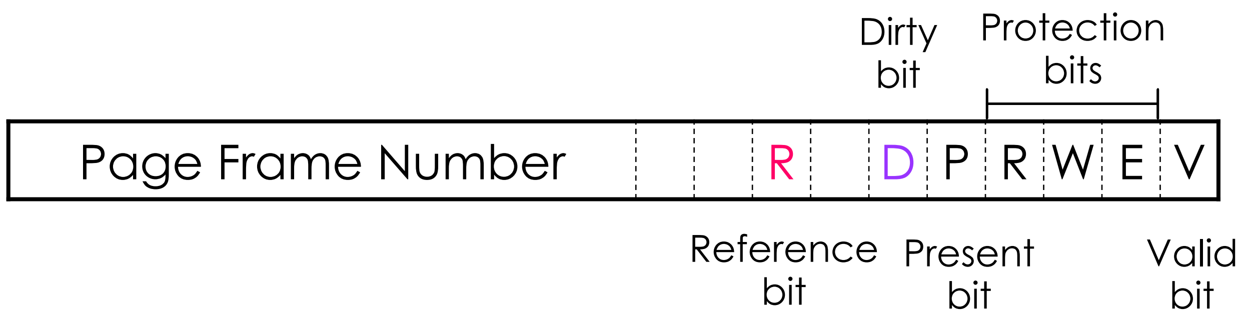
Page Table Entry (PTE)



- Valid bit
  - Indicates whether this PTE contains valid translation information
    - e.g., if this virtual page is unused, valid bit is set to false
  - Accessing an invalid page will generate an exception

- Present bit
  - Indicates whether this virtual page is in physical memory or on disk
  - If present bit is false, the value stored in PFN may not be correct or may not refer to a valid page frame

Page Table Entry (PTE)



- Dirty bit
  - Indicates whether this virtual page has been modified since it was brought into memory
- Reference (or accessed or use) bit
  - Indicates whether this virtual page has been recently accessed by the process
  - As a useful indicator for the **page replacement strategies**

Where are page tables stored?

- Page table is quite large
  - The no. of entries = no. of virtual pages in the process address space
  - Need to store contiguously in memory
    - 32-bit with 4KiB page size
    - $2^{32}/2^{12} = 2^{20}$ PTEs
    - assume each PTE occupies 4 bytes, we need **4 MiB** for a page table
- Each process has its own page table
  - There are many processes running in the system !!!
- Must keep in physical memory – relatively slow speed of access

Where are page tables stored?

- MMU needs to know where to find the page table for current running process

- CPU uses a register called **Page-table base register** to point to the **starting physical address** of the page table
  - OS must **reload this register during context switch** for the next process

- To translate each virtual address
  - MMU needs to access page table (via page-table base register) which is in the **physical memory**
  - Reads the PTE and performs the translation
  - Fetches the desired data from the **physical memory**

  - In summary, **we need to access physical memory twice** for each memory reference

The Crux

- How to make paging faster and page table smaller?
  - Each memory access results in two physical memory accesses
  - Page table must be stored contiguously in physical memory

Summary

- Segmentation allows better utilization of physical memory as only segments in use are allocated with physical memory
- To support segmentation, MMU has to include more base-and-bounds register pairs (or even a segment table) for the address translation and protection
- Allocating variable-sized segments in physical memory leads to external fragmentation, which results in a waste of memory
- No matter how smart the allocation policies, external fragmentation still exists.

Summary (2)

- By dividing the memory into fixed-size memory blocks, this avoids external fragmentation and simplifies the free-space management

- To translate a virtual address, MMU
  - Extracts the virtual page number from the virtual address
  - Accesses process's page table in physical memory to get the physical page frame number
  - Concatenates PFN with offset to form the physical address
  - Fetches the data from the physical memory

Operating Systems

- **Virtualization**
  - CPU Virtualization
    - Process Abstract
      - Address space
      - Process states
      - Process control block
      - Process operations API
      - Signals
    - Limited Direct Execution
      - System calls
      - Context switch
      - Interrupts
    - Scheduling
      - Scheduling metrics
      - FIFO, SJF, HRRN, STCF, RR, MLFQ
      - Multi-core scheduling, Linux CFS
  - Memory Virtualization
    - Address space
    - Address translation: dynamic relocation
    - Segmentation
    - Paging
    - TLB
    - Multi-level paging
    - Inverted page table
    - Swap space
    - Page replacement policy: FIFO, LFR, LRU, Clock
    - Thrashing
- **Concurrency**
  - Thread
    - POSIX threads (pthreads)
    - Race conditions, critical sections, mutual exclusion, atomic operations, synchronization
  - Locks
    - Atomic instructions: test-and-set, compare-and-swap
    - Mutex locks
  - Condition Variables
    - Pthread CVs
    - Producer-Consumer problem
  - Semaphores
    - Binary Semaphores
    - Counting Semaphores
    - Ordering
    - Readers-Writers problem
  - Deadlock
    - Dining philosophers' problem
    - Four necessary conditions
    - Deadlock prevention, avoidance, detection&recovery
- **Persistence**
  - I/O devices (HDD, SSD)
  - Files and Directories
    - Inode
    - File descriptor
    - Hard/Symbolic links
  - File System Implementation
    - On-disk data structure
      - Superblock, Bitmap, Inodes, Data blocks
    - Free space management
      - Bitmap, linked-list, block-list
    - Caching and buffering
    - Access control and protection
    - Journaling file system
      - Data journaling
      - Metadata journaling
- ***Advanced Topics***