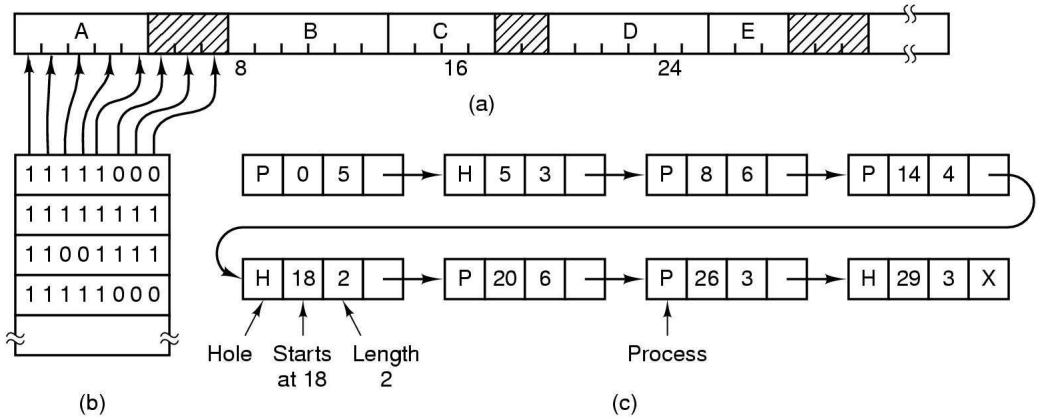

Figure 1: (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded region (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

0.0.1 Memory Management cont.

- Memory allocation and freeing operations are partially predictable. Since the organization is hierarchical, the freeing operates in reverse (opposite) order (Stack Organization)
- Allocation and release of heap space is totally random. Heaps are used for allocation of arbitrary list structures and complex data organizations. As programs execute (and allocate and free structures), heap space will fill with holes (unallocated space, i.e., fragmentation) (Heap Organization)
- How do we know *when* memory can be freed? It is trivial when a memory block is used by one process
- However, this task becomes difficult when a block is shared (e.g., accessed through pointers) by several processes
- Two problems with reclaiming memory:
  - **Dangling pointers**: occur when the original allocator *frees* a shared pointer.
  - **Memory leaks**: occur when we forget to free storage, even when it will not or cannot be used again. This is a common and serious

problem. Unacceptable in an OS.

- Memory Management with Bitmaps (see Fig. 1)
  This technique divides memory into fixed-size blocks (e.g., sectors of 256-byte blocks) and keeps an array of bits (bit map), one bit for each block

- Memory Management with Linked Lists (see Fig. 1)
  A *free list* keeps track of the unused memory. There are several algorithms that can be used, depending on the way the unused memory blocks are allocated (Dynamic Partitioning Placement Algorithm):

  - **First-fit**: Allocate *first* hole that is big enough
    - Scan the list for the first entry that fits
    - If greater in size, break it into an allocated and free part
    - May have many processes loaded in the front end of memory that must be searched over when trying to find a free block
    - May have lots of unusable holes at the beginning.
    - External fragmentation
  - **Next-fit**
    - Like first-fit, except it begins its search from the point in list where the last request succeeded instead of at the beginning.
    - More often allocates a block of memory at the end of memory where the largest block is found
    - The largest block of memory is broken up into smaller blocks
    - Compaction is required to obtain a large block at the end of memory
    - Simulations show it is slightly slower
  - **Best-fit**: Allocate *smallest* hole that is big enough;
    - Chooses the block that is closest in size to the request
    - Poor performer
    - Has to search complete list, unless ordered by size.
    - Since smallest block is chosen for a process, the smallest amount of fragmentation is left memory compaction must be done more often
  - **Worst-fit**: Allocate *largest* hole;
    - Chooses the block that is largest in size (worst-fit)

    * Idea is to leave a usable fragment left over
    * Poor performer
    * must also search entire list to find largest leftover hole (keep list in size order)
    * Simulations show it is not a good idea

0.1 Virtual Memory

- Differentiation of user logical memory from physical memory
  - only part of program needs to be in memory for execution
  - logical address space can be >> than physical address space
  - allows address spaces to be shared by processes
  - allows more efficient process creation
- Virtual memory (VM) can be implemented via:
  - demand paging
  - demand segmentation

0.1.1 Paging (see the Fig. 2)

- A VM Larger Than Physical Memory
- Logical address space of a process can be noncontiguous; process allocated physical memory wherever latter available
- Bring page into memory only when needed
  - less I/O needed
  - less memory needed
  - faster response
  - more processes
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - Not in memory ⇒ bring to memory
- Reference may result from:

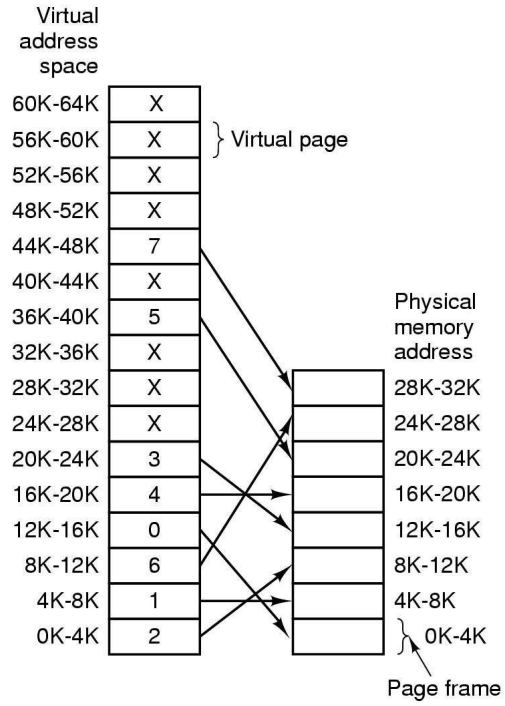

Figure 2: The relation between virtual address and physical memory addresses is given by a page table.

  - instruction fetch
  - data reference

- Divide physical memory into fixed-sized blocks called **frames** (size power of 2, typically 512 bytes - 8KB)
- Divide logical memory into blocks of same size called **pages**
- A **Present/Absent** bit keeps track of which pages are physically present in memory
- A page table defines (maps) the base address of pages for each frame in the main memory
- The major goals of paging are to make memory allocation and swapping easier and to reduce fragmentation
- Paging also allows allocation of non-contiguous memory (i.e., pages need not be adjacent.)

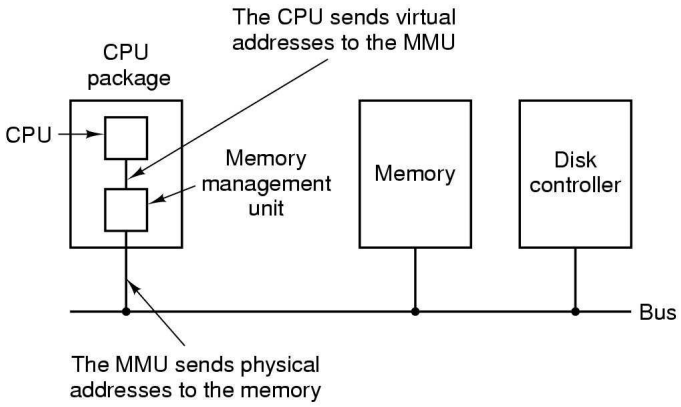

Figure 3: The position and function of the MMU.

- Keep track of all free frames
- Set up page table to translate logical to physical addresses
- Internal fragmentation, sometimes less than full page needed
- Dynamic relocation, each program-generated address (logical address) is translated to hardware address (physical address) at runtime for every reference, by a hardware device known as the memory management unit (MMU)
- Memory-Management Unit
  - Hardware maps logical to physical address
  - With MMU, value in relocation register added to every address generated by user process when sent to memory
  - User program deals with *logical* addresses; never sees *real* physical addresses
- Address Translation Scheme; address generated by CPU divided into:
  - *Page number (p)* used to index into *page table* with base address of each page in physical memory
  - *Page offset (d)* combined with base address defines physical address for memory system

- What happens when an executing program references an address that is not in main memory? (see Fig. 4)
- The page table is extended with an extra bit, present/absent bit
- **Page Fault:** Access to a page whose present bit is not set causes a special hardware trap, called page fault
- Initially, all the present bits are cleared. While doing the address translation, the MMU checks to see if this bit is set.
    1. Trap to OS
    2. Save user registers and process state
    3. Determine that interrupt was page fault
    4. Check that page reference was legal, determine location of page on disk
    5. Issue read from disk to free (physical) frame:
        - Wait in queue for device to service read request
        - Wait for device seek / rotational latency
        - Wait for transfer
    6. While waiting for disk: allocate CPU to another process
    7. Interrupt from disk
    8. Save registers and process state for other process
    9. Determine that interrupt was from disk
    10. Update page / OS tables to show page is in memory
    11. Wait for CPU to be allocated to this process
    12. Restore registers, process state, page table
    13. Restart instruction
- When a page fault occurs the operating system brings the page into memory, sets the corresponding present bit, and restarts the execution of the instruction
- What happens if no free frame?
    - Page replacement find some page in memory, ideally not in use, swap it out
        * algorithm performance

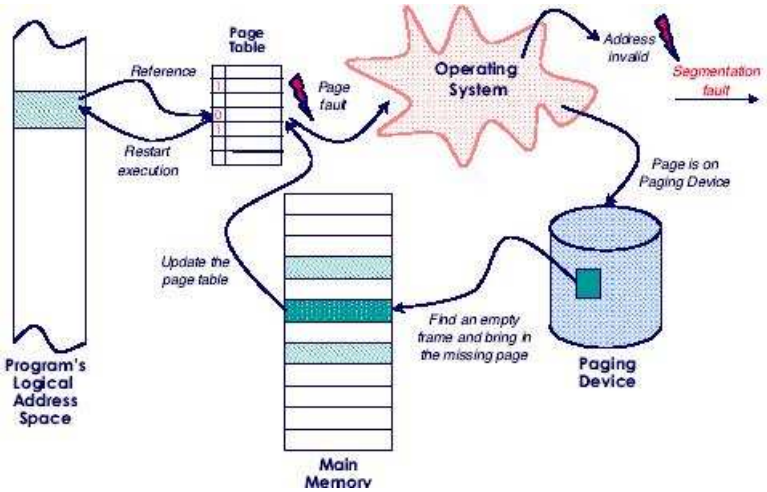

Figure 4: Page fault handling by picture.

  - ∗ want to minimize number of page faults
  - Same page may be brought into memory several times

- Problem: Both paging and segmentation (we will discuss later) schemes introduce extra memory references to access translation tables.
- Solution? Translation buffers (like caches)
- Based on the notion of **locality** (at a given time a process is only using a few pages or segments), a very fast but small associative (content addressable) memory is used to store a few of the translation table entries
- This memory is known as a translation look-aside buffer or TLB
- Similar to storing memory addresses in TLBs, frequently used data in main memory can also be stored in fast buffers, called cache memory, or simply cache (see Fig. 5)
- Basically, memory access occurs as follows:

```
for each memory reference
 if data is not in cache <miss>
  if cache is full
   remove some data
  if read access
```

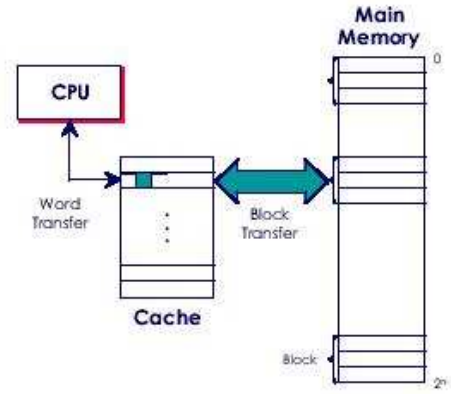

Figure 5: Memory Caching.

```
  issue memory read
  place data in cache
  return data
else <hit>
 if read access
  return data
 else
  store data in cache
```

- The idea is to make frequent memory accesses faster!
- Cache terminology;
  - **Cache hit**: item is in the cache.
  - **Cache miss**: item is not in the cache; must do a full operation. **Categories of cache miss**:
    - *Compulsory*: the first reference will always miss.
    - *Capacity*: non-compulsory misses because of limited cache size.
  - **Effective access time**: P(hit) * cost of hit + P(miss)* cost of miss, where P(hit) = 1 - P(miss)

0.1.2 Page Tables (see Fig. 6)

- Page table kept in main memory

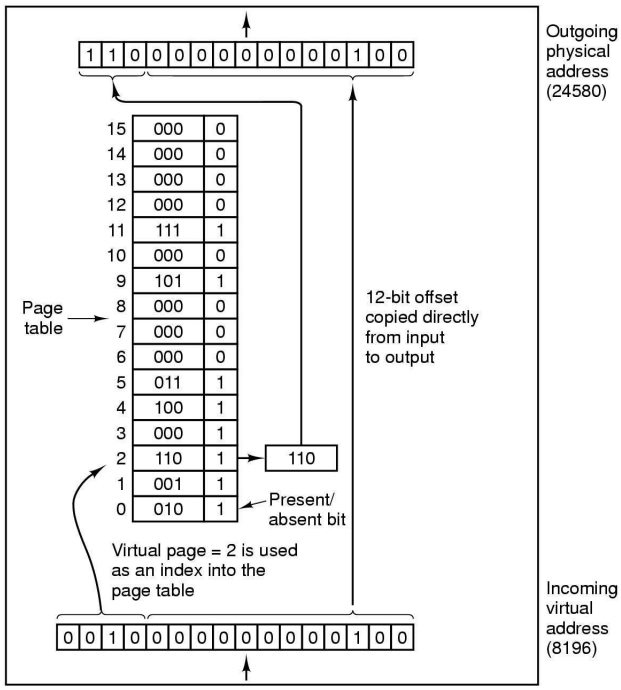

Figure 6: The internal operation of the MMU with 16 4-KB pages.

- The Modified and Referenced bits keep track of the page usage. If the page in it has been modified, it must be written back to the disk. This bit s sometimes called the **dirty bit.**
- *Page-table base register* (PTBR) points to page table
- *Page-table length register* (PRLR) has size of page table
- In this scheme every data/instruction access needs 2 memory accesses: page table and data/instruction
- 2-memory access problem can be solved by special fast-lookup hardware cache using *associative memory* called *translation look-aside buffer (TLB)*
- Address space may be very large, e.g.:
  - 32-bit addresses 4GB
  - 64-bit addresses millions of TB
- May have big gaps (*sparse*), e.g.

  - stack grows down from high memory
  - dynamic allocation grows up from low memory

- Page table very large, big waste of memory

0.1.3 Inverted Page Tables

- The inverted page table has one entry for each memory frame.
- Entry is virtual address of page stored in real location, with information about owning process
- Decreases memory to store page table, increases time to search for page translation
- Hashing is used to speedup table search. Hash table limits search to O(1) entries
- Popular with virtual space >> physical
- Hard to handle *aliases* (> 1 virtual page maps to 1 physical page)
- The inverted page table can either be per process or system-wide. In the latter case, a new entry, PID (process id)is added to the table.
- Adv: independent of size of address space; small table(s) if we have large logical address spaces.

0.1.4 Basic policies

- The hardware only provides the basic capabilities for virtual memory. The operating system, on the other hand, must make several decisions:
  - *Allocation*–how much real memory to allocate to each (*ready)* program?
    - In general, the allocation policy deals with conflicting requirements:
      - The fewer the frames allocated for a program, the higher the page fault rate
      - The fewer the frames allocated for a program, the more programs can reside in memory; thus, decreasing the need of swapping.

        - Allocating additional frames to a program beyond a certain number results in little or only moderate gain in performance.
    - The number of allocated pages (also known as *resident set size*) can be *fixed* or can be *variable* during the execution of a program.
- *Fetching*–when to bring the pages into main memory?
    - **Demand paging**; Start a program with no pages loaded; wait until it references a page; then load the page (this is the most common approach used in paging systems.)
    - **Request paging**; Similar to overlays, let the user identify which pages are needed (not practical, leads to over estimation and also user may not know what to ask for.)
    - **Pre-paging**; Start with one or a few pages preloaded. As pages are referenced, bring in other (not yet referenced) pages too.
    - Opposite to fetching, the *cleaning policy* deals with determining when a modified (dirty) page should be written back to the paging device.
- *Placement*–where in the memory the fetched page should be loaded?
    - This policy usually follows the rules about paging and segmentation.
    - Given the matching sizes of a page and a frame, placement with paging is straightforward.
    - Segmentation requires more careful placement, especially when *not* combined with paging. Placement in pure segmentation is an important issue and *must* consider free memory management policies.
    - With the recent developments in *non-uniform memory access (NUMA)* distributed memory multiprocessor systems, placement does become a major concern.
- *Replacement* (see the Fig. 7) –what page should be removed from main memory?
    - The most studied area of the memory management is the replacement policy or victim selection to satisfy a page fault.

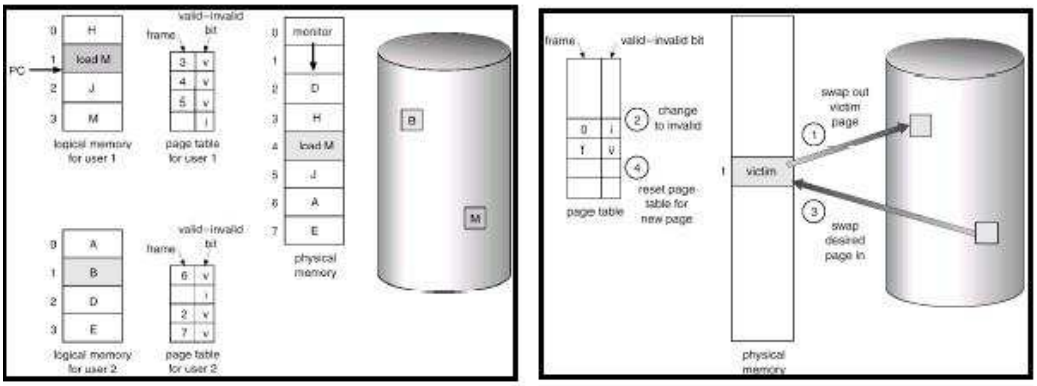

Figure 7: Page Replacement.

0.1.5 Page Replacement Algorithms

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use *dirty (modify)* bit to reduce overhead of page transfers; only modified pages written back to disk
- Page replacement completes separation between logical memory and physical memory; large virtual memory can be provided on smaller physical memory
- Find location of desired page on disk
- Find free frame:
  - if free frame, use it.
  - if no free frame, page replacement algorithm selects *victim* frame
  - If victim *dirty* write back to disk
- Read desired page into (newly) free frame; update page and frame tables
- Restart faulting process
- Want lowest page-fault rate
- Evaluate algorithm by running on given string of memory references (reference string) and compute number of page faults

Table 1: Page replacement algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU	Very crude
FIFO	Might throw out important pages
Second Chance	Big improvement over FIFO
Clock	Realistic
LRU	Excellent, but difficult to implement exactly
NFU	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

- The Optimal Page Replacement Algorithm; the page that will not be referenced again for the longest time is replaced (prediction of the future; purely theoretical, but useful for comparison.)
- The Not Recently Used Page Replacement Algorithm; algorithm removes a page at random
- The First-In, First-Out (FIFO) Page Replacement Algorithm; FIFO the frames are treated as a circular list; the oldest (longest resident) page is replaced
- The Second Chance Page Replacement Algorithm; look for an old page that has not been referenced in the previous clock interval, avoids the problem of throwing out of heavily used page
- The Least Recently Used (LRU) Page Replacement Algorithm; LRU the frame whose contents have not been used for the longest time is replaced
- Summary of Page Replacement Algorithms

0.1.6 Page Replacement Cont.

- Global vs. Local Allocation
  - **Global** replacement; process selects replacement frame from set of all frames; process can take frame from another
  - **Local** replacement each process selects only from its own set of allocated frames

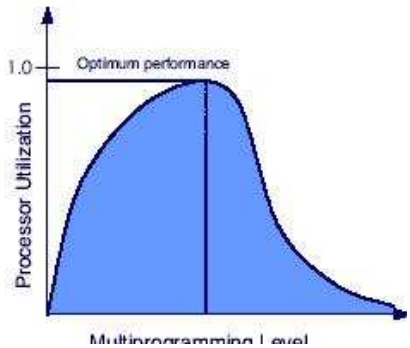

Figure 8: Trashing.

- **Frame locking**; frames that belong to resident kernel, or are used for critical purposes, may be locked for improved performance.
- **Page buffering**; victim frames are grouped into two categories: those that hold unmodified (clean) pages and modified (dirty) pages
- **Thrashing (see Fig. 8)**;
  - The number of processes that are in the memory determines the multiprogramming (MP) level.
  - The effectiveness of virtual memory management is closely related to the MP level.
  - When there are just a few processes in memory, the possibility of processes being blocked and thus swapped out is higher.
  - When there are far too many processes (i.e., memory is overcommitted), the resident set of each process is smaller.
  - This leads to higher page fault frequency, causing the system to exhibit a behavior known as thrashing.
  - In other words, the system is spending its time moving pages in and out of memory and hardly doing anything useful.
  - process spends more time paging than executing
  - The only way to eliminate thrashing is to reduce the multiprogramming level by suspending one or more process(es).
  - Victim process(es) can be the: lowest priority process, faulting process, newest process, process with the smallest resident set, process with the largest resident set

– Student analogy to thrashing: Too many courses!