H2

IX. Virtual Memory

- Modern systems provide an abstraction of main memory known as **virtual memory**
  - to manage memory more efficiently and with fewer errors
  - VM provides...
    1. uses main memory efficiently by treating it as a cache for an address space stored on disk, keeping only the active areas in main memory and transferring data back and forth between disk and memory as needed
    2. simplifies memory management by providing each process with a uniform address space
    3. protects the address space of each process from corruption by other processes

- Why would we need to understand VM?
  1. VM is central
     - pervades all levels of computer systems
  2. VM is powerful
     - can create / destroy chunks of memory, map chunks of memroy to portions of disk files, and share memory with other processes
  3. VM is dangerous
     - if used improperly, applications can suffer from perplexing and insidious memory-related bugs

- This chapter focuses on...
  1. How VM works
  2. How to use and manage VM in programs

H3

§9.1 Physical and Virtual Addressing

- Main memory of a computer system is organized as an array of M contiguous byte-size cells
  - each has a unique **physical address (PA)**
    - **physical addressing** -- CPU uses physical addresses to access memory,
      - early PCs used physcal addressing

    - digital signal processors, embedded microcontrollers, Cray supercomputers still use physical addressing
- ***Virtual addressing*** -- used by modern processors
    - CPU accesses main memory by generating a **virtual address (VA)**
        - converted to the appropriate physical address before being sent to main memory -- **address translation** by **memory mangagement unit (MMU)** (hardware on the CPU chip)

H3 §9.2 Address Space

- **Address space** - ordered set of nonnegative integer addresses
    - if the integers are consecutive --> **linear address space**
    - for simplicity, in CSAPP, always assume linear address spaces

- In a system with **virtual memory**, the CPU generates **virtual addresses** from an address space of. $N = 2^n$ addresses (**virtual address space**)

$$\{0, 1, 2, \ldots, N-1\}$$

    - size of an address space is characterized by the number of bits that are needed to represent the largest addrress
        - **virtual address space** with $N = 2^n$ address is called an **n-bit address space**
            - modern systems typically support **32-bit** or **64-bit**

- System also has a **physical address space** that corresponds to the M bytes of physical memory in the system

$$\{0, 1, 2, \ldots, M-1\}$$

    - M is **not required** to be a power of 2, but assume that $M = 2^m$

- **Address space** makes a clean distinction between data objects (**bytes**) and their attritbutes (**addresses**)
    - allow each data object to have multiple independent addresses

- Each byte of main memory has a virtual address chosen from the virtual address space, and a physical address chosen from the physical address space

H3 §9.3 VM as a Tool for Caching

- VM is organized as an array of N contiguous byte-size cells stored on disk
- Data on disk (lower level) is partitioned into blocks that serve as the transfer units between the disk and the main memory (high level) (like **memory hierarchy**)
  - VM partitioned into **virtual pages** (VPs) -- $P = 2^p$ bytes
    1. **Unallocated** -- pages that have not yet been created by the VM system. Do not have any data associated with them --> **do not occupy any space on disk**
    2. **Cached** -- allocated pages that are currently cached in physical memory
    3. **Uncached** -- allocated pages that are not cached in physical memory
  - PM partitioned into **physical pages** (PPs) -- P bytes in size

H5 ¶9.3.1 DRAM Cache Organization

- **SRAM cache** -- L1, L2, L3 cache memories (CPU <-> main memory)
- **DRAM cache** -- **VM system's cache** (caches virtual pages in main memory)

- Misses in DRAM caches are very expensive compared to misses in SRAM caches
  - ∵ DRAM caches misses are served from disk (1,000,000 times slower than SRAM)
- Cost of reading the first byte from a disk sector is about 100,000 times slower than reading successive bytes in the sector

--> **Virtual pages tend to be largue** (4KB ~ 2MB)

--> **DRAM caches are <u>fully associative</u>**

```
- any virtual page can be placed in any physical page
```

--> OS uses much more **sophisticated replacement algorithms for DRAM caches**

--> **DRAM caches** always use **write-back** instead of write-through

H5 ¶9.3.2 Page Tables

- VM system must have some way to determine **if a virtual page is cached** somewhere **in DRAM**
  - which **physical page** it is **cached in** ?
  - If there is a miss, the system must determine where the virtual page is stored on disk, select a victim page in physical memory, and copy the virtual page from disk to

DRAM, replacing the victim page

--> done by **OS SW** + **address translation HW in MMU** + **Data structure stored in physical memory (page table)**

--> **page table** maps virtual pages to physical pages

--> OS is responsible for maintaining the contents of the page table and transferring pages back and forth between disk and DRAM

- Page table is an array of **page table entries (PTEs)**
  - each page in the virtual address space has a PTE at a fixed offset in the page table
  - each PTE = **valid bit + n-bit address field**
    - **valid bit** - indicates whether the virtual page is currently cached in DRAM
      - if valid bit is **set** , **address field** indicates the **start of the corresponding physical page in DRAM** where the virtual page is currently cached in DRAM
      - if valid bit is **not set** , **null address** indicates that the **virtual page has not yet been allocated**
      - Otherwise, **address points to the start of the virtual page on disk**

- **DRAM cache is fully associative --> any physical page can contain any virtual page**

H5 ¶9.3.3 Page Hits

- When the CPU reads a word of virtual memory (in DRAM), the address translation hardware uses the virtual address as an index to locate PTE and read it from memory
  - If the valid bit is set, the address translation hardware knows that VP is cached in memory

  --> uses the physical memory address in the PTE to construct the physical address of the word

H5 ¶9.3.4 Page Faults

- **Page fault** -- DRAM cache miss
- Inferred from the valid bit, if VP is not cached, **page fault exception** is triggered
  - invokes **page fault exception handler** in the kernel, which selects a **victim page**
    - If **victim page** has been modified, then the kernel copies it back to disk

        - in either case, the kernel modifies the PTE for VP to reflect the fact that VP is no longer cached in main memory
    - then, the kernel copies VP from disk to PP in memory, and updates PTE
    - when the **handler returns**, it restarts the faulting instruction (resends the faulting virtual address to the address translation hardware)
        - now VP is cached in main memory, hence the page hit is handled normally by the address translation hardware

- In VM, different terminologies are used
    - **pages** = blocks
    - **swapping (paging)** -- transferring a page between disk and memory
    - **demand paging** -- waiting until the last moment to swap in a page (when a miss occurs)

H5 ¶9.3.5 Allocating Pages

- VP is allocated by creating room on disk and updating PTE to point to newly created page on disk
- **malloc, calloc**

H5 ¶9.3.6 Locality to the Rescue Again

- Total number of distinct pages that programs reference during an entire run might exceed the total size of physical memory, but the principle of **locality** promises that at any point in time they will tend to **work on a smaller set of active pages (working set = resident set**
    - after an initial overhead (working set paged into memory) , subsequent references result in hits with no additional disk traffic

--> **requires good temporal locality**

- If the working set size exceeds the size of physical memory

  --> **thrashing** --> pages swapped in and out continuously

H3 §9.4 VM as a Tool for Memory Management

- VM greatly simplified memory management and provided a natural way to protect memory
- Operating systems provide a **separate page table** --> **separate virtual address space for each process**
    - multiple virtual pages can be mapped to **the same shared physical page**

- **Demand paging** + **separate virtual address space** simplifies...
    1. Linking and loading
    2. Sharing of code and data
    3. Allocating memory to applications

H6 Simplifying linking

- Separate address space allows each process to use the same basic format for its memory image (regardless of where the code and data actually reside in physical memory)
    - every process on Linux system has a similar memory format
        - 64-bit address spaces, code segment **always start at virtual address 0x400000**
        - data segment follows the code segment after a suitable alignment gap
        - stack occupies the highest portion of the user process space and grows downward

    --> such uniformity **simplifies** the design and implementation of **linkers**, allowing them to produce **fully linked executables** that are independent of the ultimate location of the code and data in physical memory

H6 Simplifying loading

- VM makes it easy to load executable and shared object files into memory
- To load **.text** and **.data** sections of an object file into a newly created process, the Linux loader **allocates virtual pages** for the code and data segments, marks them as **invalid (not cached)** and points their page table entries to the appropriate locations in the object file
    - **Loader never actually copies any data from disk into memory**
- Mapping a set of contiguous virtual pages to an arbitrary location in an arbitrary file is known as **memory mapping**
    - system call **mmap** -- allows application programs to do their own memory mapping

H6 Simplifying sharing

- Separate address spaces provide the OS with consistent mechanism for managing sharing between **user processes and the OS**
  - rather than including separate copies of the kernel and standard C library in each process, the OS can arrange for multiple processes to share a single copy of this code by **mapping the appropriate virtual pages in different prcesses to the same physical pages**

H6 Simplifying memory allocation

- VM provides a simple mechanism for **allocating additional memory to user processes**
  - If a program running in a user process **requests additional heap space** (e.g. calling **malloc**)
    - the OS allocates an appropriate number of contiguous virtual memory pages and maps them to arbitrary physical pages located anywhere in physical memory

    --> **No need for the OS to locate contiguous pages of physical memory** (can be scattered randomly in PM)

H3 §9.5 VM as a Tool for Memory Protection

- User process should not be allowed to...
  - modify its read-only code section
  - read/modify any of the code and data structures in the kernel
  - read/write the private memory of other processes
  - modify any virtual pages that are shared with other processes (unless all parties explicitly allow it)

  --> **providing separate virtual address spaces** makes it easy to **isolate the private memories of different processes**

- **Address translation** provides even **finer access control** by adding some **additional permission bits to the PTE**
  - e.g. SUP bit (indicate whether processes must be running in kernel (supervisor) mode), READ bit, WRITE bit ....

--> if an instruction violates these permissions, then the CPU triggers a general proetection fault that transfers control to an exception handler in the kernel, which sends a **SIGSEGV** signal to the offending process

H3 §9.6 Address Translation

H6 Address translation symbols

- Basic parameters
  - $N = 2^n$: number of addresses in virtual address space
  - $M = 2^m$: number of addresses in physical address space
  - $P = 2^p$: page size(bytes)
- Components of a virtual address (VA)
  - VPO: virtual page offset (bytes)
  - VPN: virtual page number
  - TLBI: TLB index
  - TLBT: TLB tag
- Components of a physical address (PA)
  - PPO: physical page offset (bytes)
  - PPN: physical page number
  - CO: byte offset within cache block
  - CI: cache index
  - CT: cache tag

- Formally, address translation is a mapping between the elements of an N-element virtual address space (VAS) and an M-element physical address space (PAS)

$$MAP : VAS \rightarrow PAS \cup \emptyset$$

where

$$MAP(A) = \begin{cases} A' & if\ data\ at\ virtual\ addr.\ A\ are\ present\ at\ physical\ addr.\ A'\ in\ PAS \\ \emptyset & if\ data\ at\ virtual\ addr.\ A\ are\ not\ present\ in\ physical\ memory \end{cases}$$

- **MMU** uses the page table to perfroms such mapping
  - **page table base register (PTBR)** points to the current page table
  - n-bit virtual address has two componenets
    - p -bit **virtual page offset (VPO)**
    - $(n-p)$ -bit **virtual page number (VPN)**

- MMU uses the **VPN** to select the appropriate PTE
- then the corresponding physical address is the concatenation of the **physical page number (PPN)** from the page table entry and the **VPO** from the virtual address
    - **physical page offset (PPO)** is identical to the **VPO**

- Steps when there is a **page hit**
    1. processor generates a virtual address and sends it to the MMU
    2. MMU generates the PTE address and requests it from the cache / main memory
    3. Cache / main memory returns the PTE to the MMU
    4. MMU constructs the physical address and sends it to the cache / main memory
    5. Cache / main memory returns the requested data word to the processor

- Steps when there is a **page fault**
    1. processor generates a virtual address and sends it to the MMU
    2. MMU generates the PTE address and requests it from the cache / main memory
    3. Cache / main memory returns the PTE to the MMU
    4. Valid bit in the PTE is zero, so the MMU triggers an exception, which transfers control in the CPU to a page fault exception handler in the OS kernel
    5. Fault handler identifies a victim page in PM, and if that page has been modified, pages it out to disk
    6. Fault handler pages in the new page and updates the PTE in memory
    7. Fault handler returns to the original process, causing the faulting instruction to be restarted. The CPU resends the offending virtual address to the MMU. Since the VP is now cached in PM, there is a hit, hence performs step 4, 5 as if there was a hit

H5 ¶9.6.1 Integrating Caches and VM

- When the system uses both virtual memory and SRAM caches...
    - most systems opt for **physical addressing**
        - straightforward for multiple processes to have blocks in the cache at the same time and to share blocks from teh same virtual pages
        - cache does not have to deal with protection issues (·.· access rights are checked as part of the address translation process)
- Address translation occurs before the cache lookup

H5 ¶9.6.2 Speeding Up Address Translation with a TLB

- Every time the CPU generates a VA, the MMU must refer to a PTE in order to translate the VA into PA
  - in the worst case, this requires an additional fetch from memory

  --> including a **small cache of PTEs in the MMU** called a **translation lookaside buffer (TLB)**

```
- each line holds a block consisting of a single PTE
- uses high degree of associativity
- IF TLB has $T=2^t$ sets, then the **TLB index (TLBI)**
consists of the $t$ least significant bits of the VPN, and the
**TLB tag (TLBT)** consists of the remaining bits in the VPN
```

- Address translation steps with TLB
  - all of the address translation steps are performed inside the on-chip MMU --> fast

1. CPU generates a VA
2. MMU fetches the appropriate PTE from the TLB
3. MMU translates VA to PA and sends it to the cache / main memory
4. Cache / main memory returns the requested data word to the CPU

--> IF there is a TLB miss, MMU must fetch the PTE from the L1 cache, and newly fetched PTE is stored in the TLB (possibly overwriting an existing entry)

H5 ¶9.6.3 Multi-Level Page Tables

- **Hierarchy of page tables** for compacting the page table
- Each PTE in the **level 1 table** is responsible for mapping a **4MB** chunck of the VAS
  - each chunk consists of 1,024 contiguous pages
    - if every page in chunk i is unallocated, level 1 PTE i is null
    - if at least one page in chunk i is allocated, then level 1 PTE i points to the base of a level 2 page table
- Each PTE in the **level 2 table** is responsible for mapping a **4KB** page of VM
  - same size as a page

--> Reduces memory requirements in two ways

1. If a PTE in the level 1 table is null, then the corresponding level 2 page table does not even have to exist

- most of the 4GB virtual address space for a typical program is unallocated

2. Only the level 1 table needs to be in main memory at all times
    - level 2 page tables can be created and paged in and out by the VM system as they are needed

--> seems expensive and impractical but **TLB** caches PTEs from the page tables at the different levels

--> not significantly slower than with single-level page tables

H5 ¶9.6.4 Putting It Together: End-to-End Address Translation

- Assumptions for simplcity
    - Memory is byte addressable
    - Memory accesses are to 1-byte word
    - VA are 14 bits wide (n = 14)
    - PA are 12 bits wide (m = 12)
    - Page size is 64 bytes (P = 64)
    - TLB is 4-way set associative with 16 total entries
    - L1 d-cache is physically addressed and direct mapped, witha 4-byte line size and 16 total sets

--> since each page is $2^6 = 64$ bytes, the low-order 6bits of the VA and PA serve as the VPO and PPO

--> hence, high-order 8 bits of the VA serve as the VPN + high-order 6 bits of the PA serve as the PPN

- **TLB** -- virtually addressed using the bits of the VPN
    - since the TLB has four sets, the 2 low-order bits of the VPN serve as the TLBI and the remaining 6 high-order bits serve as TLBT that distinguishes the different VPNs
- **Page table**
    - single-level design with a total of $2^8 = 256$ page table entries
- **Cache**
    - direct-mapped cache is addressed by the fields in the physical address
    - each block is 4 bytes, the low-order 2 bits of the physical address serve as the block offset(CO)

- there are 16 sets --> next 4 bits serve as the set index (CI)
- remaining 6 bits serve as the tag (CT)

- Basic procedure
  1. MMU extracts the VPN from the virtual address and checks with the TLB to see if it has cached a copy of PTE from some previous memory reference
  2. TLB extracts the TLBI and the TLBT from the VPN, hits on a valid match then returns the cached PPN to the MMU
     2. If the TLB had missed, then the MMU would need to fetch the PTE from main memory
  3. MMU forms the PA by concatenating the PPN from the PTE with the VPO from the VA
  4. MMU sends the PA to the cache, which extracts the CO, CI, and CT from the PA
  5. If the tag matches, the cache detects a hit, reads out the data byte at offset CO, and returns it to the MMU, which then passes it back to the CPU

H3 §9.7 Case Study: The Intel Core i7 / Liunx Memory System

Skipping this section on note

H3 §9.8 Memory Mapping

- Linux initializes the contents of a virtual memory area by associating it with an object on disk by **memory mapping**

  --> can be mapped to one of two types of objects

  1. **Regular file in the Linux file system**
     - area can be mapped to a contiguous section of a regular disk file (e.g. executable object file)
     - file section is divided into page-size pieces, with each piece containing the initial contents of a VP
     - due to demand paging, none of these VP's is actually swapped into physical memory until the CPU first **touches** the page
     - if the area > file section, then the area is padded with zeors

2. **Anonymous file**
    - area can also be mapped to an anonymous file, created by the kernel, that contains all **binary zeros**
    - the first time the CPU touches a VP in such an area, the kernel finds an appropriate victim page in physical memory, swaps out th evictim page, overwrites the victim page with binary zeros, and updates the page table to mark the page as resident
    - no data are actually transferred between disk and memory
    - sometimes called **demand-zero pages**

--> In either case, once a VP is initialized, it is swapped back and forth between a specail **swap file** maintained by the kernel (swap file = swap space = swap area)

- at any point in time, the swap space bounds the total amount of virtual pages that can be allocated by the currently running processes

H5 ¶9.8.1 Shared Objects Revisited

- Process abstraction promises to provide each process with its own private virtual space that is protected from errant writes or reads by other processes
    - However, many processes have identical read-only code areas & read-only run-time library code
        - bash, standard C library functions

      --> extremely wasteful for each process to keep duplicate copies of these commonly used codes in physical memory

- Object can be mapped into an area of virtual memory as either a **shared object** or a **private object**

1. **Shared object**
    - If a process maps a shared object into an area **(shared area)** of its virtual address space, then any writes that the process makes to that area are visible to any other processes that have also mapped the shared object into their virtual memory
    - changes are reflected in the original object on disk
2. **Private object**
    - changes made to an area **(private area)** mapped to a private object are not visible to other processes

- any writes that the process makes to the are are not reflected back to the object on disk

- Single copy of the shared object needs to be stored in physical memory, even though the object is mapped into multiple shared areas
  - physical pages can not be contiguous (actually not contiguous in general)

- **Private objects** are mapped into virtual memory using **copy-on-write (COW)**
  - only one copy of the private object stored in physical memory
    - multiple processes map a private object into different areas of their virtual memories but share the same physical copy of the object
      - for each process that maps the private object, the page table entries for the corresponding private area are flagged as read-only, and the area struct is flagged as **private COW**
  - as long as neither process attempts to write to its respective private area, they continue to share a single copy of the object in physical memory
  - **BUT!** as soon as a process attempts to write to some page in the private area, the write triggers a protection fault
    - fault handler notices that the protection exception was caused by the process trying to write to a page in a private COW area, creating a new copy of the page in physical memory, updates the page table entry to point to the new copy, and then restores write permission to the page
    - when the fault handler returns, the CPU re-executes the write, which now proceeds normally on the newly created page

H5 ¶9.8.2 The fork Function Revisited

- **fork** function creates a new process with its own independent virtual address space
- When the **fork** function is called by the current process, the kernel creates various data structures for the new process and assigns it a unique PID
  - creates exact copies of the current process's **mm_struct** , area structs, and page tables
  - flags each page in both processes as **read-only** , and flags each area struct in both processes as **private COW (copy-on-write)**
- When the **fork** returns in the new process, the new process now has an **exact copy** of the virtual memory as it existed when the fork was called

- when either of the processes performs any subsequent **writes,** the **COW** mechanism creates new pages, preserving the abstraction of a private address space for each process

H5 ¶9.8.3 The execve Function Revisited

- Virtual memory and memory mapping also play key roles in the process of loading programs into memory
- Suppose we are running

```
1   execve("a.out", NULL, NULL);
```

--> **execve** loads and runs the program contained in the executable object file **a.out** within the current process, replacing the current program with the **a.out** program

- Loading and running **a.out** requires the following steps

1. **Delete existing user areas**
   - delete the existing area structs in the user portion of the current process's virtual address
2. **Map private areas**
   - create new area structs for the code, data, bss, and stack areas of the new program
     - all of these new areas are **privae COW**
   - code & data areas are mapped to the **.text** and **.data** sections of the **a.out** file
   - **bss** area is **demand-zero**, mapped to an anonymous file whose size is contained in **a.out**
   - **stack & heap** are also **demand-zero**, initially of zero length
3. **Map shared areas**
   - if **a.out** was linked with shared objects (e.g. C library), then these objects are **dynamically linked into the program** , and then mapped into the shared region of the user's virtual address space
4. **Set the program counter (PC)**

- set the program counter in the current process's context to point ot the entry point in the code area

H5 ¶9.8.4 User-Level Memory Mapping with the mmap Function

- Linux processes can use the **mmap** function to create new areas of virtual memory and to map objects into these areas

```
#include <unistd.h>
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
  // Returns: pointer to mapped area if OK, MAP_FAILED(-1) on error
```

```
- **mmap** asks the kernel to create a new virtual memory area that starts at address **start**, and to map a contiguous chunk of the object specified by file descriptor **fd** to the new area
- contiguous object chunk has a size of **length** bytes and starts at an offset of **offset** bytes from the beginning of the file
- **start** address is usually specified as **NULL** (for convenience, we will always assume it to be **NULL**)
```

- **prot** contains bits that describe the access permissions of the newly mapped virtual memory area
  - **PROT_EXEC** -- pages in thea area consist of instructions that may be executed
    - **PROT_READ** -- pages in the area may be read
  - **PROT_WRITE** -- pages in the area may be written
  - **PROT_NONE** -- pages in the area cannot be accessed
- **flags** (consists of bits) describe the type of the mapped object
  - **MAP_ANON** bit set -- backing store is an **anonymous object** ---> **demand-zero**
  - **MAP_PRIVATE** -- **private COW**
  - **MAP_SHARED** -- shared object

[Example C code]

```
bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE|MAP_ANON, 0, 0);
```

--> asks the kernel to create a new read-only, private, demand-zero area of virtual memory containing **size** bytes

```
- If the call is successful, then **bufp** contains the address of the new area
```

- **munmap** function deletes regions of virtual memory

```
#include <unistd.h>
#inculde <sys/mman.h>
int munmap(void *start, size_t length);
      // Returns: 0 if OK, -1 on error
```

--> deletes the area starting at virtual address **start** and consisting of the next **length** bytes

--> subsequent references to the deleted region result in **segmentation fault**

H3 §9.9 Dynamic Memory Allocation

- C programmers use **dynamic memory allocator** when they need to acquire additional virtual memory at run time
  - Dynamic memory allocator maintains an area of a process's virtual memory == **heap**
    - area of demand-zero memory that begins immediately after the uninitialized data area and grows upward (toward higher addresses)
    - for each process, the kernel maintinas a variable **brk (break)** that points to the top of the heap
    - allocator maintains **heap** as a collection of various-size **blocks**
      - each block is a contiguous chunk of virtual memory that is either **allocated** or **free**
        - **allocated block** -- explicitly reserved for use by the application
          - remains allocated until it is freed
            - freed explicitly by the application or implicitly by the memory allocator itself

        - **free block** -- available to be allocated
            - remains free until it is explicitly allocated by the application
    - two types of allocators (differ about which entry is responsible for freeing allocated blocks)
        - **Explicit allocators** -- require the application to explicitly free any allocated blocks
            - **malloc** & **free**
                - allocate a block by calling **malloc** function
                - free a block by calling **free** function
        - **Implicit allocators** -- require the allocator to detect when an allocated block is no longer being used by the program and then free the block
            - known as ***garbage collectors***
                - automatically freeing unused allocated blocks (**garbage collection**)
            - higher-level languages (Lisp, ML, Java)

H5 ¶9.9.1 The malloc and free Functions

```
#include <stdlib.h>
void *malloc(size_t size);
    // Returns: pointer to allocated block if OK, NULL on error
```

- **malloc** function returns a pointer to a block of memory of at least **size** bytes that is suitably aligned for any kind of data object that might be contained in the block
    - alignment depends on whether the code is compiled to run in 32-bit mode (**gcc -m32**) or 64-bit mode (default)
        - in 32-bit mode, **malloc** returns a block whose address is always a multiple of 8
        - in 64-bit mode, address is always a multiple of 16
- If **malloc** encounters a problem, it returns **NULL** and sets *errno*

- **malloc** does not initialize the memory it returns
    - if you want initialized dynamic memory, use **calloc** instead
        - thin wrapper around the **malloc** that initializes the allocated memory to zero

- To change the size of a previously allocated block, use **realloc** function

- Dynamic memory allocators can allocate or deallocate heap memory explicitly by using the **mmap** & **munmap** functions
  - or use **sbrk** function

```
#include <unistd.h>
void *sbark(intptr_t incr);
  // Returns: old brk pointer on success, -1 on error
```

  - **sbrk** grows or shrinks the heap by adding **incr** to the kernel's **brk** pointer
    - if successful, returns the old value of **brk**
    - if fails, returns -1 and sets **errno** to **ENOMEM**
    - if **incr** is zero, then **sbrk** returns the current value of **brk**

- Free allocated heap blocks by calling the **free** function

```
#include <stdlib.h>
void free(void *ptr);
    // Returns: nothing
```

- **ptr** must point to the beginning of an allocated block that was obtained from **malloc, calloc, realloc**
  - **IF not, undefined behavior**
    - since **free** returns nothing, **free** gives no indication to the application that something is wrong

H5 ¶9.9.2 Why Dynamic Memory Allocation?

- Since programs do not know. the sizes of certain data structures until the program actually runs

[Example **BAD** C code]

```c
#include "csapp.h"
#define MAXN 15213

int array[MAXN];

int main() {
  int i, n;

  scanf("%d", &n);
  if (N > MAXN)
    app_error("Input file too big");
  for (i = 0; i < n; i++)
    scanf("%d", &array[i]);
  exit(0);
}
```

- Value of MAXN is arbitrary and has no relation to the actual amount of available virtual memory on the machine
- Further, if the user wanted to read a file that was larger than MAXN, the only resource would be to recompile the program with a larger MAXN

--> Hence, it is better to allocate the array **dynamically at run time**

[Example **Better** C code]

```c
#include "csapp.h"

int main() {
  int *array, i, n;

  scanf("%d", &n);
  array = (int *)Malloc(n * sizeof(int));
  for (i = 0; i < n; i++)
    scanf("%d", &array[i]);
  free(array);
  exit(0);
}
```

- Maximum size of the array is limited only by the amount of available virtual memory

H5 ¶9.9.3 Allocator Requirements and Goals

- Explicit allocators must operate within some rather strict constraints
  1. **Handling arbitrary request sequences**
     - allocator cannot make any assumptions about the ordering of allocate and free requests
  2. **Making immediate responses to requests**
     - allocator must respond immediately to allocate requests
       - not allowed to reorder or buffer requests in order to improve performance
  3. **Using only the heap**
     - in order for the allocator to be scalable, any nonscalar data structures used by the allocator must be stored in the heap itself
  4. **Aligning blocks**
     - must align blocks in such a way that they can hold any type of data object
  5. **Not modifying allocated blocks**
     - can only manipulate or change free blocks
       - not allowed to modify or move blocks once they are allocated
       - compaction of allocated blocks are not permitted

- Maximizing **Throughput** vs **Memory utillization**?
  1. **Maximizing throughput**

     Given some sequence of n allocate and free requests

     $$R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$$

     --> we would like to maximize an allocator's **throughput** (number of requests that it completes per unit time)

     --> can maximize throughput by **minimizing the average time** to satisfy allocate and free requests

  2. **Maximizing memory utilization**
     - total amount of virtual memory allocated by all of the processes in a system is **limited by the amount of swap space on disk**
     - use **peak utilization**

       Given some sequence of n allocate and free requests

       $$R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$$

If an application requests a block of p bytes, then the resulting allocated block has a **payload** of p bytes

--> After request R_k has completed, let the **aggregate payload,** denoted P_k, the sum of the payloads of the currently allocated blocks, and let H_k denote the current size of the heap

---> then the **peak utilization** over the first $k+1$ requests, denoted by U_k is given by

$$U_k = \frac{max_{i \leq k} P_i}{H_k}$$

H5 ¶9.9.4 Fragmentation

- Primary cause of poor heap utilization : **fragmentation**
  - unused memery is not available to satisfy allocate requests
    1. **Internal fragmentation**
    2. **External fragmentation**

1. **Internal fragmentation**
   - when an allocated block is larger than the payload
     - implemenation of an allocator might impose a minimum size on allocated blocks that is greater than some requested payload
     - allocator might increase the block size in order to satisfy alignment constraints
   - depends only on the pattern of previous requests & allocator implementation --> easy to spot

2. **External fragmentation**
   - when there is enough **aggregate free memory** to satisfy an allocate request, but **no single free block** is large enough to handle the request
   - depends not only on the pattern of previous requests and the allocator implementation but also on the **pattern of *future* requests**
   - Hence, allocators typically employ heuristics that attempt to maintain **small numbers of larger free blocks** rather than large numbers of smaller free blocks

H5 ¶9.9.5 Implementation Issues

- Practical allocator that maintains a better balance between **throughput & utilization** must consider...
  1. **Free block organization**
     - how do we keep track of free blocks?
  2. **Placement**
     - how do we choose an appropriate free block in which to place a newly allocated block?
  3. **Splitting**
     - after placing a newly allocated block in some free block, what do we do with the remainder of ther free block?
  4. **Coalescing**
     - what do we do with a block that has just been freed?

H5 ¶9.9.6 Implicit Free Lists

- Practical allocator needs some data structure that allows it to distinguish block boundaries and to distinguish **allloated and free blocks**
  - embed this information in the blocks themselves
    - block consists of a **one-word header** + payload + (possible) padding
      - **header** encodes the **block size** + **indicator** to show whether the block is allocated or free
        - suppose the block size is always a multiple of 8, then the 3 low-order bits are always zero --> can use these bits to indicate whether the block is allocated or free
          - e.g. allocated block size of 24 (0x18) bytes
            - header would be 0x00000018 | 0x1 = 0x00000019
          - free block with a block size of 40 (0x28) bytes
            - 0x00000028 | 0x0 = 0x00000028
      - header is followed by the **payload** that the application is requested when it called **malloc**
      - followed by an **unused padding** that can be any size
        - may be part of an allocator's strategy for reducing external fragmentation

- Named **implicit free list** because free blocks are linked implicitly by the size fields in the headers
    - allocator can indirectly traverse the entire set of free blocks by traversing **all of the blocks in the heap**
        - need specailly marked **end block**

- Advantage: Simplicity
- Disadvantage: Cost of any operation that requires a search of the free list will be **linear in the total number of allocated + free blocks in the heap**

- System's alignment requirement and the allocator's choice of block format impose a **minimum block size** on the allocator
    - no alloated or free block may be smaller than this minimum

H5 ¶9.9.7 Placing Allocated Blocks

- When an application requests a block of k bytes, the allocator searches the free list for a free block that is large enough to hold the requested block

  --> determined by the **placement policy**

  1. **First fit**
     - searches the free list form the beginning and chooses the first free block that fits
     - **Advantage**
       - tends to retain larger free blocks at the end of the list
     - **Disadvantage**
       - tends to leave "splinters" of small free blocks toward the beginning of the list, increasing the search time for larger blocks

  2. **Next fit**
     - starts each search where the previous search left off
     - **Advantage**
       - significantly faster than first fit
         - if we found a fit in some free block the last time, there is a good chance that we will find a fit the next time in the remainder of the block

- **Disadvantage**
  - worse memory utilization

3. **Best fit**
   - examines every free block and chooses the free block with the smallest size that fits
   - **Advantage**
     - best memory utilization
   - **Disadvantage**
     - with simple free list organizations (implicit free list), it requires an exhaustive search of the heap

H5 ¶9.9.8 Splitting Free Blocks

- Once the allocator has located a free block that fits, it must make another policy decision about **how much of the free block to allocate**
  - **entire free block?**
    - simple & fast
    - **But... internal fragmentation!**
      - (with good placement policy, might be acceptable though
  - if the fit is not good, then the allocator will usually **split** the free block into **allocated block + new free block**

H5 ¶9.9.9 Getting Additional Heap Memory

- If the allocator is unable to find a fit for the requested block, it tries to create some larger free blocks by **merging (coalescing) free blocks** that are physically adjacent in memory
- **However,** if it does not yield a sufficiently large block, or if the free blocks are already maximally coalesced,
  - allocator asks the **kernel for additional heap memory** by calling the **sbrk** function
    - allocator transforms the additional memory into one large free block, inserts the block into the free list, then places the requested block into this new free block

H5 ¶9.9.10 Coalescing Free Blocks

- When the allocator frees an allocated block, there might be other free blocks that are adjacent to the newly freed block
  - such adjacent blocks can cause **false fragmentation**
    - lot of available free memory divided into small, unusable free blocks

    --> **MUST** merge adjacent free blocks in a process known as **coalescing**

- **Immediate coalescing**
  - merging any adjacent blocks each time a block is freed
  - can cause a form of **thrashing**
    - block repeatedly coalesced and then split soon thereafter
  - In CSAPP, assume allocators use immediate coalescing

- **Deferred coalescing**
  - waiting to coalesce free blocks at some time later
    - until some allocation request fails --> scan the entire heap, coalescing all free blocks
  - fast allocators often use some form of deferred coalescing

H5 ¶9.9.11 Coalescing with Boundary Tags

- Let's refer to the block we want to free as the **current block**
  - header of the current block points to the header of the next block --> can determine if the next block is free
    - if the next block is free, its size is simply added to the size of the current header and the blocks are coalesced in **constant time**
  - what if we need to coalesce the **previous block**?
    - given an implicit free list, need to **search the entire list** until we reached the current block
      - each call to **free** would require time linear in the size of the heap

- Use **Boundary tags!**
  - it allows for constant-time coalescing of the previous block

- add **footer (boundary tag)** at the end of each block
    - replica of the header
    - can determine the starting location and status of the previous block in constant time

- Four cases exist when the allocator frees the current block
    1. Preivous & Next are both allocated
        - no coalescing possible
        - status of the current block is simply changed from allocated to free

    2. Previous is allocated, Next is free
        - current block is merged with the next block
        - header of the current block and the footer of the next block are updated with the combined sizes of the current and next blocks

    3. Previous if free, Next is allocated
        - previous block is merged with the current block
        - header of the previous block and the footer of the current block are updated with the combined sizes of the two blocks

    4. Previous & Next are both free
        - all three blocks are merged to from a single free block, with the header of the previous block and the footer of the next block updated with the combined sizes of the three blocks

--> In each case, **coalescing is performed in <u>constant time!</u>**

--> however, it can cause significant **memory overhead** if an application manipulates many small blocks

--> **include footers for free blocks only!**

H5 **¶9.9.12 Putting It Together: Implementing a Simple Allocator**

Skipping this section on note

H5 ¶9.9.13 Explicit Free Lists

- Implicit free list is not appropriate for a general purpose allocator because block allocation time is linear in the total number of heap blocks

- What if we organize the free blocks into some form of **explicit data structure?**
  - by definition, the body of a free block is not needed

    --> pointers that implement the data structure can be stored within the bodies of the free blocks
    - heap can be organized as a **doubly linked free list** by including a **pred** (predecessor) and **succ** (successor) pointer in each free block
      - reduces the first-fit allocation time from linear in the total number of blocks to **linear in the number of free blocks**
      - **However,** the time to free a block can be either linear or constant, depedning on the policy we choose for **ordering the blocks** in the free list
        - **LIFO** (last-in first-out) ordering?
          - inserting newly freed blocks at the beginning of the list
          - with LIFO & first-fit placement policy, the allocator inspects the most recently used blocks first

          ---> freeing a block can be performed in constant time

          ---> If boundary tags are used, coalescing can also be performed in constant time
        - **Maintain the list in <u>address order?</u>**
          - address of each block in the list is less than the address of its successor
          - freeing a block requires a linear-time search to locate the appropriate predecessor
          - better memory utilization (≈ best fit) than LIFO

- **Disadvantage** of explicit lists in general?
  - free blocks must be large enough to contain all of the necessary pointers + header + (possibly) footer

--> larger minimum block size --> increases the potential for internal fragmentation

H5 ¶9.9.14 Segregated Free Lists

- To reduce the allocation time, **segragated storage** is used to maintain multiple free lists, where each list holds blocks that are roughly the same size
  - partition the set of all possible block sizes into equivalent classes calssed **size classes**
    - partition the block sizes by powers of 2

      {1}, {2}, {3,4}, {5-8}, ... , {1,025-2,048}, {2,049-4,096}, ...
    - assign small blocks to their own size classes and partition large blocks by powers of 2

      {1}, {2}, {3}, ..., {1,023}, {1,024}, {1,025-2,048}, ...
  - Allocator maintains an array of free lists, with one free list per size class, ordered by increasing size
    - when the allocator needs a block of size n, it searches the appropriate free list
      - if it cannot find a block that fits, it searches the next list, and so on

1. **Simple segregated storage**
2. **Segragated fits**

H6 Simple Segregated Storage

- Free list for each size class contains same-size blocks, each the size of the largest element of the size class
  - e.g. if some size class is defined as [17-32], then the free list for the class consists entirely of blocks of size 32

- To **allocate** a block of some given size, we check the appropriate free list
  - if the list is **not empty**, simply **allocate the first block** in its entirety
    - **free blocks are never split** to satisfy allocation requests
  - if the list is **empty**, the allocator **reqeusts a fixed-size chunk of additional memory from the OS** (typically a multiple of the page size), dividing the chunk into equal-size blocks, and links the blocks together to **form the new free list**

- To **free** a block, the allocator simply **inserts the block at the front of the appropriate free list**

- **Advantages**
  - Allocating and freeing blocks are both fast constant-time operations
  - Same-size blocks in each chunk, no splitting, and no coalescing --> **little per-block memory overhead**
    - size of an allocated block can be inferred from its address
    - no coalescing --> allocated blocks do not need an allocated / free flag in the header & footer
    - allocate & free operations insert / delete blocks at the beginning of the free list, the list need only be **singly linked** (not doubly)

- **Disadvantages**
  - **free blocks never split** --> ***internal fragmentation***
  - **free blocks never coalesced** --> ***external fragmentation***

H6 Segregated Fits

- Allocator maintains an array of free lists
  - each free list is associated with a size class and is organized as some kind of explicit or implicit list
  - each list contains potentially different-size blocks whose sizes are members of the size class

- To allocate a block,
  - determine the size class of the request and do a **first-fit search** of the appropriate free list for a block that fits
    - If found, (optionally) split it and insert the fragment in the appropriate free list
    - If not found a block that fits, search the free list for the next larger size class (repeat until we find a block that fits)
    - If **none of the free lists** yields a block that fits, request **additional heap memory from the OS** , allocate the block in this new heap memory, and place the remainder in the appropriate class size

- To free a block,
  - coalesce and place the result on the appropriate free list

- **Popular choice** (**GNU malloc package uses it**)
  - fast and memory efficient
    - searches are limited to particular parts of the heap --> **reduced search times**
    - simple fast-fit search of a segregated free list approximates a best-fit search of the entire heap --> **better memory utilization**

H6 Buddy Systems

- **Buddy system** -- segregated fits where each size class is a power of 2
  - Given a heap of 2^m words, we maintain a separate free list for each block size 2^k, where $0 \leq k \leq m$
    - originally, there is one free block of size 2^m words
- Requested block sizes are rounded up to the nearest power of 2

- To allocate a block of size 2^k, we find the first available block of size 2^j, s.t. $k \leq j \leq m$
  - if $j = k$, done!
  - otherwise, recursively split the block in half until $j = k$
    - each remaining half (**buddy**) is placed on the appropriate free list

- To free a block of size 2^k, **continue coalescing with the free buddies** (until encountering an allocated buddy)

- Given the address and size of a block, it is easy to compute the address of its buddy
  - addresses of a block and its buddy differ in exactly one bit position

- **Advantage**
  - fast searching and coalescing
- **Disadvantage**
  - Power-of-2 req. on the block size can cause **significant internal fragmentation**

  --> not appropriate for general-purpose workloads

H3 §9.10 Garbage Collection

- **Garbage collector** -- dynamic storage allocator that automatically frees allocated blocks that are no longer needed by the program
  - periodically identifies the garbage blocks and makes the appropriate calls to **free** to place those blocks back on the free list

- John McCarthy's **Mark&Sweep algorithm**
  - can be built on top of an existing **malloc** package to provide garbage collection for C and C++ programs

H5 ¶9.10.1 Garbage Collector Basics

- Garbage collector views memory as a directed **reachability graph**
  - each nodes are partitioned into a set of **root nodes** and a set of **heap nodes**
    - each **heap node** corresponds to an **allocated block in the heap**
      - directed edge $p \rightarrow q$ -- some location in block p points to some location in block q
    - **root nodes** correspond to locations not in the heap that conatin pointers into the heap
      - registers, variables on the stack, global variables in the read/write data are of VM

- p is **reachable** if there exists a **directed path** from any root node to p
- At any point in time, the **unreachable nodes** correspond to **garbage** that can never be used again by the application
  - **garbage collector** maintains some representation of the reachability graph and periodiclaly reclaim the unreachable nodes by freeing them and returning them to the free list

- Garbage collectors for ML and Java can maintain an **exact representation of the reachability graph** --> reclaim all garbage
- **However,** collectors for C and C++ cannot maintain exact representations of the reachability graph in general

  --> **Conservative garbage collectors**

    - each reachable block is correctly identified as reachable
    - but... some unreachable nodes might be incorrectly identified as reachable

- Collectors can provide their service on demand, or can run as **separte threads in parallel** with application, continuously updating the reachability graph and reclaiming garbage

- Basic process..
    - the application calls **malloc** when it needs heap space
        - if **malloc** is unable to find a free block that fits, it calls the **garbage collector** to reclaim some garbage to the free list
            - the collector identifies the garbage blocks and returns them to the heap by calling the **free** function
                - **NOTE** that **collector calls free** instead of the application
    - when the collector returns, **malloc** tries again to find a free block that fits
        - if fails, it can ask the OS for additional memory

H5 ¶9.10.2 Mark&Sweep Garbage Collectors

[C codes for **mark** & **sweep** functions]

```
/* mark function */
void mark(ptr p) {
  if ((b = isPtr(p)) == NULL)
    return;
  if (blockMarked(b))
    return;
  markBlock(b);
  len = length(b);
  for (i = 0; i < len; i++)
    mark(b[i]);
  return;
}

/* sweep function */
void sweep(ptr b, ptr end) {
```

```
16    while (b < end) {
17      if (blockMarked(b))
18        unmarkBlock(b);
19      else if (blockAllocated(b))
20        free(b);
21      b = nextBlock(b);
22    }
23    return;
24  }
```

- Some defined functions
  - **ptr** is defined as **typedef void *ptr**
  - **ptr isPtr(ptr p)** -- if **p** points to some word in an allocated block, returns a pointer **b** to the beginning of that block, returns **NULL** otherwise
  - **int blockMarked(ptr b)** -- returns **true** if block **b** is already marked
  - **int blockAllocated(ptr b)** -- returns **true** if block **b** is allocated
  - **void markBlock(ptr b)** -- marks block **b**
  - **int length(ptr b)** -- returns the length in words (excluding the header) of block **b**
  - **void unmarkBlock(ptr b)** -- changes the status of block **b** from marked to unmarked
  - **ptr nextBlock(ptr b)** -- returns the successor of block **b** in the heap

- **Mark&Sweep** garbage collector = **mark phase + sweep phase**
  - **mark phase** -- marks all reachable and allocated descendants of the root nodes
    - calls the **mark** function once for each root node
      - returns immediately if **p** does not point to an allocated and unmarked heap block
      - otherwise, marks the block and calls itself recursively on each word in block

      --> marks any unmarked and reachable descendants of some root node
    - at the end of the mark phase, any allocated block that is **not marked** is guaranteed to be unreachable --> **garbage** that can be reclaimed in the sweep phase

  - **sweep phase** -- frees each unmarked allocated block
    - single call to the **sweep** function
      - iterates over each block in the heap, freeing any unmarked allocated blocs that it encounters

  --> one of the spare low-order bits in the block header is used to indicate whether a block is marked or not

¶9.10.3 Conservative Mark&Sweep for C Programs

- Mark&Sweep is an appropriate approach for garbage collecting C program since it works in place without moving any blocks
- However... the C language has problems for the implementation of the **isPtr** function
  1. C does not tag memory locations with any type information

     --> no obvious way for **isPtr** to determine if its input parameter **p** is a pointer or not
  2. there would be no obvious way for **isPtr** to determine whether **p** points to some location in the payload of an allocated block
     - one possible **solution** is to maintain the set of allocated blocks as a balanced binary tree that maintains the invariant that all blocks in the left subtree are located at smaller addresses and all blocks in the right subtree are located in larger addresses
       - requires two additional fields (**left** and **right**) in the header of each allocated block
         - each field points to the header of some allocated block
       - **isPtr(ptr p)** function uses the tree to perfrom a **binary search** of an alloated blocks

- Prrimary reason that Mark&Sweep collectors for C programs must be **conservative** is that the C language does not tag memory locations with type information
  - scalars like **int**s or **float**s can masquerade as pointers
    - no way for collectors to distinguish the two

§9.11 Common Memory-Related Bugs in C Programs

¶9.11.1 Dereferencing Bad Pointers

- **scanf** bug

```
scanf("%d", val); // ?
/* in the best case, the program terminates immediately with an exception
   in the worst case, the contents of val correspond ot some valid read/write area of VM, and overwrite memory
*/
```

H5 ¶9.11.2 Reading Uninitialized Memory

- **bss memory locations** (uninitialized global C variables) are always initialized to **zeros** by the loader
- **However,** not true for **heap memory**
- Assuming that heap memory is initialized to zero can cause an unexpected result

```
/* Return y = Ax */
int *matvec(int **A, int *x, int n) {
  int i, j;

  int *y = (int *)Malloc(n * sizeof(int));

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      y[i] += A[i][j] * x[j];
  return y;
}
```

--> **Should not** assume that vector **y** has been initialized to zero

--> need to explicitly zero **y[i]** or use **calloc**

H5 ¶9.11.3 Allowing Stack Buffer Overflows

```
void bufoverflow() {
  char buf[64];

  gets(buf);  // possible stack buffer overflow
  return;
}
```

- **gets** function copies an arbitrary-length string to the buffer
  - use **fgets** function that limits the size of the input string

H5 ¶9.11.4 Assuming That Pointer and the Objects They Point to Are the Same Size

```
1  /* Create an nxm array */
2  int **makeArray1(int n, int m) {
3    int i;
4    int **A = (int **)Malloc(n * sizeof(int));
5
6    for (i = 0; i < n; i++)
7      A[i] = (int *)Malloc(m * sizeof(int));
8    return A;
9  }
```

- the program is intended to create an array of n pointers
  - but in L4, since we use **sizeof(int)** , we are actually creating an array of **int** s

H5 ¶9.11.5 Making Off-by-one

```
1  /* Create an nxm array */
2  int **makeArray2(int n, int m) {
3    int i;
4    int **A = (int **)Malloc(n * sizeof(int *));
5
6    for (i = 0; i <= n; i++)
7      A[i] = (int *)Malloc(m * sizeof(int));
8    return A;
9  }
```

--> created an n-element array of pointers in L4, but in L6, 7, initalizes $n + 1$ of its elements

H5 ¶9.11.6 Referencing a Pointer Instead of the Object It Points To

```
1  int *binheapDelete(int **binheap, int *size) {
2    int *packet = binheap[0];
3
4    binheap[0] = binheap[*size - 1];
5    *size--;    // should be (*size)--
6    heapify(binheap, *size, 0);
7    return(packet);
8  }
```

- In L5, the intent is to decrement the integer value pointed to by the **size** pointer, but the code is decrementing the pointer itself
  - should be (*size)--

H5 ¶9.11.7 Misunderstanding Pointer Arithmetic

```
1  int *search(int *p, int val) {
2    while (*p && *p != val)
3      p += sizeof(int); // should be p++
4    return p;
5  }
```

- Arithmetic operations on pointers are performed in units that are the **size of the objects they point to**, which are not necessarily bytes
  - L3 should be **p++**

H5 ¶9.11.8 Referencing Nonexistent Variables

```
1  int *stackref() {
2    int val;
3
4    return &val;
5  }
```

- referencing local variables
- the function above returns a pointer to a local variable on the stack and then **pops** its stack frame

  --> **p** still points to a valid memory address, it no longer points to a valid variable

H5 ¶9.11.9 Referencing Data in Free Heap Blocks

```
int *heapref(int n, int m) {
  int i;
  int *x, *y;

  x = (int *)Malloc(n * sizeof(int));

  // other calls to malloc and free

  free(x);

  y = (int *)Malloc(m * sizeof(int));
  for (i = 0; i < m; i++)
    y[i] = x[i]++;  // x[i] is a word in a free block

  return y;
}
```

- reference data in heap blocks that have already been freed
  - array **x** might be part of some other allocated heap block and may have been overwritten

H5 ¶9.11.10 Introducing Memory Leaks

```
void leak(int n) {
  int *x = (int *)Malloc(n * sizeof(int));

  return; // x is garbage at this point
}
```

- if **leak** is called frequently, then the heap will gradually fill up with garbage --> may consume the entire virtual address space
- serious for programs such as daemons and servers