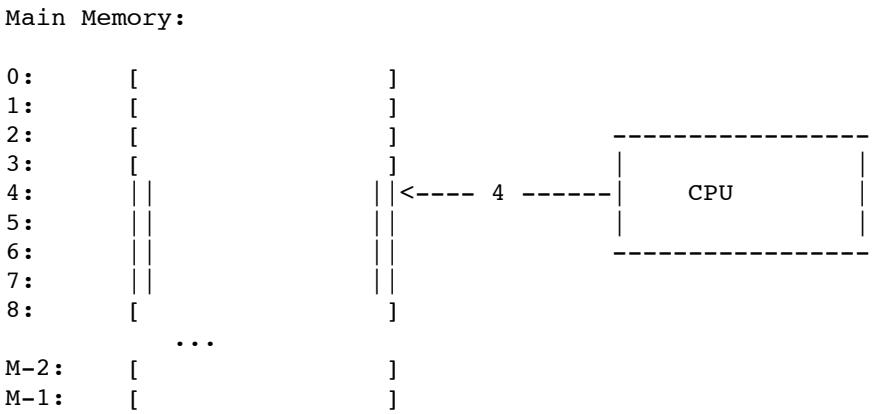
Virtual Memory

Virtual memory is one of the most complicated yet most invisible aspects of a modern computer system. *Virutal memory* is an abstraction of main memory that introduces three key capabilities:

1. It makes efficient use of main memory by treating it as a cache for an address space stored on disk, keeping only the active areas in main memory.
2. It provides each process on a system with a uniform view of memory
3. Processes cannot corrupt the address space of another process

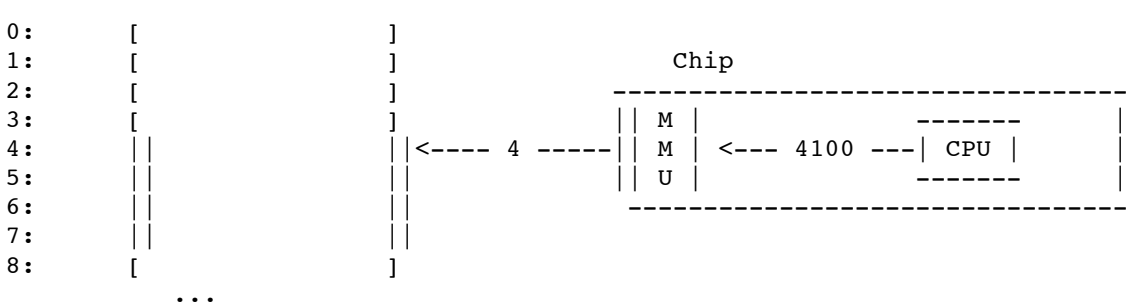
Physical vs. Virtual Addresses

Recall that a computer system's main memory is organized as an array of M contiguous byte-sized cells. Each byte has a unique *physical address* ranging from 0 to M-1. When a program running on a computer system accesses a memory location using the physical address, we call this ***physical addressing***



Memory organization for system using physical addressing with 4-byte words

A system with ***virtual addressing*** runs programs that access memory using *virtual addresses* which are converted to physical addresses by a dedicated peice of hardware called the *memory management unit* through a process called *address translation*.



```
M-2:    [                ]
M-1:    [                ]
```

Memory organization for system using virtual addressing with 4-byte words.

Address Spaces

The concept of address spaces is important because it helps separate and define the distinction between data objects (the actual bytes of data) and the attributes of the data (addresses). Much like you can have more than one C pointer variable "point" to the same data, data objects may have multiple independent addresses, each chosen from a different address space. *Each byte* of main memory has a virtual address chosen from the virtual address space and a physical address chose from the physical address space. We will define an address space as a *set of nonnegative integer addresses*:

S= {0,1,2, ... }

If the integers of an address space are consecutive, we call it a *linear address space*

In systems with virtual addressing, the CPU generates a linear address space from an address space of N = 2^n addresses. This is called the *virtual address space*:

Virtual Address Space { 0, 1, 2, ... N - 1 }
The size of an addresses space is characterized by the number of bits needed to represent the largest address. A virtual address space with N = 2^n addresses is called an n-bit address space. 32-bit systems generally support a 32-bit virtual address space, for example.

Example: Table with virtual address entities:

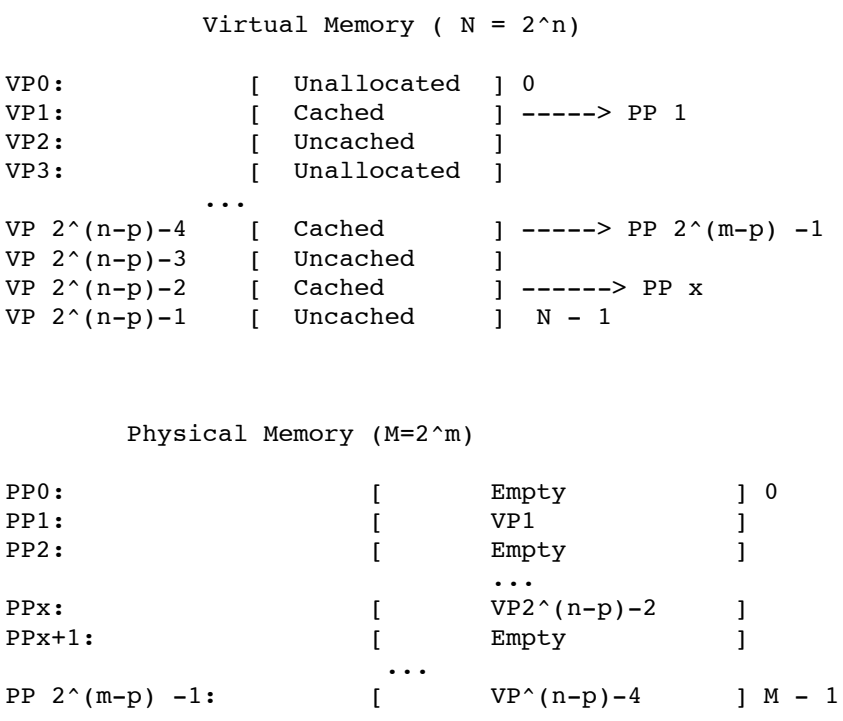
# of virtual address bits	**# of Virtual addresses (N)**	**Largest possible virtual address**
8	2^8=256	2^8-1=255
16	**2^16=64K**	2^16-1 = 64K - 1
32	2^32=4G	**2^32-1 = 4G - 1**
48	**2^48=256T**	2^48-1 = 256T - 1
64	2^64=16E(Exa)	2^64-1 = 16E - 1

Virtual Memory as a Tool for Caching

Virtual memory systems use cacheing as an efficient method of using main memory. This is accomplished by partitining the virtual address space into fixed-sized blocks called *virtual pages (VPs)*. Each virtual page is some P = 2^p bytes in size. The physical pages that a virtual page maps to in main memory is called a *page frame*. At any point in time, a virtual page is in one of three mutually exclusive states:

1. **Unallocated**: Pages that have not been allocated. Unallocated pages do not have any data associated with them and do not occupy and space on disk.

2. **Cached**: Pages that are allocated and are also in physical memory.
3. **Uncached**: Pages that are allocated but are not in physical memory.



Recall that the DRAM which is typical for main memory is about 10 times slower than the SRAM that makes up L1, L2, etc. cache memory however, disk is about 100K slower than DRAM. Cache misses from DRAM incur a large miss penalty since data must be loaded from disk. For this reason, virtual pages are generally very large. On the order of 4KB to 2MB. DRAM-based cacheing is also fully associative and use more sophisticated replacement/eviction policies than what is implemented in the hardware. The replacement algorithms are handled by the operating system.

The Page Table

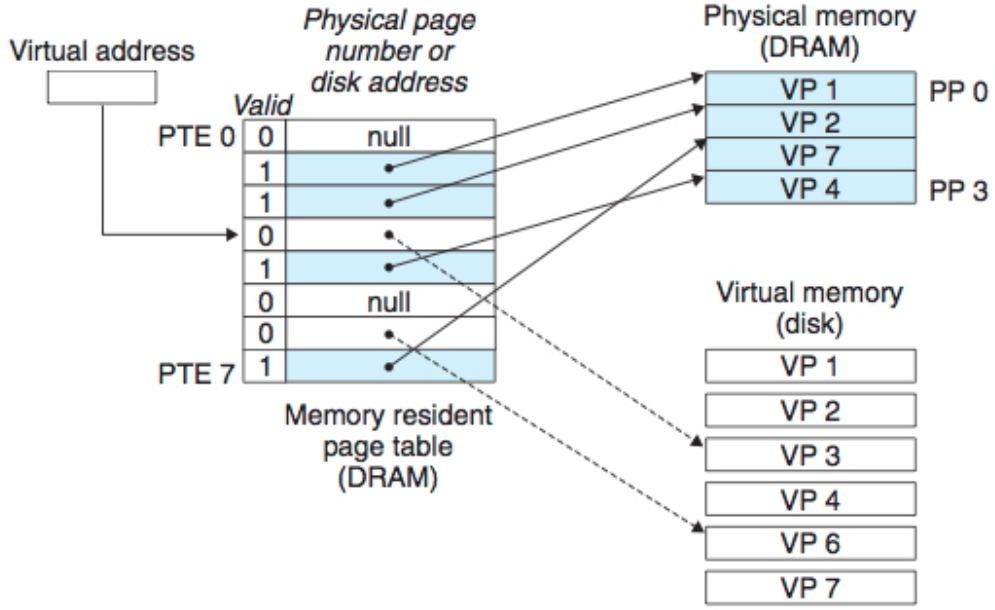
The *page table* is a data structure that is managed by the operating system stored in physical memory and maintains a mapping of virtual pages to physical pages. The MMU reads the page table when it converts a virtual address into a physical address. The page table is an array of *page table entries (PTEs)*. There is one PTE for each page in the virtual address space. Each PTE consists of a valid bit and an n-bit address field that indicates the start of the corresponding physical page in DRAM where the virtual page is cached. If the valid bit is not set then a null address indicates that the virtual page has not been allocated. Otherwise, the address points to the start of the virtual page on disk:

Valid bit	**Address**	**Virtual Page Status**
0	NULL	Unallocated
0	Address A	Uncached, on disk at location A
1	Address A	Cached, in physical memory at location A

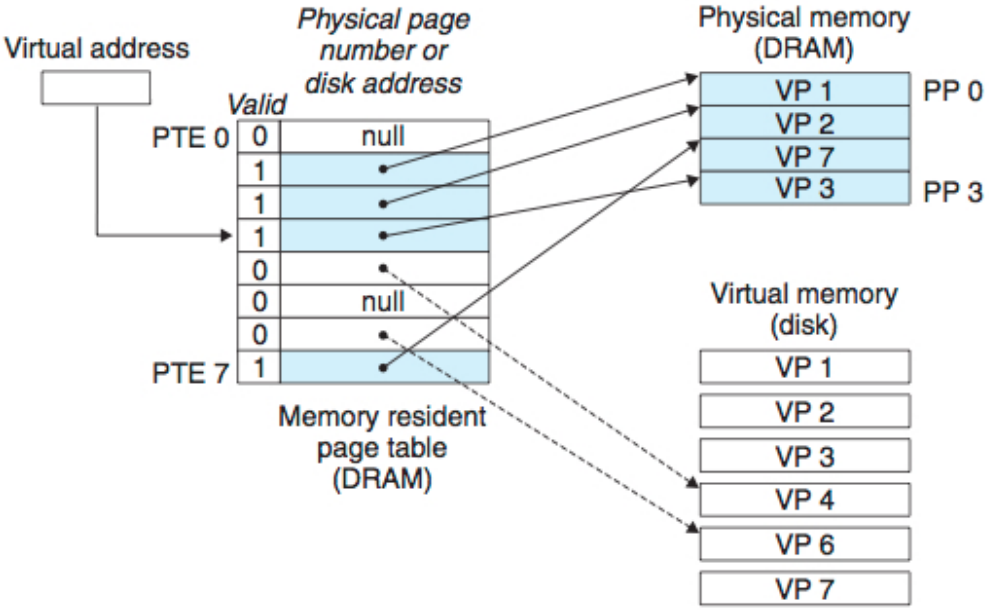
```
PTE0:    [ 0 |     NULL    ]
PTE1:    [ 1 |    PP0       ]
PTE2:    [ 1 |    PP1       ]
PTE3:    [ 0 |    0xX_3     ]
PTE4:    [ 1 |    PP3       ]
PTE5:    [ 0 |    NULL      ]
PTE6:    [ 0 |    0xX_5     ]
PTE7:    [ 1 |    PP2       ]

Physical Memory

PP0:     [        VP1       ]
PP1:     [        VP2       ]
PP2:     [        VP7       ]
PP3:     [        VP4       ]

Disk:

0xX_1
         [        VP1       ]
0xX_2
         [        VP2       ]
0xX_3
         [        VP3       ]
0xX_4
         [        VP4       ]
0xX_5
         [        VP6       ]
0xX_6
         [        VP7       ]
```

When page table is not in memory, it is referred to as a ***page fault***. When there is a page fault, the operating system selects a page to be *paged out* and writes it to disk. Then it selects a page to *page in* and updates the page table entries. This process is referred to as *paging* (or swapping).

Before page fault



After page fault



Example: How many page table entries are needed for the following combinations of virtual address size(n) and page size(P):

1. n=16, P = 4K

   2^16 / 4K = 2^16 / 2^12 = 2^4 = 16 PTEs
2. n=16, P = 8K

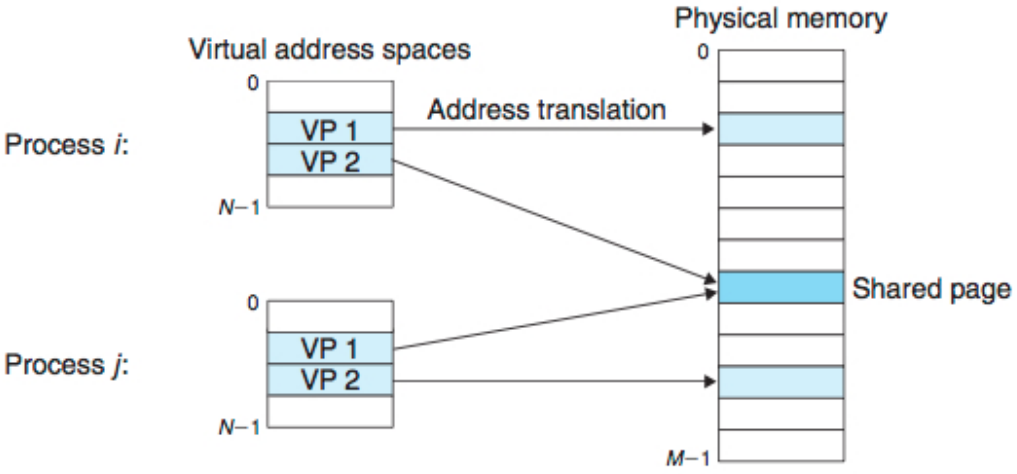
   2^16 / 8K = 2^16 / 2^13 = 2^3 = 8 PTEs
3. n=32, P = 4K

   2^32 / 4K = 2^32 / 2^12 = 2^20 = 1M PTEs
4. n=32 P = 8K

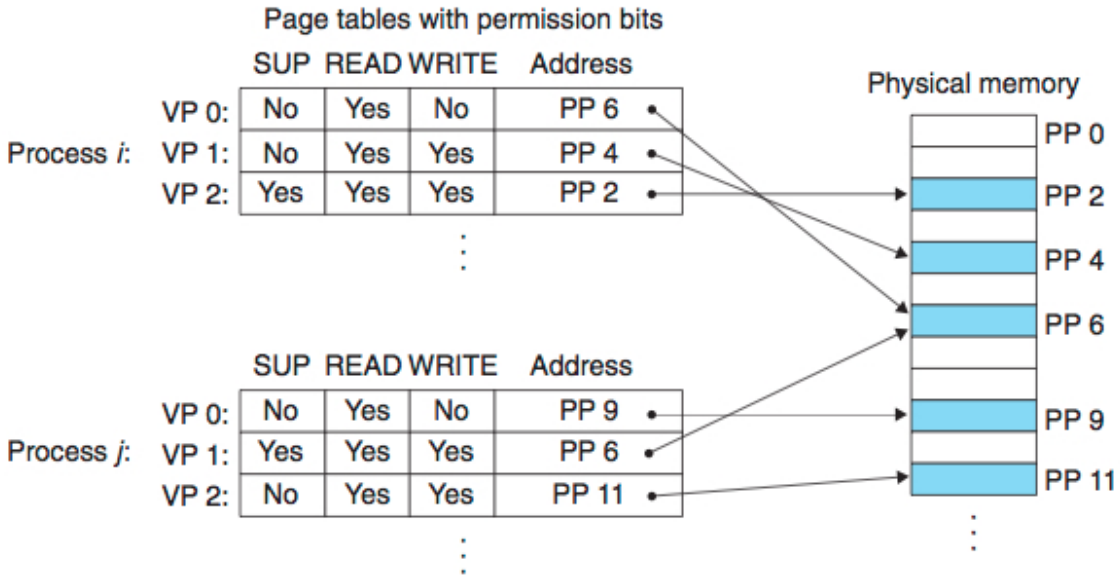
   2^32 / 8K = 2^32 / 2^13 = 2^19 = 512K PTEs

Virtual Memory as Tool for Memory Management and Protection

SO far, we have assumed a single page table maps a single virtual address space to the physical address space. In reality, the operating system provides each process with its own virtual address space and thus its own page table. This can be quite useful if, for example, two processes both make use of the same C libraries (i.e. printf). printf only need have one physical location but can be referenced through several virtual address spaces.



Virtual memory can also be used to add access protection to memory locations. Remember how one approach to thrawting a buffer overflow attack was to copy a "canary" value from a protected memory location. This can be implemented by the operating system by taking advantage of the protection level provided by the virtual memory system. TO support protection, we need to add permission bits to the page table entries.



Remember the short story as to why you get a segmentation fault in C programming is basically, you are accessing a memory location you don't have access to or is otherwise invalid. The long story is demonstrated by page-level memory protection provided by virtual memory.

Dynamic Memory Allocation

Creating and deleting areas within virtual memory requires the programmer to be aware of the virtual address space. **Dynamic Memory Allocators** are a convenient method for allowing programmers to request or release additional virtual memory at run time. Dynamic memory allocators maintain the *heap* of a process's virtual memory. The operating system maintains a special variable called ***brk*** ("break") for each process that points to the top of the heap. Dynamic memory allocators maintain a list of heterogenous blocks within the heap, with each block flagged as either *allocated* or *free*. Each block is a contiguous chunk of virtual memory.There are two types of dynamic memory allocators:

- **Explicit Allocators**: are dynamic memory allocators that require the process to explicitly free any allocated blocks. Dynamic memory allocation in C is an example of explicit allocation. Memory allocated by a call to *malloc* will remain allocated until either the program exits or there is a call to *free*.
- **Implicit Allocators**: are dynamic memory allocators that automatically detect when an allocated block is no longer in use by the program and frees the block. This process is known as ***garbage collection***. Objects created in languages like Java or C# are examples of implicit allocation. There are no deconstructors to call, the garbage collector will automatically detect when a block is no longer in use and dealloc the memory.

In C, there are a set of functions that can help in allocated and delloacted memory:

- **void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)**: performs *memory mapping* which is the process of associating an area of virtual memory with an physical location (data object) on disk. It asks the operating system to create a new virtual memory area and map it to a contiguous chunk of the data object specified by the file descriptor fd. The function returns a pointer to the virtual address where the mapped memory chunk begins.Using mmap does not allocate from the heap, it actual tells the operating system to *create* a new area in the virtual memory space

```
void *buffer = mmap(NULL,   /// tell OS to choose any address

                    size,   //  size in bytes of space to allocate

                    PROT_READ | PROT_WRITE, // flags space as read/write

                    MAP_ANON | MAP_PRIVATE, //  anonymous/non-file backed

                    0, // no FD

                    0) // offset of 0
```

- **int munmap(void *start, size_t length)**: Deletes the area starting at the given virtual address, consisting of the next length bytes.

  Question: What do you think happens if you try to call munmap on an address twice?

  You will get a segmentation fault. Remember how segmentation faults and page-level protection work. Presumable when the virtual address area is UN-memory-mapped, the operating system invalidates the page table entry that indicates the address is a memory mapped space.

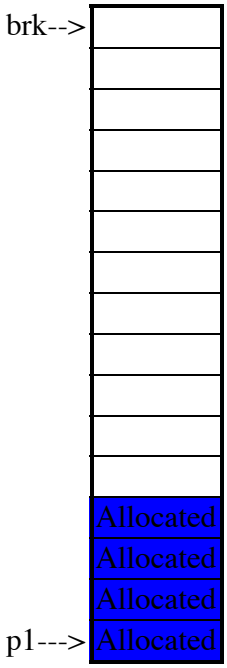
- **void *sbrk(intptr_t incr)**: the sbrk function grows or shrinks the *heap* by adding incr to the kernel's *brk* pointer. If it is successful, it returns the old value of brk, otherwise it returns -1. While calling sbrk with negative numbers is legal, the return value may not be meaningful as the old address it returns is abs(incr) bytes *past* the top of the heap.

  In general, dynamic memory allocators make use of calls to sbrk() in order to allocate memory from the heap. This is now changing as more sophisticated implementations exist that may additionally may use mmap/munmap for very large allocations but these can run into portability issues as there are operating system specific conventions to using mmap/munmap with anonymous mappings. Some modern allocators like malloc for OS X use only mmap/munmap.

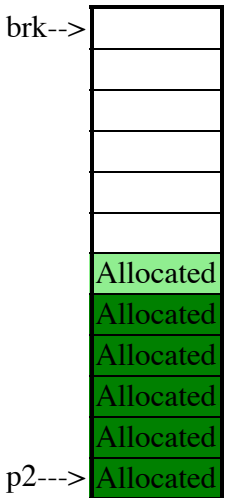
Example:: Ignoring additional indexing bytes, we can trace what happens when we make the following series of malloc and free calls. We will assume our allocator grows the heap by 64-bytes on demand and maintains pointers on a *double word* boundary.

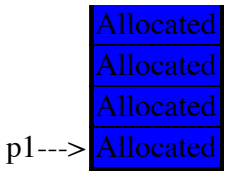
```
int *p1, *p2, *p3, *p4;

p1 = (int *)malloc(4* sizeof(int));
p2 = (int *)malloc(5*sizeof(int));
p3 = (int *)malloc(6*sizeof(int));
free(p2);
p4 = (int *)malloc(2 * sizeof(int));
```

1. When the first malloc call is processed, our allocator will need to grow the heap and use the space to generate our initial free block. sbrk(64) will grow the heap by 64 bytes, giving us enough room for 16 32-bit integers. the allocator will then reserve the first 4 words and return a pointer to the first word in the block:

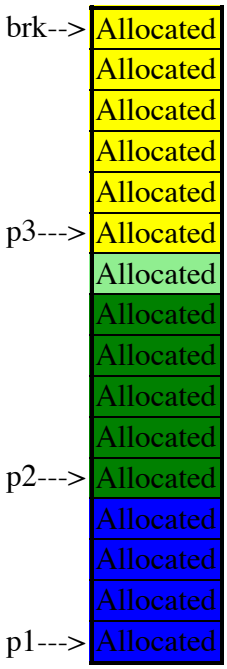


2. a five-word block is requested through malloc. the allocator reserves the next six words in the free block. the extra word is allocated in order to preserve the double word boundary:

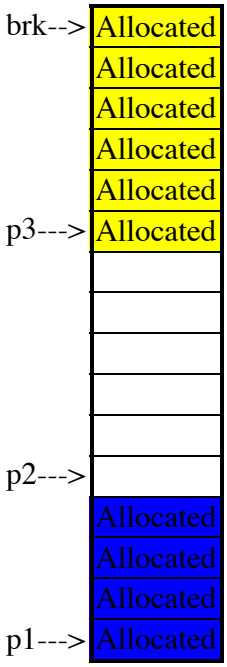




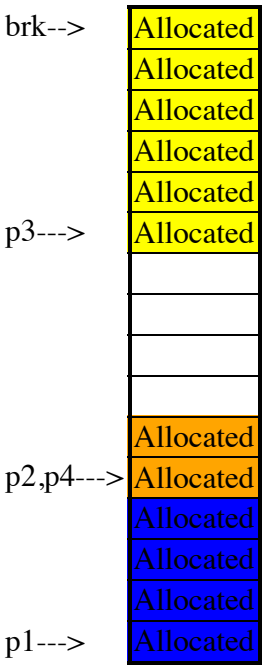
3. A six word block is requested by malloc, the allocator returns a pointer to the next 6 words in the free block. notice that we are now at the top of the heap. another malloc call would force another call to sbrk():



4. free(p2) results in the 6 word block where p2 points to be deallocated. It is up to the programmer not to use p2 anymore.



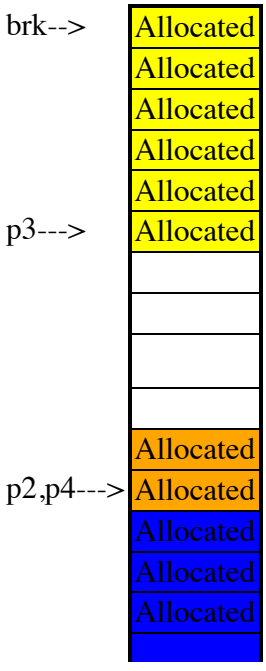
5. malloc receives a request for a two word block. the allocator maintains a list of blocks that were previously allocated but are now free, so it can allocate space for p4 without another call to sbrk():

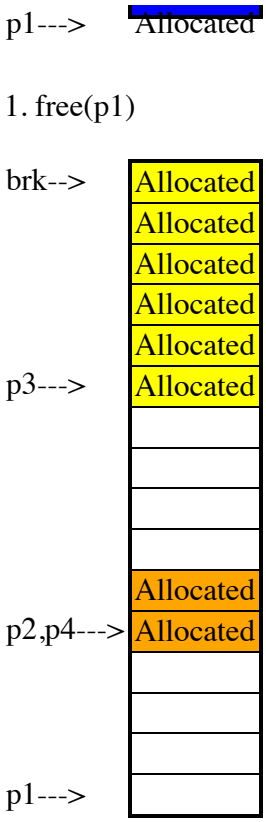


Balancing Throughput and Memory Utilization

A good memory allocator should be fast but it should also make good use of the space it has available. We could blindy allocate more memory from the heap as needed and ignore space that has been deallocated. This might improve the throughput of our allocator, but at the expense of poor heap utilization. Memory, even virtual memory, is a scarce resource in the system. Memory allocators must strike a good balance between *throughput* and *heap utilization*. When there is poor heap utilization ***fragmentation*** may occur. fragmentation is when otherwise unused memory is not available to satisfy allocation requests. There are two types of framentation:

1. **Internal fragmentation**: occurs when an allocated block is larger than the payload. For example, when allocated block size is increased beyond the payload size in order to adhere to a word-boundary.
2. **External fragmentation**: occurs when there is enough aggregate free memory to satisfy an allocation request, but no single free block is large enough to handle the request:





If we try:

```
int *p5 = (int *)malloc(6*sizeof(int));
```

Our allocator would have to ask for more heap space using sbrk() even though there is enough memory in the two free blocks to complete the allocation request.

Internal fragmentation is easy to identify and quantify: it simply the sum of the differences between the sizes of the allocated blocks and their payloads. External fragmentation is harder to quantify since it depends on all past, currt AND future requests

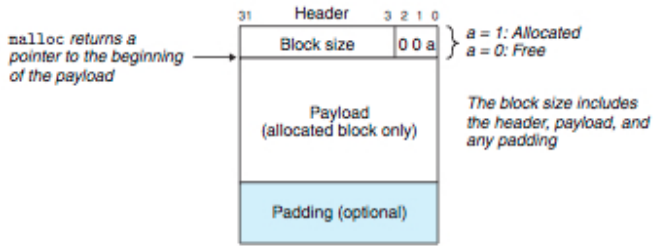
Example: After k requests, each free block is exactly 4 words in size. Does this heap suffer from external fragmentation?

It depends! If all future requests are for blocks of size 4, then no. if on the other hand one or more future requests asks for a block larger than size 4, then it does!
There are four primary things to consider in order to implement an allocator that strikes a pragmatic balance between throughput and between heap utilization:

1. *free block organization:*:How do we keep track of free blocks? What data structures do we use, what operations (search, add, remove, etc). do we supports?
2. *placement:* How do we choose an appropriate free block in which to place a newly allocated block
3. *Splitting*: After an allocated block is placed in a free block, what do we do with the remainder of the free block?
4. *Coalescing:* what do we do with blocks when they are freed?

A simple data structure for block organization is the *implicit free list*. With an implicit free list the free block

includes a header, the payload and optional padding at the end:



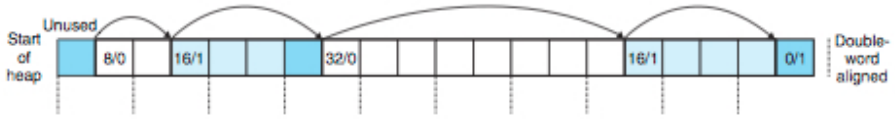
The header encodes the block size as well as noting if the block is allocated or not. We can then traverse between heap blocks by using the block size as the distance to the next block. On a system with double-word boundaries, the lowest 3-bits of the header will always be zero since block size is a multiple of 8. We can encode our free/allocated bit using the least-significant bit:

Example:What is the value of the header for a block of size 24 bytes if the block has been allocated

```
0x0000000018 (block size = 24) | 0x00000001 (set LSB to 1 to indicate allocated)


= 0x00000019
```

Example: What is the value of the header for a block size of 40 (0x28) if the block is free?

```
0x0 | 0x00000028 =  0x00000028
```



Notice that the word-boundary requirement and data structure impose a minimum block size for the allocator:
Example: For the following malloc calls, determine the block size and the block header given that allocations are padded to ensure double word boundaries (8 bytes) and the allocator uses an implicit free list.

```
malloc(1)

  Block Size: 8 bytes
  Header:     0x00000008 | 0x01 = 0x00000009


malloc(5)

      Block Size: 16 bytes
      Header:  0x00000010 | 0x01 = 0x00000011


malloc(12)

      Block Size: 16 bytes
      Header:  0x00000010 | 0x01 = 0x00000011
```

```
malloc(13)

     Block Size: 24 bytes
      Header: 0x00000018 | 0x01 = 0x00000019
```

Common placement policies:

- **First Fit:**: the implicit free list is scanned from the beginning and the first block that is large enough for the payload is selected.
- **Next Fit:** Next fit is similar to first fit, but it begins the traversal where the last search left off.
- **Best Fit:** Best fit scans the entire implicit free list for the smallest free block that can handle the payload.

Splitting

Instead of allocating an entrire free block, an allocator may choose to split a free block in order to help reduce internal fragmentation. When a free block is split, the first block becomes the allocated block and the remainder becomes a new free block:

```
Before:

Heap [    ][8/1 ]|[xxxx][32/0]|[    ][    ]|[    ][    ]|[    ][    ]|[    ][8/1 ]|[xxxx][0/1 ]

After:
Heap [    ][8/1 ]|[xxxx][16/1|[xxxx][xxxx]|[xxxx][16/0]|[    ][    ]|[    ][8/1 ]|[xxxx][0/1 ]
```

Coalescing Free Blocks *Coalescing* is when an allocator merges two adjacent free blocks into one. Coalescing can help combat external fragmentation. One implementation makes use of a footer at the end of the data payload+padding. The footer is a replica of the header.

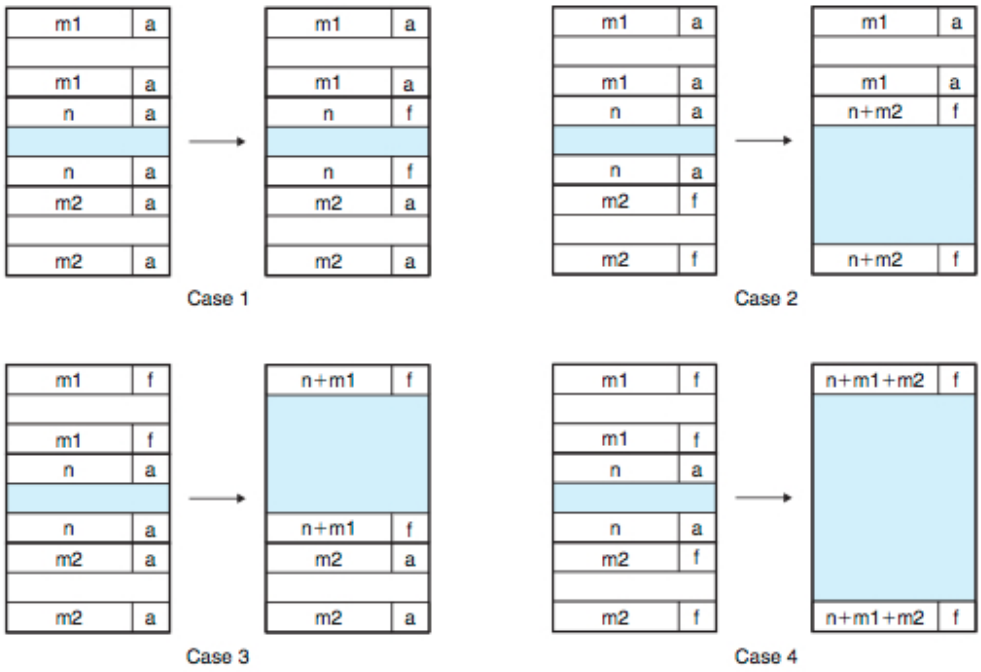


Figure 9.40 **Coalescing with boundary tags.** Case 1: prev and next allocated. Case 2: prev allocated, next free. Case 3: prev free, next allocated. Case 4: next and prev free.

Example:What is the minimum block size for each of the following combinations of alignment requirements and block formats. Assume an implicit free list, zero-sized payloads are not allowed and headers and footers are stored in four byte words.

ALIGNMENT	ALLOCATED BLOCK	FREE BLOCK	MIN. BLK SIZE
Single word	Header+footer	Header+footer	12 bytes
Single word	Header only	Header+footer	8 bytes
Double word	Header+footer	Header+footer	16 bytes
Double word	Header only	Header+footer	8 bytes