Managing free space

With the goal of keeping several hundred or more processes in the physical memory – allocating new area for an incoming process is frequent so is getting allocated area freed for the terminating processes. Staring with a solid block of contiguous memory space we could visualize that external fragmentation is inevitable with the progress of time and we gert chopped up available address space into smaller pieces. This smaller pieces are of no use until we move them to have a larger free space to accommodate any challenging need. Let us reiterate some of the basic issues and assumptions for the foregoing discussions.

- Free space management is a fundamental task of the OS
- Managing free space is much easier for fixed size slots (paged) but gets complicated for variable sizes (segmented)
- One of the main reasons for free space management is to reduce external fragmentation
- internal fragmentation can be largely avoided by allocationg small space that accommodates the present need and a provision to allocate more; if necessary.
- Discussion would be primarily based on *malloc* and *free* calls

Basic Mechanism used

For free space management irrespective of the broader strategy low level mechanisms, like splitting and coalescing, used are same.

Consider a tiny heap

Free	Used	Free
10	20	30

The corresponding free list might look like below

```
head -> (a:0; l:10) ->(a:20;l=10) -> X
```

Allocation request of more than 10 bytes would fail. On the other hand a request for 3 bytes might be solved by allowing 3 bytes from the 2nd free chunk (the malloc(3) call would return a pointer to 20) and the corresponding change in free list would be

```
head -> (a:0; l:10) ->(a:23;l=7) -> X
```

So, the free list would be more or less intact after the *split*; 3 bytes less in the 2nd free chunk.

Basic Mechanism used ... contd.

To understand the reverse of split i.e., the coalescing operation consider a call to free(10) on the original heap on the left and the free space (on the right) is shown below:

Consider a tiny heap

Free	Used	Free
10	20	30

Free	Free	Free
10	20	30

The corresponding free list might look like below

```
head -> (a:0; l:10) ->(a:20;l=10) -> (a:30;l=10)-> X
```

Allocation request of more than 10 bytes would fail again though we technically have 30 bytes of contiguous space. To avoid this the allocator *coalesces* the free space with the adjacent (if any) free space. Thus, after coalescing, in our case, the list would be a singular piece of free space. Thus coalescing does the recerse of split.

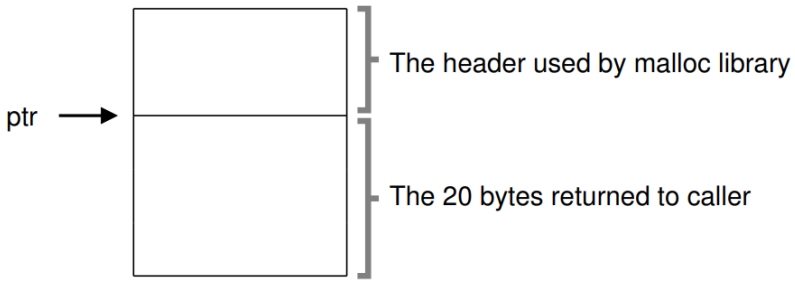
```
head -> (a:0; l:30) -> X
```

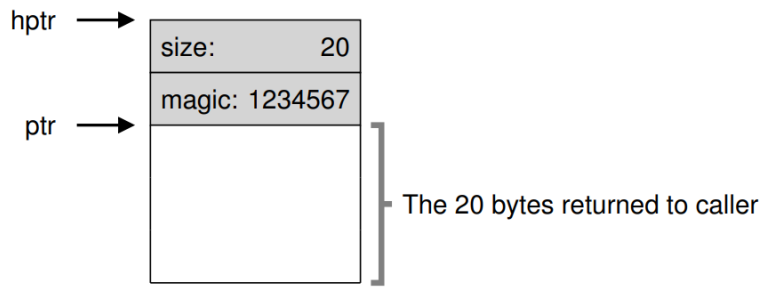
Keeping track of the used space

Note that the *free* call takes *void *ptr* i.e., it does not take the size parameter. Then how does the malloc library know the size of the memory being freed. To know the size extra piece of informatin (a header) is added during the allocation in front of the allocated chunk. For faster deallocation more pointers, a magic number or some extra information may also be added to this header. The structure might be

```
typedef struct __header_t {
int size; int magic; } header_t;
```

Thus for a call to malloc(20) the scenario is depicted below. Figures show an allocated region with the header and the contents of the header.





Keeping track of the used space ...contd.

When the user calls free(ptr) the library is able to find out the beginning of the header

```
void free(void *ptr) {
header_t *hptr = (void *)ptr - sizeof(header_t);
}
```

After getting the pointer to the header malloc library can easily determine whether the magic number matches the expected value as a sanity check $(assert(hptr->magic == 1234567))$ and calculate the total size of the newly-freed region by adding the size of the header to size of the region. Thus, when a user requests N bytes of memory, the library searches for a free chunk of size N plus the size of the header.

Free list within the free space

Assume we have a 4K-byte heap. To manage this as a free list, we first have to initialize said list; initially, the list should have one entry, of size 4096 (minus the header size). The node of the list could be

```
typedef struct __node_t {
int size;
struct __node_t *next;
} node_t;
```

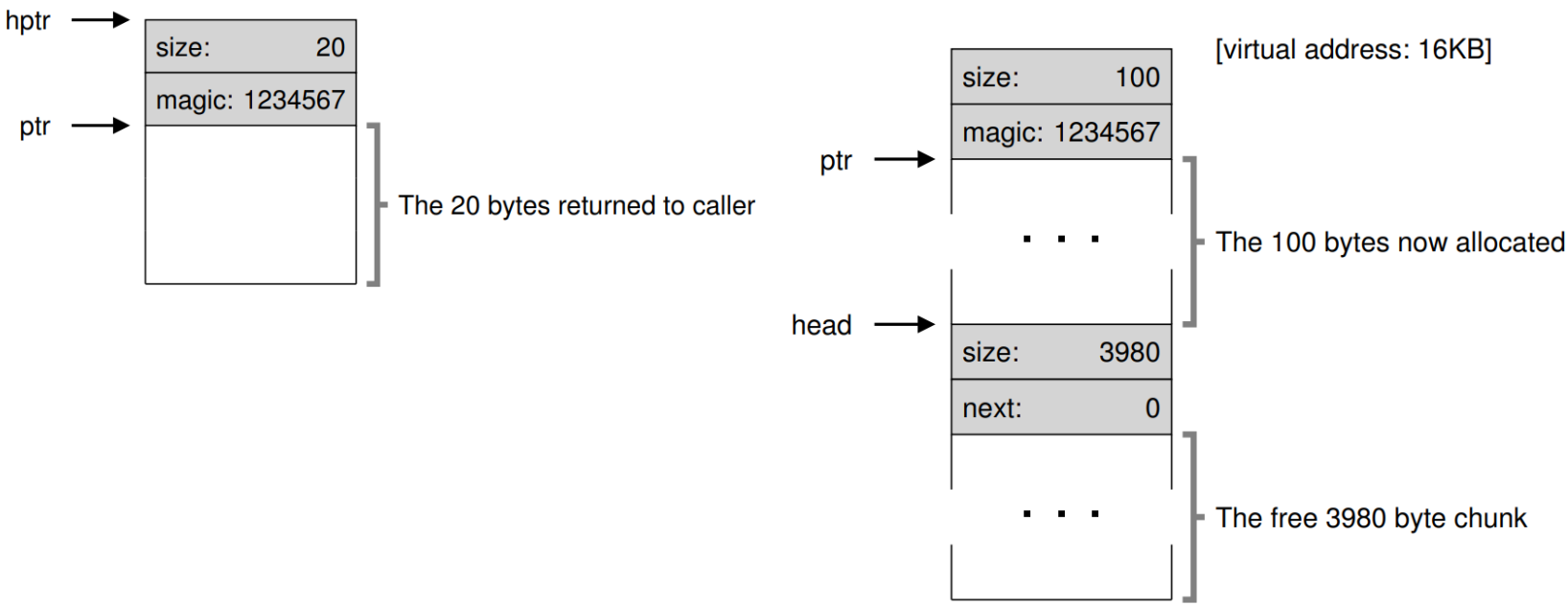
Now let's look at some code that initializes the heap and puts the first element of the free list inside that space. We are assuming that the heap is built within some free space acquired via the system call mmap().

```
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL; // mmap() returns a pointer to a  free space
```

Executing the code we get a list is that it has a single entry of size 4088.

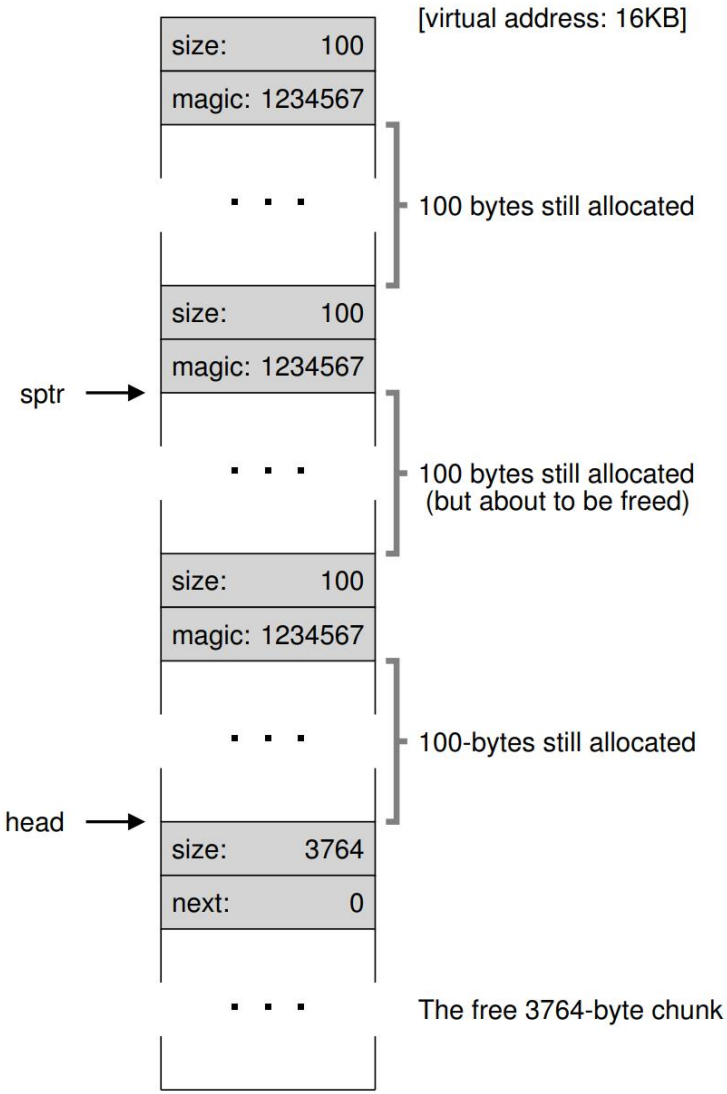
Keeping track of the used space

Visually, the heap looks like the figure on the left. For an allocation request of 100 bytes To service this request, , the chunk will be split into two: Assuming an 8-byte header the space in the heap now looks like what you see on the right



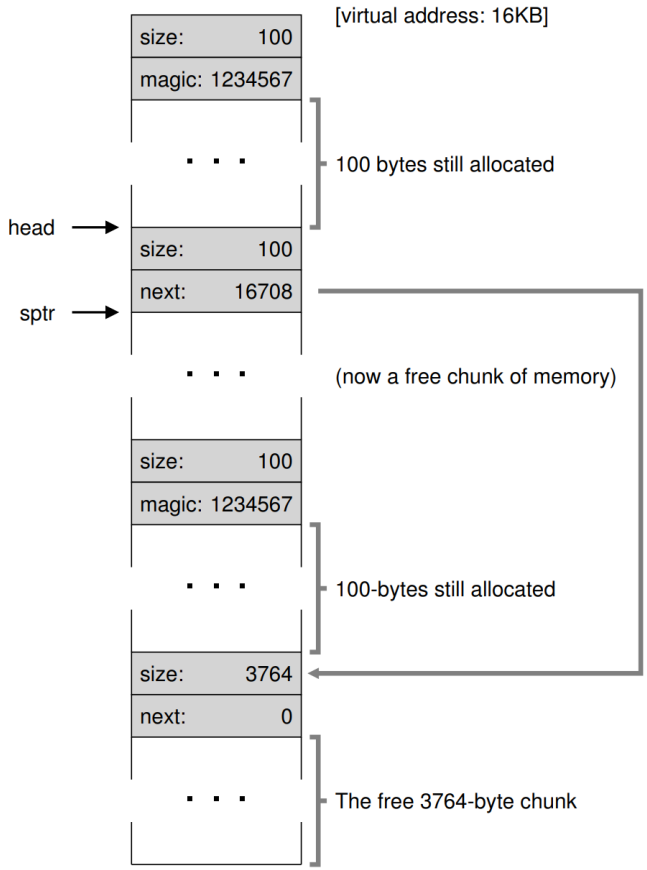
Keeping track of the used space

Visualisation of the heap after 3 allocations, each 100 bytes (or 108 to be precise)



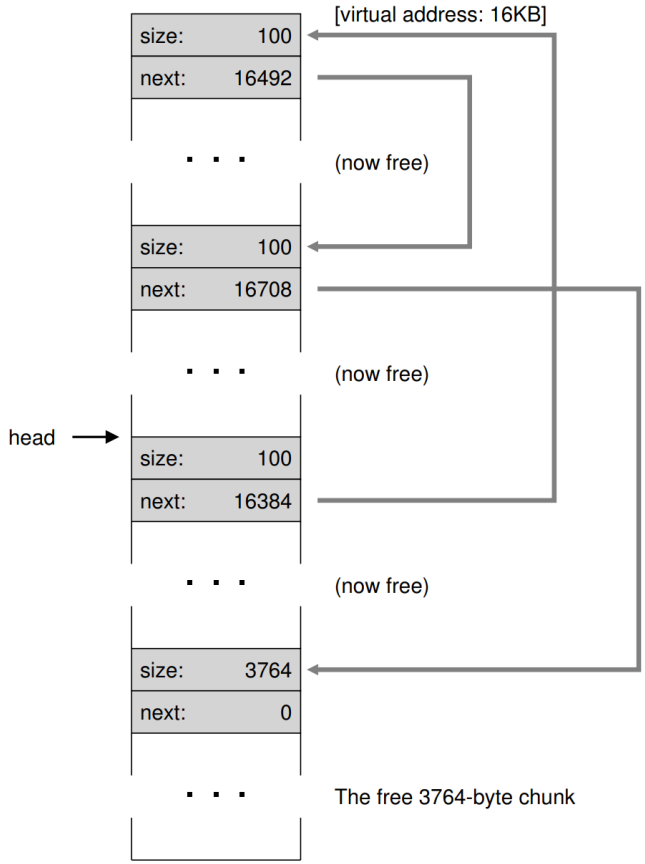
Keeping track of the used space

What happens when we get back some memory via free() call? Assume that the application returns the middle chunk of allocated memory, by calling free(16500) (the value 16500 is arrived upon by adding the start of the memory region, 16384, to the 108 of the previous chunk and the 8 bytes of the header for this chunk). This value is shown in the previous diagram by the pointer sptr.



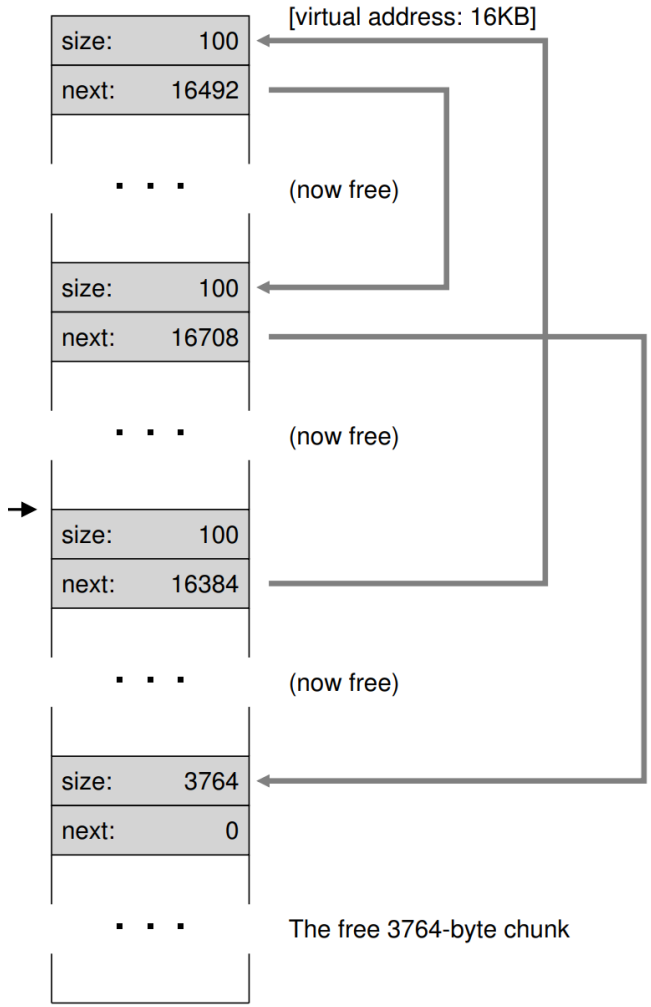
Keeping track of the used space

The library immediately figures out the size of the free region, and then adds the free chunk back onto the free list. Assuming we insert at the head of the free list, the space now looks like the figure. And now we have a list that starts with a small free chunk (100 bytes, pointed to by the head of the list) and a large free chunk (3764 bytes).



Keeping track of the used space

let's assume now that the last two in-use chunks are freed. Without coalescing, you might end up with a free list that is highly fragmented (see Figure).



Heap – growing if it is needed

If we need more space and the heap runs out of space. In many cases that's accepted and we return a null pointer to indicate failure.
Traditional allocators starts with small heap space and request for more memory from the OS. This means system call like sbrk (in most UNIX system) is used to grow the heap and allocates new chunk from there. To service *sbrk* reuest the system

- Find free physical pages
- Maps them to the address space of the requesting process; and
- returns the value of the end of the new heap; so a
- larger heap is available and the request served.

Allocation strategies

The primary goal of the allocator is to reduce fragmentation by selecting appropriate free memory as quickly as possible. Considering the dynamic and uncontrolled nature of the memory space requirements there is perhaps no best policy and better depending on siuations.

- Best fit
- Worst fit
- First fit
- Next fit

Allocation strategies

Best fit/smallest fit is a natural selection for the space which is exactly same as the requested size or slightly more . One scan through the free list would be enough to find out the appropriate chunk.

- This reduces the wasted space; however
- it might be time consuming to find out the fittest candidate.

Worst fit is opposite to the best fit

- Finds out the largest free chunk – allocate the requested, small, amount and keep the (slightly) reduced chunk in the free list
- it might be time consuming to find out the fittest candidate.

First fit – scan through the free list and choose the first chunk to allocate that fits; it reduces the exhaustive search associated with the best/worst fit policies

- Keep the remaining after allocation in the free list
- The free list will have many small objects in the beginning. Thus a proper ordering is a necessity. One approach is to order the list by the address of the free space for the ease of coalescing and reduced fragmentation.

Allocation strategies ... contd.

Next fit – kees an extra pointer to point where the last access is made and scan begins from that point instead of the first. The idea was to allocate evenly instead of splintering of the beginning of the list.

- exhaustive search is avoided
- performance would be similar to first fit

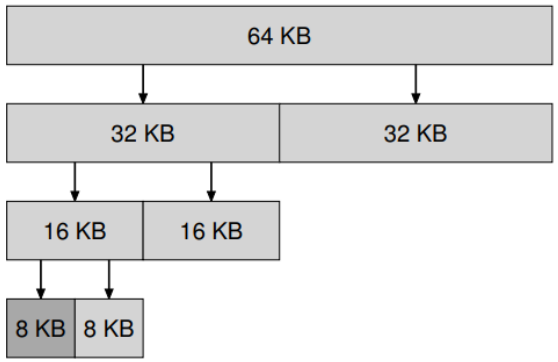
There are more strategies based on other aspects of interest. For example, the **segregated lists**. The basic idea is simple: if a particular application has one (or a few) popular-sized request that it makes, keep a separate list just to manage objects of that size; all other requests are forwarded to a more general memory allocator. By having a chunk of memory dedicated for one particular size of requests, fragmentation is much less of a concern; moreover, allocation and free requests can be served quite quickly when they are of the right size, as no complicated search of a list is required. Determining how much memory should one dedicate to the pool of memory that serves specialized requests of a given size, as opposed to the general pool?

Allocation strategies ... ease of coalescing

Whatever we do we cannot fully prevent external fragmentation and compaction in some form like coalescing will be important. The **Buddy allocation** was designed to make coalescing simple.

- free memory is f thought of as one big space of size 2^N .
- the search for free space recursively divides free space by two until a block big enough to accommodate the request is found

Here is an example of a 64KB free space getting divided in the search for a 7KB block. Buddy may introduce internal fragmentation. However, coalescing is simple. For example, the 8KB block when returned can be coalesced to 16 KB and recursively to 64KB (initial position; if all other blocks are free)



Summary : Managing free space

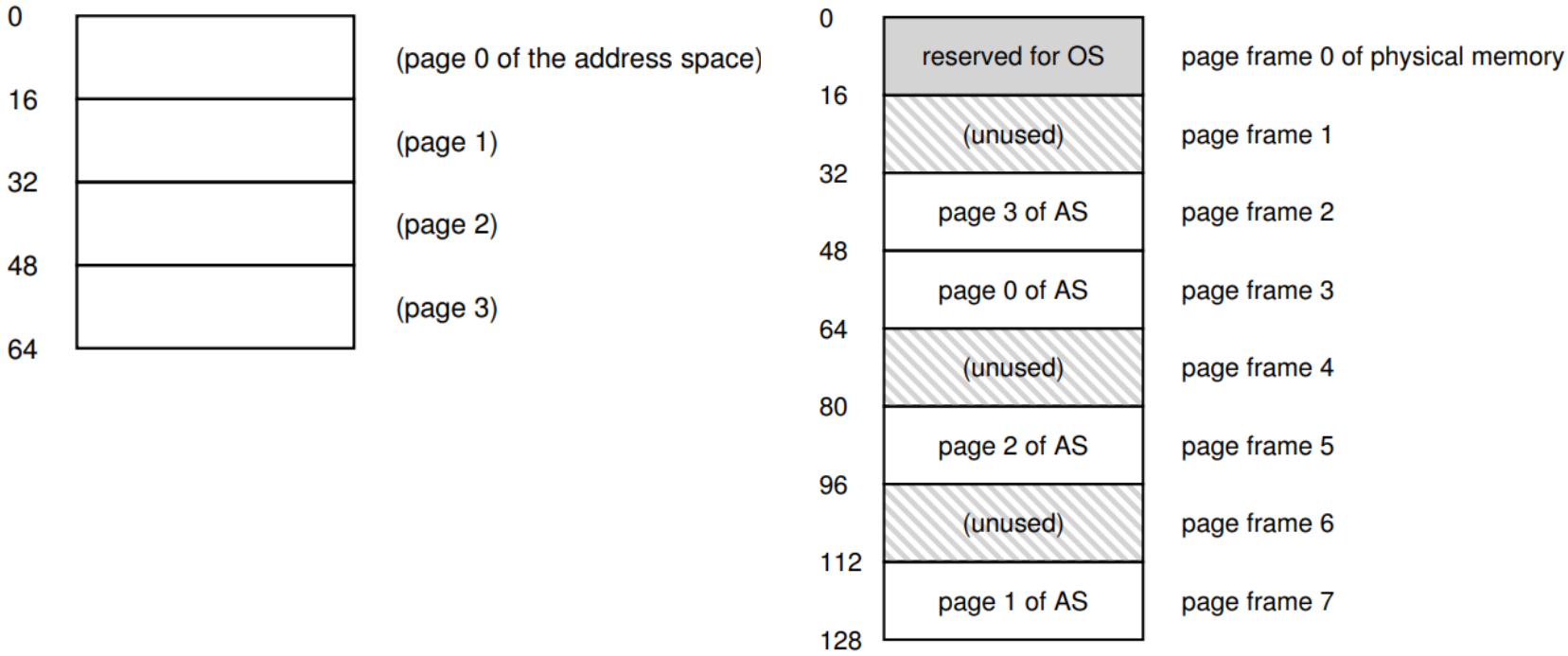
We've see a couple of the most rudimentary forms of memory allocators.

- Such allocators exist everywhere, linked into every C program you write, as well as in the underlying OS which is managing memory for its own data structures.
- There are many trade-offs to be made in building such a system
- Making a fast, space-efficient, scalable allocator that works well for a broad range of workloads remains an on-going challenge in modern computer systems.

Paging

Conceptually Segmentation solves many problems to virtualize memory – however the varying size of the segments leads to complicated memory management and fragmentation.

The idea of *paging* dates back to early systems where instead of separate segments of variying sizes we rely on dividing the address space into small fixed sized allocation units, called pages (or page frame). A 16-byte 4-page address space and corresponding mapping to 128 bytes 8-page frames is shown here



Paging: ADVANTAGES

Paging, has a number of advantages over our previous approaches.

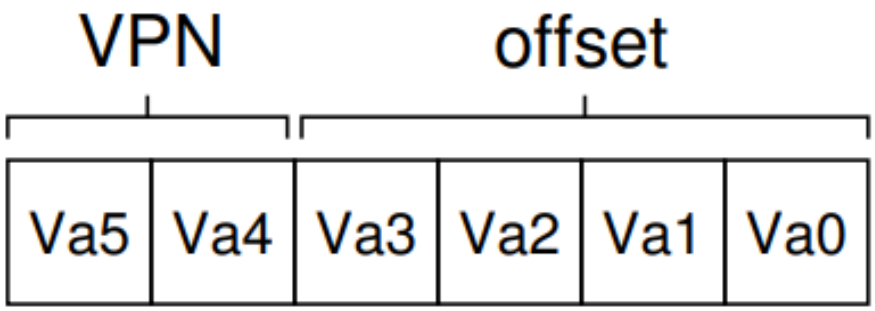
- Flexibility: we won't, for example, have to make assumptions about how the heap and stack grow and how they are used.
- Simplicity of free-space management that paging affords. For example, when the OS wishes to place our tiny 64-byte address space from above into our 8-page physical memory, it simply finds four free pages; perhaps the OS keeps a free list of all free pages for this, and just grabs the first four free pages off of this list.

To record where each virtual page of the address space is placed in physical memory, the operating system keeps a per-process data structure known as a **page table**. The major role of the page table is to store address translations for each of the virtual pages of the address space thus letting us know where in physical memory they live (VP 0 $\rightarrow$ PF 3; VP $\rightarrow$PF7; etc.)

Page Tables

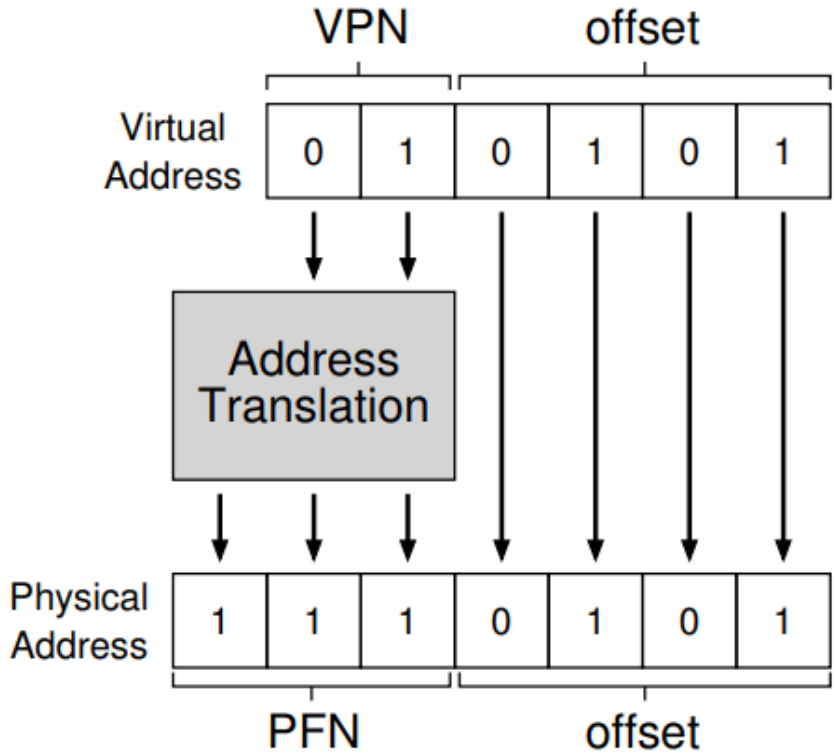
Note that the page-table is a per-process data structure (excdept the inverted page table). If another process were to run, the OS would have to manage a different page table for it, as its virtual pages obviously map to different physical pages.

Address Translation : Let us take the 64 address space (4-pages of 16 bytes each) as we have presented in the beginning. Length of an address would be 6-bits VA(0 to 5). We need two higher bits (VA5 and VA4) to represent the page and rest 4-bits (VA3, VA2, VA1 and VA0) for the offset within that page (see figure).



Page Tables ... contd.

Suppose we need translation of address 21. Note that it is in page 1 (VPN $= 1$) at an offset of 5 (see figure). The translation mechanism should be able to choose the correct physical page frame number (PFM) consulting the corresponding page-table entry (note that page VPN 1 is mapped to PFN 7). This precisely the role to be played by the page-table (PT). In its simplest form the PT could be an array and here index 1 would store 7.



Page Tables ... where to store.

For keep fragmentation at bay we go for more pages – so we have big or very big page table to deal with. QUestion is where to keep it as it requires to be in main-memory for faster translation but main memory is a very important resource. Note,

- For a typical 32- bit machine with 4 KB page; we have
- 2^{20} pages. That is 20 bit of VPN and 12 bit offset. Now,
- with 4-bytes per page table entry (PTE) the size of the PT would be 4-MB for each process. With 100 processes running;
- there would be 400 MB of page table occupying a major portion of main memory; and thus not acceptable.

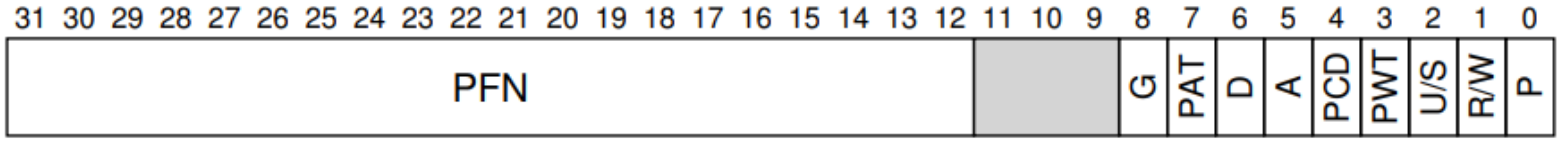
Page tables are so big we do not provide any special on-chip cache in the MMU to hold the PT of the currently running process.
Instead we save the page table for each process elsewhere.

Page Table Entry (PTE)

Primary purpose of the page table is to get the PFN from VPN. So, it can be any data structure; might be an array as well where VPN indexes to PTE to get PFN. However, in real-life PTE we find many bits to implement error checking, paging strategies, cache control and other administrative actions. Some of the common bits are

- Vaild (V) – indicates if the translation is valid or not
- Dirty (D) – tracking wheather the page is changed (written into) once it is brought to the memory.
- Present (P) – to know if the page is present in main memory or in disk
- Referenced (R) – A page has been accessed or not

Figure shows the contents of the PTE for X86-32 bit system. Note that the bits G, PAT, PCT, and PWT are for cache control. The U/S bits determines user access to the page while R/W allows writing to this page.



Paging is a slow process

Consider reading a memory location at virtual address 21 (like the previous example) of our 64-byte system with four 16-bytes pages. Note that the corresponding physical address is 117 (PFN 7 + offset of 5 = 7 x 16 + 5= 117). For translation locating the page table for the current process is done through PTBR (The page table base register – holds the physical base address of the page table for the current process). The PTE (or page table entry) address arithmetic is shown below.

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
[Note that in our case  VPN_MASK is  0x110000 and the SHIFT is 4]
Offset does not require any translation and can be computed as
offset = VirtualAddress & OFFSET_MASK
and the physical address  can be extracted
from the PFN as PhysAddr = (PFN << SHIFT) | offset
```

Steps to get PA from VA in paging for memory access

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
 // Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))
 // Fetch the PTE
 PTE = AccessMemory(PTEAddr)
 // Check if process can access the page
 if (PTE.Valid == False)
 RaiseException(SEGMENTATION_FAULT)
 else if (CanAccess(PTE.ProtectBits) == False)
 RaiseException(PROTECTION_FAULT)
 else
// Access is OK: form physical address and fetch it
offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
Register = AccessMemory(PhysAddr)
```

So, for each memory reference we need two memory references and that makes *paging* a slow process by a factor of 2 (or more). Expediting the translation must be done

Memory Trace

Let us see the memory access for paging using a memory trace for the following program (array.c)

```
int array[1000];
...
for (i = 0; i < 1000; i++)
array[i] = 0;
```

Compilation and running the code in UNIX may be done using the command

```
prompt> gcc -o array array.c -Wall -O
prompt> ./array
```

Finally the *objdump* of the array initialisation is shown below [For simplicity we have assumed all instructions are 4-bytes long; which is not true for x86]

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

Memory Trace ...contd.

- The first instruction $movl\ 0x0,\ (\%edi, \%eax, 4)$ moves 0 to the VA of the array. This is computed as edi holds the base address (assumed) of the array with which we add the contents of eax * 4 (a scale factor of 4 is used as it is an integer array)
- eax is increased by 1 ($incl\ \%eax$) to get the next array index; and it is
- compaerd with $0x03e8 = 1000$ using $jne\ 0x03e8, \%eax$
- if not we go back to repeat moving 0 else we move to the next instruction beyond the for loop

To understand which memory accesses this instruction sequence makes (at both the virtual and physical levels), we'll have to assume something about where in virtual memory the code snippet and array are found, as well as the contents and location of the page table.

Memory Trace ...contd.

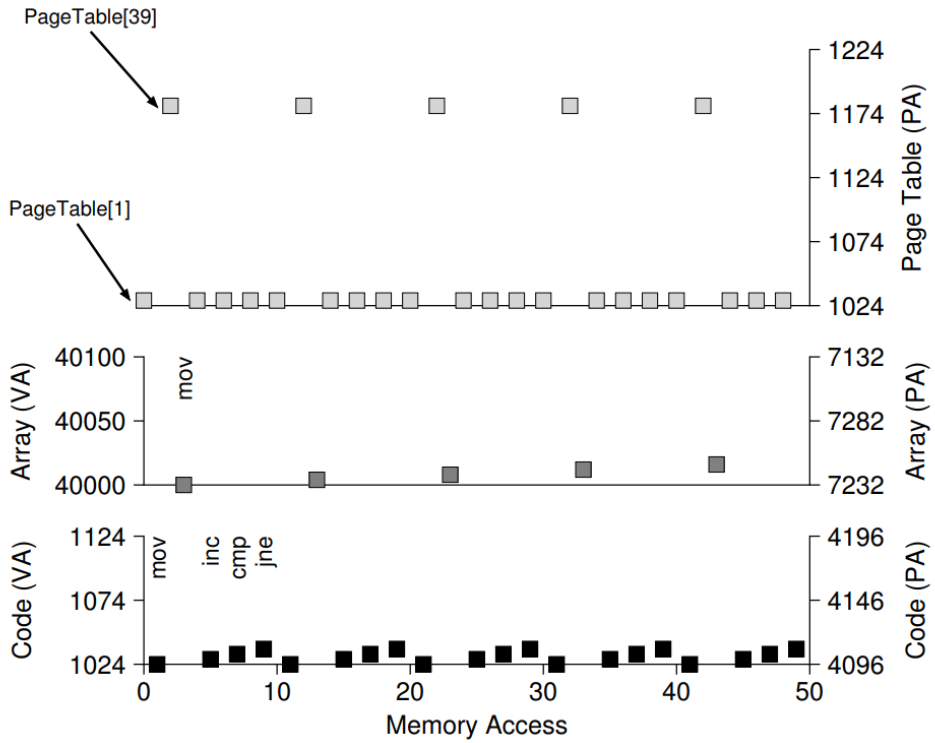
- Assume virtual address space of size 64 KB and a page size of 1 KB.
- page table is a linear array starting in PA location 1 KB = 1024
- the code lives on locaqtion 1024 (see the address of the 1st instruction) of VA. So, it is on VPN 1 (VPN 0 is for 0 to 1023). Also, assume
- VPN 1 maps to PFN 4. Aslo assume that the array (4000 bytes) and starts from VA 40000 and ends at 40000+4000-1. The corresponding VPN is from 39 to 42; also assume
- VPN 39 ->PFN 7; VPN 40->PFN 8; VPN 41 ->PFN 9; VPN 42->PFN 10

During execution

- each instruction fetch will generate two memory references: one to the page table to find the physical frame that the instruction resides within, and one to the instruction itself to fetch it to the CPU for processing.
- one explicit memory reference in the form of the mov instruction; this adds another page table access first (to translate the array virtual address to the correct physical one) and then the array access itself

Memory Trace ...contd.

The bottom one shows the instruction memory references on the y-axis (with VAs on the left, and the actual PAs on the right); the middle one shows array accesses; finally, the topmost graph shows page table memory accesses (just physical, as the PT in this example resides in PM). The x-axis, for the entire trace, shows memory accesses across the first five iterations of the loop (there are 10 memory accesses per loop, which includes 4 instruction fetches, 1 explicit update of memory, and 5 page table accesses to translate those 4 fetches and 1 explicit update).



Paging : Summary

We got the basic concept of paging which is advantageous than segmentation as it offers

- Lower external fragmentation, as paging is intentional break-up of the free space into small fixed size chunks
- It is quite flexible, enabling the sparse use of virtual address spaces.

However, paging has its blemishes as well; particularly if it is implemented carelessly

- May lead to slower access
- Waste of memory to hold the (big) page tables