Main Memory

The primary purpose of any computer is to execute programs –

- during execution; the program (or a part of it) must reside in the main memory.
- So, leaving the CPU –main memory is the most important resource of the computing system; and must be managed properly

The performance of any modern OS is based on its ability to manage processes (programs in execution; in a nutshel).

- There may be 'n' of processes concurrently (even parallely) running; and each asking for a private address space
- as well as they may be swapped in and out of the lower level storage on demand to create a virtual environment that a program is not limited by the size of the physical main memory (main memeory).

Main Memory

All such thing is controlled and supervised by memory management system of the OS which is often supported by the hardware architecture of the system. The charcateristics of the main memeory and our expectation is listed beow:

Characteristics	Expectation
Volatile	Non-volatility
Size is limited to some GBs	no limit
Costly	Cheap
Slow	at par with the CPU

Memory Hierarchy

The panacea of having a main memory with all good things (unlimited size, fast, non-volatile etc.) can be achieved through a memory hierarchy where we have

- Few words of superfast memory built-in right on the CPU (Registers)
- Few MBs of very first cache memeory (Multi-level; some inside the CPU and some outside)
- Moderately enough medium speed not so expensive main-memory (DRAM)
- Few terabytes of cheap and non-volatile magnetic memory (or solid state device) and
- More magnetic removable (or flash) device [USB etc.]

The main job of the OS is to abstract this hierarchy into a usable model to be applicable accross this hierarchy.
These are all achievable using Memory Mangement Techniques with proper architectural support form the h/w.

Memory Virtualisation

Like CPU the memory (main resource other than the CPU) needs virtualisation to offer Large (unlimited) address space to all the processes – the goals are

- Transparency
  - Virtual memeory would be invisible to the running program – running program thinks that it has got its private address space
- Efficiency
  - The virtualisation should not make a program slow – nor the overhead supporting virtualisation demands a lot of resources
- protection
  - Protect process from one another and also protect the OS from a process

The main job of the OS is to abstract this hierarchy into a usable model to be applicable accross this hierarchy.
These are all achievable using Memory Mangement Techniques with proper architectural support form the h/w.

Address: Logical/Virtual and Physical

In all modern general purpose computing system the CPU generates logical (virtual) address which is then mapped by the Memory Management Unit (mmu) to a physical (actual address) memory address providing an abstraction across the memory hierarchy. The following table helps getting the idea.

Aspect	Logical Address	Physical Address
Basic	Virtual Address	Location in Main Memory
Address Space	The set of all logical addresses	The corresponding physical addresses in Memeory
Visibility	Visible to the user	Is not visible
Access	Logical address	No direct access

Address Binding

Every program to be run is available on a secondary memory (usually a disk) as binary executable file. From the source file to executable form the addresses take different forms.

- Addresses are symbolic in a source file; (e.g., extern int mydata=5;) –
- Compiler binds these addresses to relocatable form (e.g., address of mydata is 10 words beginning of the data segment)
- Loader turns these addresses to absolute address
- During execution the process can be moved to one memory segment to another – in this case the binding is delayed to run-time. Special hardware (MMU) is needed to do the mapping

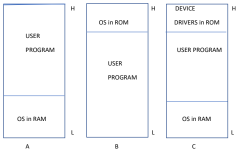
Memory Management: Simple to Abstract

The focus will be on the programmer's model of the main memory and how we can mange with an aim to keep multiple running processes in the Main memeory.

No abstraction: For early systems (before 60's for mainframe, 70s for Mini '80 for PC) – there was no mapping, the address generated is considered the final physical address. Thus the instruction MOV Rx, 10000H loads Rx the value stored in physical location 10000H.

Relative positions of the User Program and Kernel for three (A, B and C) different schemes

A : Used in early mainframes and mini. B : In some embedded system and C: In early PCs. In all it was hard to achieve multi-processing and absence of protection leads to catastrophe.



Memory Management: Multiple program no abstraction

Even without any abstraction/mapping multiple programe may be run concurrently by

- By taking the copy of the program running and swapping it with another (whose memory copy is already stored, say) in the disk
- By putting non-obverlapping programs in the memory – and switch execution between the loaded programs

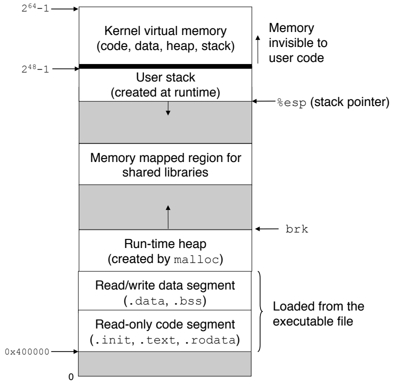
Exposing the processes to phyical memory has several drawbacks

- The process has access to every physical address – so interfaring with others including changing the OS is so easy.
- Implementation of mutiprocessing is hard or impossible

Thus memory abstraction is a necessity to implement multiprocessing and protection.

Process - execution of a program

A process has three distinct part, all stored in the memory, i) Code and Data; ii) stack and ii) heap. It goes like the following figure in LINUX.



2^64−1
2^48−1
0x400000
0
Kernel virtual memory (code, data, heap, stack)
Memory invisible to user code
User stack (created at runtime)
%esp (stack pointer)
Memory mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write data segment (.data, .bss)
Read-only code segment (.init, .text, .rodata)
Loaded from the executable file

Addresses of different parts of a running program

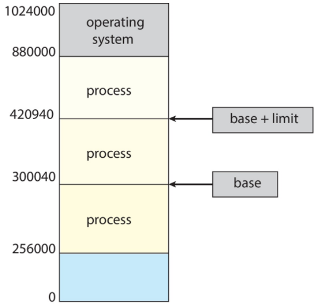
In order to see the addresses from where they start run the following program

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
printf("location of code  : %p\n", (void *) main);
printf("location of heap  :%p\n", (void *) malloc(1));
int x = 3;
printf("location of stack : %p\n", (void *) &x);
return x;
}
location of code/data : 0x55dfdba48189
location of heap      : 0x55dfdd64336b0
location of stack     : 0x7ffeaa21do34
```

Though all the addresses are virtual – we see that the heap starts right after the code/data section and the stack starts from the top

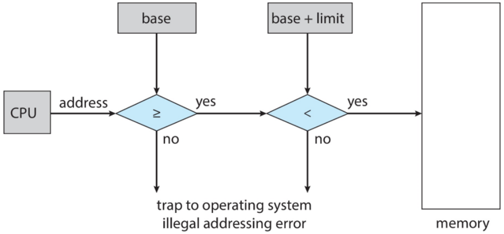
The notion of an address space

Assume that a process lives in an address space (0 to n-1) – the set of addresses that a process can generate to access a memeory location. This address space is private to each process. Naturally, there would be a mechanism to map each process in separate physical memory and some technique to protect interference.



The base and a limit register

This simple technique (used in the first supercomputer the CDC 6600) can be used to map different processes to occupy different physical adresses (start address loaded in base register) and how far you can go from the base in a limit register (program size).



Every time for a memory reference (opcode fetch or data read/write) the validity is checked by the hardware if there is any encroachment to the address space of another process as well as any reference beyond the permitted limit.

Addresses – no translation

Consider the following function

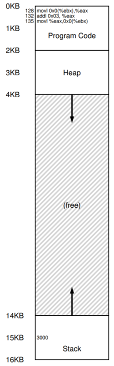
```
void f()
{             low level code for the instruction x=x+3
int x=3000; assuming that ebx is loaded with the address of x
        :               movl 0x0(%ebx), %eax
                        addl  0x3, %eax
x = x + 3;              movl %eax, 0x0(%ebx)
:
}
```

Addresses : No translation

If the three lines of assembly instruction is stored in location 128, 132 and 135 in the code section – then the actions are :

- Read the instruction from location 128
- execute it to read x using the address of x in ebx and place it in eax
- fetch and Execute the add instrucion starting at location 132 which adds 3 to eax
- fetch and execute the instruuction from location 135 to write the new value of x

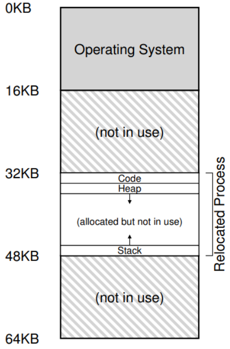
Addresses – no translation



```
128 movl 0x0(%ebx),%eax
132 addl 0x03, %eax
135 movl %eax,0x0(%ebx)
```

From the program's perspective, its address space starts at address 0 and grows to a maximum of 16 KB; all memory references it generates should be within these bounds. However, to virtualize memory, the OS wants to place the process somewhere else in physical memory, not necessarily at address 0.

Addresses – Single process – Relocation



OS using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32 KB. The other two slots are free (16 KB-32 KB and 48 KB-64 KB).

Single process – Relocation, ...contd.

To understand the relocation process better, let's trace through what happens when a single instruction (128: movl 0x0(%ebx), %eax) one instruction from our earlier sequence is executed.

- The program counter (PC) is set to 128; to fetch this instruction it first adds the value to the the base register value of 32 KB (32768) to get a physical address of 32896; and the h/w then fetches it from that physical address.
- Next, the processor begins executing the instruction. At some point, the process then issues the load from virtual address 15 KB, which the processor takes and again adds to the base register (32 KB), getting the final physical address of 47 KB and thus the desired contents.

Software based– Relocation

In the early days, before hardware support came handy, a relocation purely via software methods was used

- The technique was static relocation, in which a loader takes an executable that is about to be run and rewrites its addresses to the desired offset in physical memory.
- e.g., if an instruction was a load from address 1000 into a register (e.g., movl 1000, %eax), and the address space of the program was loaded starting at address 3000 (and not 0, as the program thinks), the loader would rewrite the instruction to offset each address by 3000 (e.g., movl 4000, %eax). In this way, a simple static relocation of the process's address space is achieved.
- However, static relocation has numerous problems. First and most importantly, it does not provide protection, as processes can generate bad addresses and thus illegally access other process's or even OS memory.

Dynamic (H/W based Relocation)

Two hardware registers, *base* and *bound* within each CPU: one This base-and-bounds pair is going to allow us to place the address space anywhere we'd like in physical memory, and do so while ensuring that the process can only access its own address space.

- each program is written and compiled as if it is loaded at address zero. However, when a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value.
- In the example above, the OS decides to load the process at physical address 32 KB and thus sets the base register to this value. Interesting things start to happen when the process is running. Now, when any memory reference is generated by the process, it is translated by the processor in the following manner: physical address = virtual address + base

Dynamic (H/W based Relocation... contd)

The use of bound register is not explicit in the example shown above.

- The bounds register is there to help with protection. The processor will first check that the memory reference is within bounds to make sure it is legal; in the simple example above, the bounds register would always be set to 16 KB.
- If a process generates a virtual address that is greater than the bounds, or one that is negative, the CPU will raise an exception, and the process will likely be terminated. The point of the bounds is thus to make sure that all addresses generated by the process are legal and within the "bounds" of the process.

Dynamic (H/W based Relocation... contd)

- The base and bounds registers are hardware structures kept on the chip (one pair per CPU). Sometimes people call the part of the processor that helps with address translation the memory management unit (MMU); as we develop more sophisticated memory management techniques, we will be adding more circuitry to the MMU.
- A bound registers can be defined in one of two ways.
  - In one way (as above), it holds the size of the address space, and thus the hardware checks the virtual address against it first before adding the base.
  - In the second way, it holds the physical address of the end of the address space, and thus the hardware first adds the base and then makes sure the address is within bounds. Both methods are logically equivalent; for simplicity, we'll usually assume that the bounds register holds the size of the address space

Example Translations

To understand address translation via base-and-bounds in more detail, let's take a look at an example. Imagine a process with an address space of size 4 KB (0 to 4095; yes, unrealistically small) has been loaded at physical address 16 KB. Here are the results of a number of address translations:

- Virtual Address 0 $\rightarrow$ Physical Address 16 KB ($16 \times 1024 = 16384$)
- VA 1 KB $\rightarrow$ PA 17 KB
- VA 3000 $\rightarrow$ PA 19384 ($16384 + 3000$)
- VA 4400 $\rightarrow$ Fault (out of bounds)
- it is easy for you to simply add the base address to the virtual address (which can rightly be viewed as an offset into the address space) to get the resulting physical address.
- Only if the virtual address is "too big" or negative will the result be a fault (e.g., 4400 is greater than the 4 KB bounds), causing an exception to be raised and
- the process to be terminated (The dreaded segmention fault – core dumped message from the OS ; which frequently happens for a novice programmer)

Address Translations: OS issues

There are a number of new OS issues that arise when using base and bounds to implement a simple virtual memory. Specifically, there are three critical junctures where the OS must take action to implement this base-and-bounds approach to virtualizing memory.

- First, The OS must take action when a process is created, finding space for its address space in memory. Fortunately, given our assumptions that each address space is
  - (a) smaller than the size of physical memory and
  - (b) the same size, this is quite easy for the OS; it can simply view physical memory as an array of slots, and track whether each one is free or in use.
  - When a new process is created, the OS will have to search a data structure (often called a free list) to find room for the new address space and then mark it used.

Address Translations: OS issues ... contd

Second, the OS must take action when a process is terminated; such as

- reclaiming all of its memory for use in other processes or the OS by putting its memory back on the free list, and
- cleans up any associated data structures as need be.

Third, the OS must also take action when a context switch occurs.

- There is only one base and bounds register on each CPU, after all, and
- their values differ for each running program, as each program is loaded at a different physical address in memory.
- save and restore the base-and-bounds pair when it switches between processes. Specifically, when the OS decides to stop running a process, it must save the values of the base and bounds registers to memory, in some per-process structure such as the process structure or process control block (PCB).
- Similarly, when the OS resumes a running process (or runs it the first time), it must set the values of the base and bounds on the CPU to the correct values for this process

Address Translations: OS issues ... contd

Note that when a process is stopped (i.e., not running), it is possible for the OS to move an address space from one location in memory to another rather easily.

- To move a process's address space, the OS
  - first deschedules the process; then, the OS copies the address space from the current location to the new location;
  - finally, the OS updates the saved base register (in the process structure) to point to the new location. When the process is resumed, its (new) base register is restored, and it begins running again, oblivious that its instructions and data are now in a completely new spot in memory!
- Also note that access to the base and bounds registers is obviously privileged. Special hardware instructions are required to access base-and-bounds registers; if a process, running in user mode, attempts to do so, the CPU will raise an exception and the OS will likely terminate the process. Only in kernel (or privileged) mode can such registers be modified. Imagine the havoc a user process could wreak1 if it could arbitrarily change the base register while running.

Virtualisation : 1st Summary

With address translation, the OS can control each and every memory access from a process, ensuring the accesses stay within the bounds of the address space.

- We need h/w support (base and bound registers) which performs the translation quickly for each access, turning virtual addresses (the process's view of memory) into physical ones (the actual view).
- Address translation is transparent to the relocated process; it has no idea that the memory references are being translated.
- the efficient virtualisation needs little h/w support and offers protection

Unfortunately, this simple dynamic relocation may not be efficient

- Because we proposed to place the whole address space of each process in the memory. Now, the stack and heap may not be too big, all of the space between the two is simply wasted.
- This type of waste is usually called internal fragmentation, as the space inside the allocated unit is not all used (i.e., is fragmented) and thus wasted.

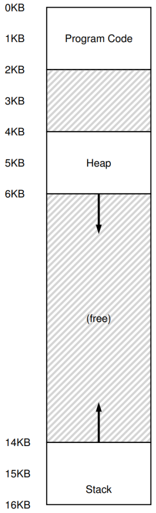
So, to reduce internal fragmentation we go for the generalisation of the base-and-bound concept in the form of segments.

Segmentation

We go for segmentation to reduce internal fragmentation and manage a large process address space that may exceed the physical memory. The basic idea is to provide multiple base and bound registers for each logical segment of the process.

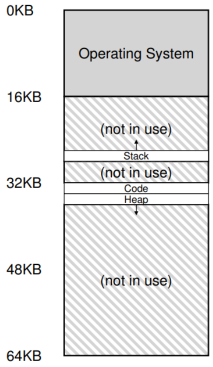
- A segment is a contiguous portion of an address space of arbitrary length
- What segmentation allows the OS to do is to place each one of those segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space (see figure below)
- Segmentation allows the OS to do is to place each one of those segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space

Segmentation...contd.



Segmentation...contd.

With a base and bounds pair per segment, we can place each segment independently in physical memory.
Only used memory is allocated space in physical memory, and thus large address spaces with large amounts of unused address space (which we sometimes call sparse address spaces) can be accommodated.



Segmentation...contd.

The MMU needs three pairs of base and bound registers (holding the size of a segment) for the three segments; namely i) code; ii) heap and iii) stack. The table below shows an allocation using the data presented earlier.

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Translation:
Code VA = 100 : PA = 100 + 32KB = 32868 (within the 2K bound)
Heap VA = 4200 : PA = 34K + 4200 = 39016 (wrong address) we have to take the offset of 4200. As heap starts from the VA of 4K (4096) so the offset is 4200 - 4096 = 104. The base register PA = 34 K (34816) so the correct PA is 34920. Illegal address like 8K which is beyond the end of the heap would be tracked as address out of bound and trapped to OS to get segmentation fault error message

Segmentation...contd.

For explicit address translation one mechanism is to divide the address space into segment (n-higher bits of VA) and the offset (rest of the Virtual address bits) as shown below with 14-bits (as used in VAX/VMS systems)

```
               X  X   Y Y Y Y Y Y Y Y Y Y Y Y
2- bits (X X) for segment | rest 12-bits are for offset
2 bits are enough to encode 00= code; 01 = heap, 10 = stack etc.
```

Referring to the previous heap VA address 4200 – the translated value would be

```
            0 1   0 0 0 0 0 1 1 0 1 0 0 0
01 --> Heap segment; 0 0 0 0 0 1 1 0 1 0 0 0 --> 104 in decimal
```

The h/w takes the first two bits to determine which base register – with which the offset is added to generate the PA. The offset makes bound checking easier; any value greater than 2K (i.e., bit $11 = 1$) would be illegal

Segmentation...contd.

Considering *base* and *bound* were arrays with one entry per segment the address translation, done by the h/w, may be written in form of a program

```
#define   SEG_MASK 0x3000
#define   SEG_SHIFT 0x0C  // 12 in decimal
#define   OFFSET_MASK 0xFFF
segment = (VA & SEG_MASK) >> SEG_SHIFT;
offset = VA & OFFSET_MASK;
if offset >= bound[segment]   raise_EXCEPTION(protection_FAULT);
else PA = base[segment] + offset;
register = access_MEMORY[PA];
```

Note that heap could start immediately after the code segment and only 1-bit may then be used to explicitly determine the segment. Some system implicitly determine the segment to beused; e.g.;

- If the address is coming from the PC – use code segment
- If it is from SP or base pointer then use stack segment
- For others it must be heap segment

Segmentation...contd.

Stack grows downward and for this reason and more better flexibility a bit (direction; 1 = +ve, 0 = -ve) may be used to assist translation.

Segment	Base	SIze	Direction
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Consider translating a virtual address of 15 KB which should map to a PA of 27 KB. The 15 KB virtual address is binary form would be 11 1100 0000 0000. This top two bits may be used to indicate the segment and the bottom, however, shows an offset of -3 KB (as the direction bit is negative). We add 2 KB segment size to get the final offset of -1KB which when added to the segment base address of 28 Kb we get 27 KB; the PA. The bound check can be done by the -ve offset which must be less than the segment size.

Segmentation...sharing

It is soon realized that segmentation offers a greater benefit to code (or data) sharing by mapping the same PA between multiple segments (The code part of the application *word* being shared by 50 simultaneous users while their data part, the documents they create, are all separate).

- the sharing reduces memeory requirements (50 copies of the *word* versus a single shared copy)
- for protection some more bit(s) would be the requirement

The allocation table may be re-drawn as shown below

Segment	Base	SIze	Direction	Protection
Code	32K	2K	1	RX
Heap	34K	2K	1	RW
Stack	28K	2K	0	RW

Note that

- the process would think that it is using its own private memory – and the sharing would be hidden
- along with checking the virtual address is within bounds, the hardware also has to check whether a particular access is permissible.

Segmentation...protection

It is soon realized that segmentation offers a greater benefit to code (or data) sharing by mapping the same PA between multiple segments (The code part of the application *word* being shared by 50 simultaneous users while their data part, the documents they create, are all separate).

- the sharing reduces memeory requirements (50 copies of the *word* versus a single shared copy)
- for protection some more bit(s) is/are required.

The allocation table may be re-drawn as shown below

Segment	Base	SIze	Direction	Protection
Code	32K	2K	1	RX
Heap	34K	2K	1	RW
Stack	28K	2K	0	RW

Note that

- the process would think that it is using its own private memory – and the sharing would be hidden
- along with checking the virtual address is within bounds, the hardware also has to check whether a particular access is permissible.

Segmentation...Fine and course ... Issues

The segmentation that we have seen are course grain segmentation as the address space is chopped into big pieces. However, some old systems (Notably; MULTICS) offered the *fine-grained* segmentation where OS supports thousands of segments of smaller sizes for greater flexibility. However, this requires

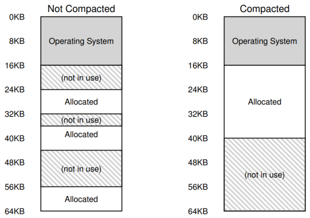
- more h/w and
- and more data structure support in the form of *segment table*

The OS in this case can manage the physical memory more efficiently as it has full control of small segments and their usage pattern.
From the foregoing discussion we get some bright prospects of segmentation; how some issues are

- Context switching – to be managed by storing and restoring the segment registers
- Managing free space in Physical memory – each process has got multiple segments of varying sizes

Segmentation...External Fragmentation/compaction

It is not difficult to see that the physical memory will have free space between the segments, known as *external fragmentation* and a new big segment may not find space in physical memory though the total unused memory may be more than the amount asked for.



Here a new segment (say, 23 KB) cannot be accommodated for the want of contiguous free space of that amount while the total free space is 24 KB. A compaction can be done to push the segments together opening up rooms of free space – however *compaction – primarily a move* takes a lot of time.

Free list

To reduce *external fragmentation* compaction cannot be a solution as it

- Time intensive; and
- Fragmentation again with the passage of time

A simpler approach is to use a *free list management* algorithm (several varities exist) that keeps track and makes large contiguous slots available whenever possible. The apporach used are

- First fit
- Best fit
- Worst fit; etc.

As the processes do occupy the physical memory and releases (OS does it) it in an extremely dynamic way – no matter the efficacy free space algorithm external fragmentation can only be reduced but not prevented.

Virtualisation ... 2nd Summary

Segmentation helps solve a number of problems

- Better sparse space management
- Faster translation using hardware
- Sharing of code (and r/o data; may be a table)

However, segmentation cannot fully manage the sparce space. For example, say, a sparsely used heap is on the memory as a single logical segment – it would remain in memory in order to be accessed every now and then. The solution is perhaps splitting of memory to a large number of smaller blocks or pages as we shall see later.