



PES UNIVERSITY

(Established under Karnataka Act No.16 of 2013)
100-ft Ring Road, BSK III Stage, Bangalore – 560 085
Department of Computer Science & Engg
Session: Jan-May 2021
UE19CS254: Operating Systems

UNIT 3 Notes

Read the sections mentioned

Chapter 8 Main Memory	
Topics	Page No
8.1 Main Memory: Hardware and control structures, OS support, Address translation	345-352
8.2-8.3 Swapping, Memory Allocation (Partitioning, relocation), Fragmentation	352-358
8.4 Segmentation	358-360
8.5 Paging, TLBs, context switches	360-371
8.6 Structure of the Page table	372-377
Chapter 9 Virtual Memory	
9.1-9.2 Virtual Memory – Demand Paging	389-399
9.3 Copy-on-Write, 9.4 Page replacement: Basic page replacement (FIFO page replacement and optimal page replacement)	400-407
9.4.4 LRU Page replacement	408
9.6 Thrashing	417-422
9.10 Case study	437-438

Text Book

[1]. “Operating System Concepts”, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne 9th Edition, John Wiley & Sons, 2016, Indian Print.

Memory management: Focus of the Unit 3 is based on the topics as listed in the course information.

Available Von Neumann system architectures follow stored program concept, which compels the program to be in memory during the time of execution. So, the role of this topic of memory management focuses on all those components where the program needs to be loaded into main memory before the start of execution and also to manage the memory space throughout the execution.

Memory consists of a large array of bytes, each with its own address. When we look at program execution cycle, we see that there are 2 broad steps –

- Fetch an Instruction from memory, Decode the instruction, operands are fetched (from memory or registers)
- Execution of an instruction.
- After the instruction is executed, results are to be stored back.

For this sequence of fetch and execute cycles, we see that stream of addresses are getting generated. Loader is a software component that loads the executable to main memory location and communicates this address to CPU. Also Operating system has to ensure proper entries are made in its process table when the program is submitted for execution and the process table gets updated. Starting address for the process is loaded into Program counter and this initiates the cycle of fetch-execute steps. Processor is responsible for generating these addresses. For now, we can call these addresses as logical address which refers to the process address space. Memory management Unit is responsible to map this logical address to physical address. Physical address refers to the actual physical memory, whereas logical address relates to the process address space.

With the concept of **demand paging and virtual memory**, it is possible to load only those pages of process into main memory when they are needed. This facilitates the execution of those processes which are larger than the actual physical memory. But this arrangement requires that the CPU should be able to access the logical address space – for example a software application size is 10 GB, but available physical memory (as RAM) is 8 GB. Even though we intend to run only one application, entire application will not fit into the memory space. So, adhering to the stored program concept, we will not be able to load this process completely.

Virtual memory and demand paging concepts alleviate this problem, and it is allowed that the entire process need not be in main memory before the execution starts. Instead, as and when the address

generated by the processor requires a page access, the page will be loaded into main memory and the read/write actions were carried out.

At the time of linking, the required library functions have to be linked to the user programs. Gcc compiler by default has dynamic linking, but if the user wants to use static linking, then command line option can be used for that.

About **dynamic linking** – system library functions that are frequently used by different applications, could be allocated main memory space once and the same copy of the function could be linked to the applications that use these functions. Thus the library functions do not need repeated memory allocations with each process, which results in optimized memory usage per process.

On the other hand, with **static linking**, all the required library function definitions are allocated main memory, explicitly per process. If there are 5 processes utilizing a particular library, then there will be 5 copies kept for the same function, across 5 different processes. It is obvious that with static linking main memory usage will be much higher.

Screenshot in the below image, illustrates the memory allocated for the case of dynamic linking as 3398 bytes, where as with static linking – it is 928289 bytes. These numbers indicate the difference in the memory requirement for the same process, when the linking is different. Thus it is easy to understand why gcc keeps dynamic linking as a default option.

```
thread1.c: In function 'printHello':
thread1.c:9:5: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
   9 | var=(int)n;
     | ^
thread1.c: In function 'main':
thread1.c:21:41: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
   21 | y=pthread_create(&tid, NULL, printHello,(void *)x);
     | ^
prafullata@DESKTOP-DJ0DIMA:~$ size t1
  text    data     bss     dec     hex filename
  2662     664      72    3398     d46 t1
prafullata@DESKTOP-DJ0DIMA:~$ gcc -static -pthread thread1.c -o t1
thread1.c: In function 'printHello':
thread1.c:9:5: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
   9 | var=(int)n;
     | ^
thread1.c: In function 'main':
thread1.c:21:41: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
   21 | y=pthread_create(&tid, NULL, printHello,(void *)x);
     | ^
prafullata@DESKTOP-DJ0DIMA:~$ size t1
  text    data     bss     dec     hex filename
882125    21140    25024  928289   e2a21 t1
```

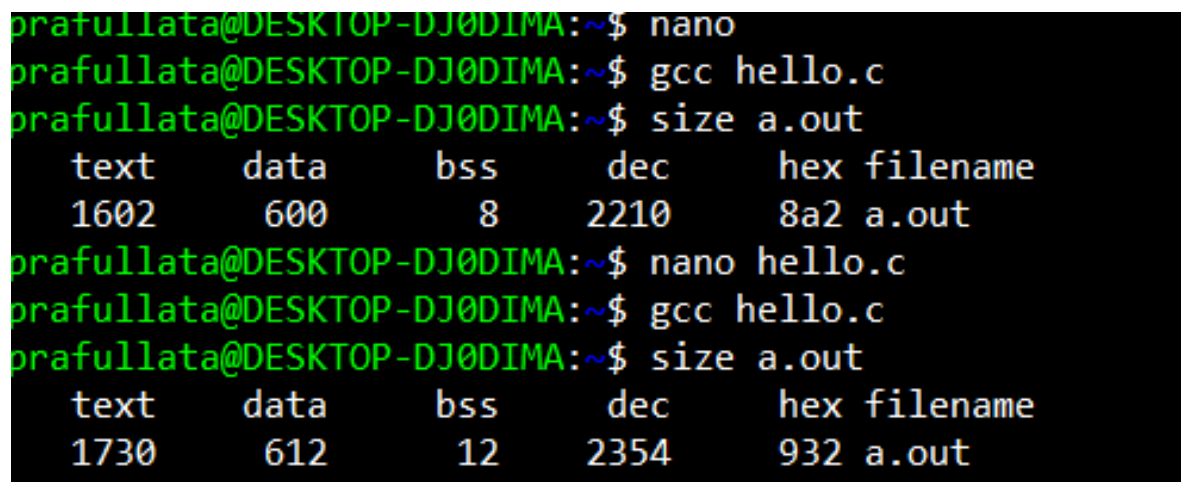
Address Binding:

As we are trying to understand the process memory requirement with the linking option used, likewise, it is also important to understand the **address binding** activity as well.

Binding is an activity that can take place during compile time or run time and also can remain static once done or may keep changing during execution. It is important to know these details.

Consider the program as a simple hello world. We compile the code with gcc and get the executable as a.out. If we check command - `size a.out`, then we will be able to see the memory size that is allocated to this process, excluding the stack and heap space. So, what we get to see here are, three sections – code or text section that indicates the code size, data section and bss are that part of main memory being allocated to process which holds the static or global variables. As we know the scope and lifetime of global or static variables are different than local variables. These variables will be visible and alive throughout the program/file. If there is a same name usage in the local scope, then the local variable gets precedence.

So, we understand that the storage allocation or address binding for global and static variable happens before runtime and remains static once done. To verify this fact, we can see the screenshot as given below.



```
prafullata@DESKTOP-DJ0DIMA:~$ nano
prafullata@DESKTOP-DJ0DIMA:~$ gcc hello.c
prafullata@DESKTOP-DJ0DIMA:~$ size a.out
   text    data     bss     dec     hex filename
   1602     600        8    2210     8a2 a.out
prafullata@DESKTOP-DJ0DIMA:~$ nano hello.c
prafullata@DESKTOP-DJ0DIMA:~$ gcc hello.c
prafullata@DESKTOP-DJ0DIMA:~$ size a.out
   text    data     bss     dec     hex filename
   1730     612       12    2354     932 a.out
```

First usage of size command above is for the simple hello world program.

Next usage of size is for that executable which has 3 additional variables added as static and global variables. So, we can observe the changes in data and bss sections.

If the variable has to be auto initialized to zero, then it will be allocated to bss (Block Started by Symbol) and otherwise, if the variable is explicitly initialized by the user, then it will be allocated to data section.

When we look at the memory allocations for these processes, we come to a point where we need to check what kind of allocations are in practice. We have an option where the total memory required could be allocated in one contiguous block or it could be non-contiguous. With the contiguous option mapping becomes easier and straight forward, but practically to have such a single block available for every process would be impossible.

Thus we can say that non-contiguous approach is more practical. With this we have a choice where the memory can be seen as chunks – again with a variety that each chunk is of fixed length or variable length.

Broadly from this approach we can say that,

Variable length non-contiguous approach – extends as **segmentation** concept

Fixed length non-contiguous approach – extends as **paging** concept.

Considering paging scheme for the memory usage, we would be curious to know the typical page size. As can be seen in the below screenshot, the typical size for a memory page is 4096 bytes or 4Kb.



```
prafullata@DESKTOP-DJ0DIMA:~$ stat thread1.c
  File: thread1.c
  Size: 571          Blocks: 0          IO Block: 4096   regular file
Device: 2h/2d Inode: 2533274790811192 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/prafullata)   Gid: ( 1000/prafullata)
Access: 2021-02-07 17:37:41.126496900 +0530
Modify: 2021-02-07 17:37:41.126496900 +0530
Change: 2021-02-07 17:37:41.126496900 +0530
 Birth: -
prafullata@DESKTOP-DJ0DIMA:~$ getconf PAGESIZE
4096
prafullata@DESKTOP-DJ0DIMA:~$
```

To clarify the notion of paging, we can look at the following example –

Given a logical address space of 256 pages with a 4-KB page size, which is mapped onto a physical memory of 64 frames, find out the following values.

- How many bits are required in the logical address?
- How many bits are required in the physical address?

Logical address space pertains to the application and as given in the statement, there are 256 pages and each page is of 4KB size.

To identify 256 pages – we need 8 bits of address – (because $2^8 = 256$)

And to access all the bytes of a 4KB page – we need 12 bits – (because $2^{12} = 4096$)

So, the logical address should be having $8+12 = 20$ bits.

Now, about the physical space – there are 64 frames, each frame is of same size as given page – i.e. 4KB.

To identify 64 frames – we need 6 bits of address – (because $2^6 = 64$)

And to access all the bytes of a 4KB frame – we need 12 bits – (because $2^{12} = 4096$)

So, the physical address should be having $6+12 = 18$ bits.

Here is another example for the page number and offset values -

Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided in decimal numbers):

- 3085
- 42095
- 215201
- 650000
- 2000001

As per the given data, each page is of 1 KB size – which can store 1024 bytes. So, we will have to first map the given address as the page number and then within that page – what will be the offset for the referenced byte. We will just divide the given decimal number by 1024. This will give the page number – assume the starting number as ZERO.

- For 3085 – page number is $3085/1024$ (because page size is 1KB) = 3.0126.
 - So, page number is 3.
 - Now for offset – $3 * 1024 = 3072$. (This is the last address of previous page.) $3085 - 3072 = 13$ is the offset.
- For 42095 – page number is $42095/1024 = 41.1083$.
 - So, page number is 41.
 - Now for the offset – $41 * 1024 = 41984$. Now do – $42095 - 41984 = 111$. So this is the offset.

On the same lines, we can calculate for other decimal values as well, and the answer is as given below.

Decimal number	Page number	offset
3085	3	13
42095	41	111
215201	210	161
650000	634	784
2000001	1953	129