Lecture [redacted] Virtual Memory Management

[redacted]

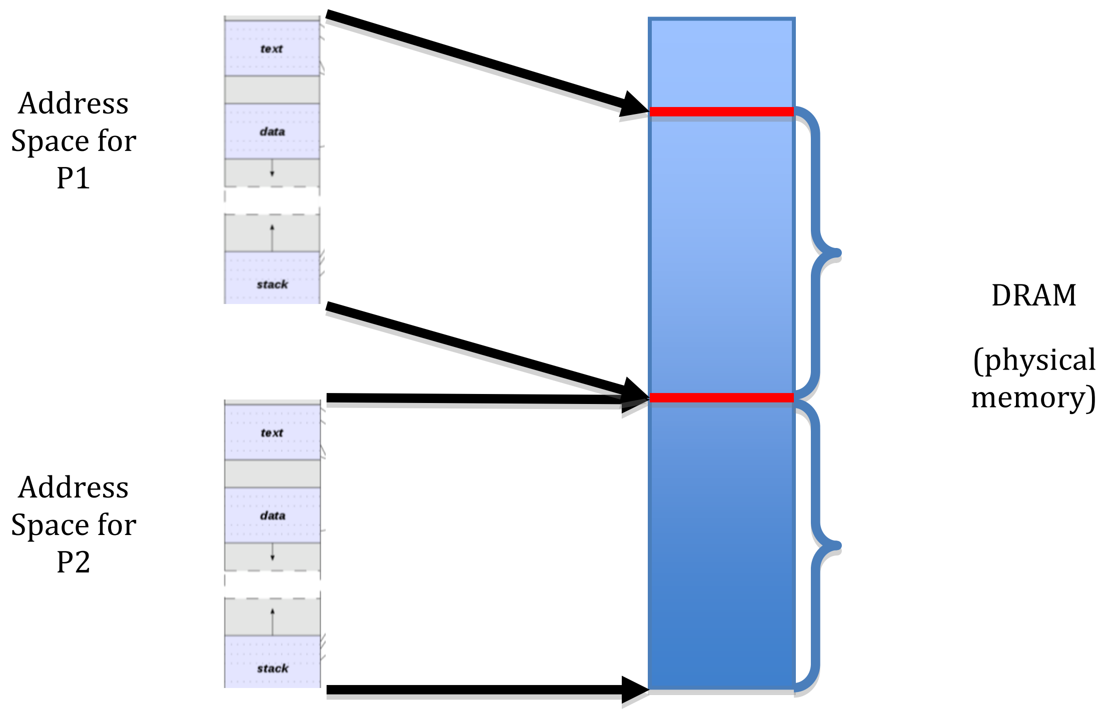
[redacted]

[redacted]

Review:

- In order to be executed, code must be brought into main memory.
- The MMU is a part of the CPU hardware responsible for virtual to physical address translation. The OS virtual memory manager is responsible for (with the help of the MMU) for allocating physical pages to processes and the OS itself.

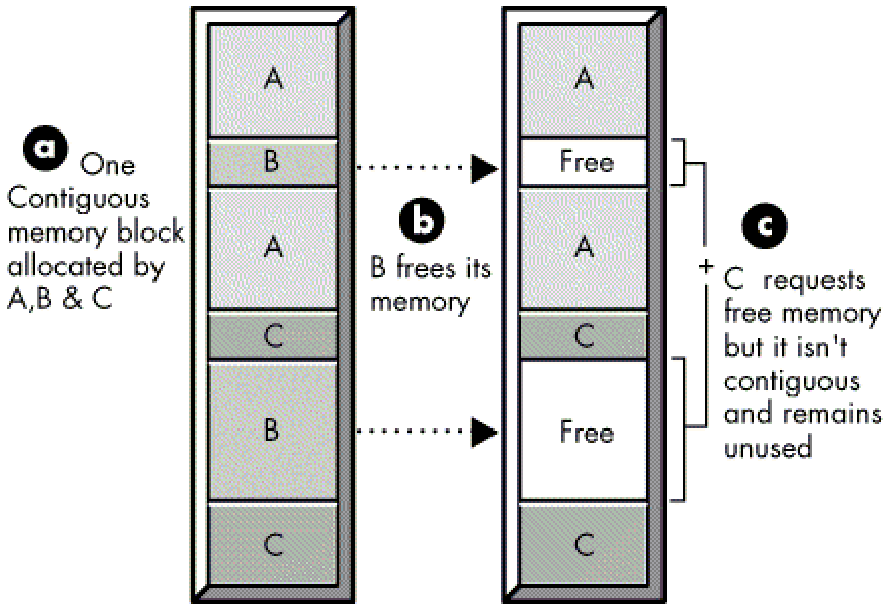
Contiguous Allocation



A major problem with this type of allocation (contiguous) is that it will result in "external fragmentation." External fragmentation is when there are empty, "unusable" spots within the memory for a new process to be allocated to. This leads to inefficient gaps in the DRAM.

This occurs because the MMU allocates the address spaces in a contiguous fashion. When a process finishes and frees the memory, the now unused block may be too small for another process to allocate, or it is not contiguous with the already allocated memory of the requesting process.

External Fragmentation



a One Contiguous memory block allocated by A,B & C
A
B
A
C
B
C
b B frees its memory
A
Free
A
C
Free
C
c C requests free memory but it isn't contiguous and remains unused

What is the solution to external fragmentation? Paging!

Paging

- The allocation of memory can be NON-contiguous
  ⇒ Eliminates external fragmentation
- Physical memory is divided into fixed-sized blocks called ***frames***
  ⇒ Frame size is a power of 2 (between 512 bytes and 8,192 bytes)
- Logical memory is divided into blocks of the same size called ***pages***
- The OS memory manager will keep track of all free frames in the physical memory.
- A ***page table*** will be stored in the OS portion of virtual memory to hold virtual to physical page translations for each process. It is possible for a page table itself to be sometimes be "swapped out" (i.e., it may not necessarily fit entirely in DRAM always).

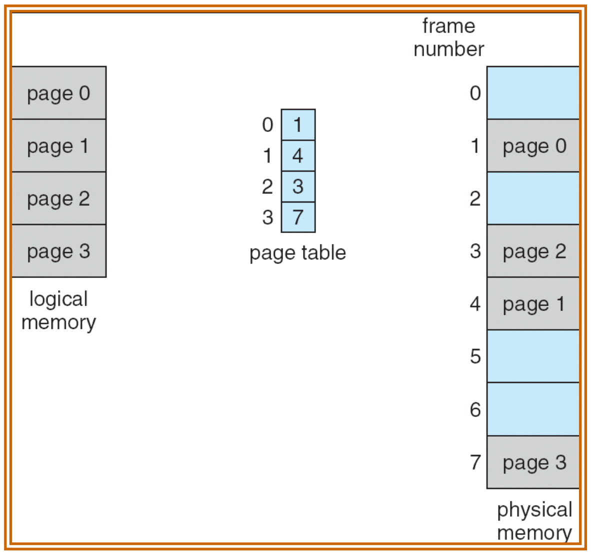
How Paging Works

If a program has ***n*** pages, we will only need to find ***n*** frames in the DRAM for allocation, regardless of their position in the physical memory.

1. In the following picture, a program logical memory it will require is divided into a total of **4 pages**.

2. These pages are then allocated into **4 separate frames** in the physical memory (non contiguously).

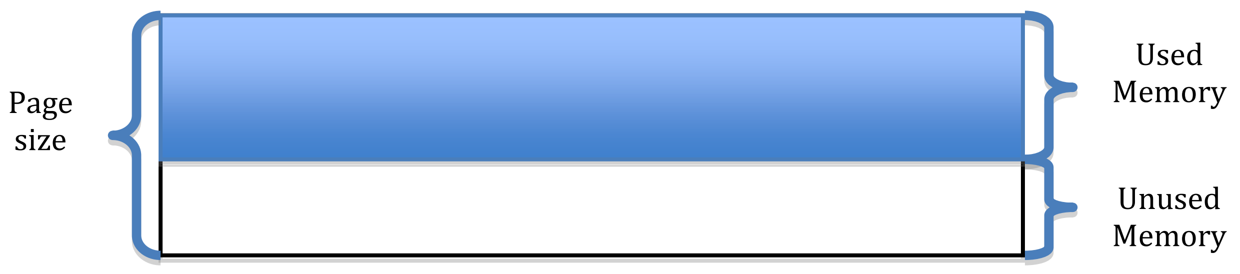
3. To find where each individual page is located in the DRAM, a **page table** isupdated that indexes the program's page number with its corresponding frame number



Although paging eliminates external fragmentation, the allocation method gives birth to a whole new problem called ***internal fragmentation.***

Internal fragmentation is when the required memory for a page is less than the page's total capacity. For instance, if a page is 4 KB, and a program will require only 2.5 KB of physical memory, 1.5 KB of memory will be unused. Memory allocation in the form of paging requires that all memory be divided into a specific quantum, so the MMU will still allocate a full 4 KB page, regardless of how much is actually used.

Internal Fragmentation

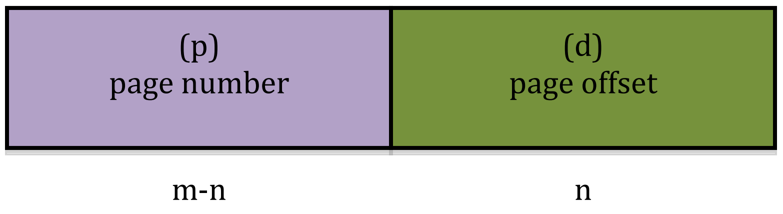


Address Translation

Now that the physical and logical memories have been split up into symmetrical sizes for this paging allocation, we need a fast and reliable method to translate the program's virtual memory to where the page was actually stored in the physical memory.

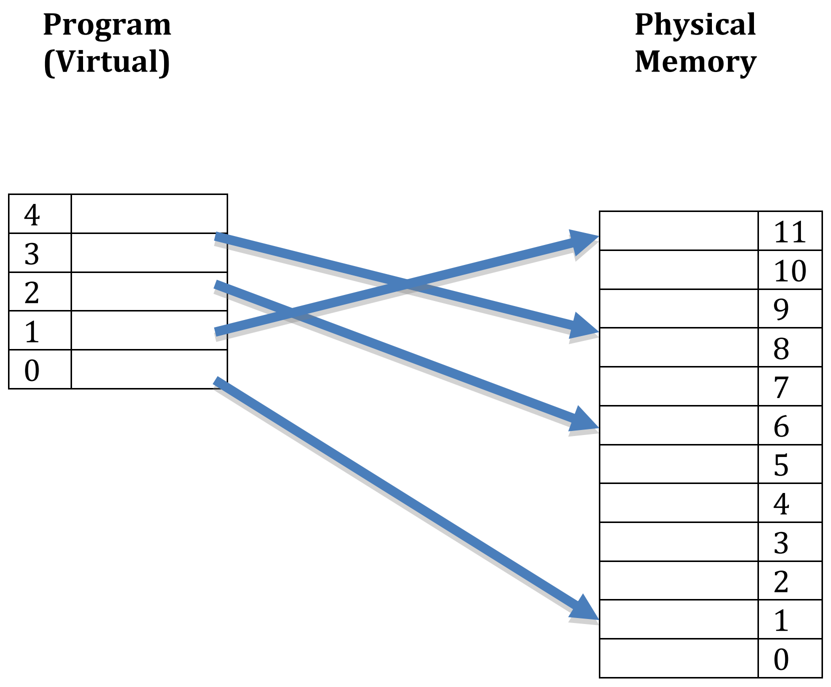
Address generated by CPU is divided into:

- **Page number (*p*)** – used as an index into a *pagetable* which contains base address of each page in physical memory
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit



** For given logical address space 2^m *and page size*2^n**

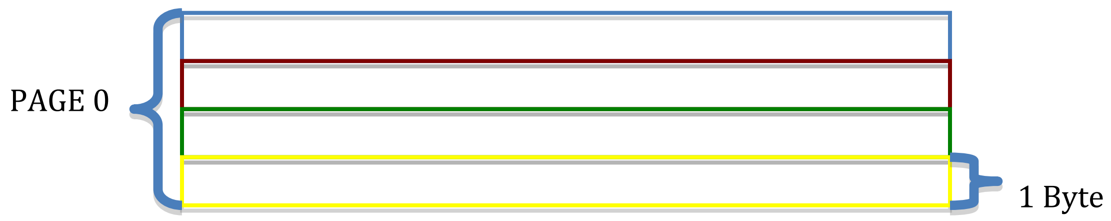
Example (4 KB per page):



Simplified Page Table

Virtual Address	Physical Address
0	0
1	10
2	5
3	8

Within the page (4 KB):

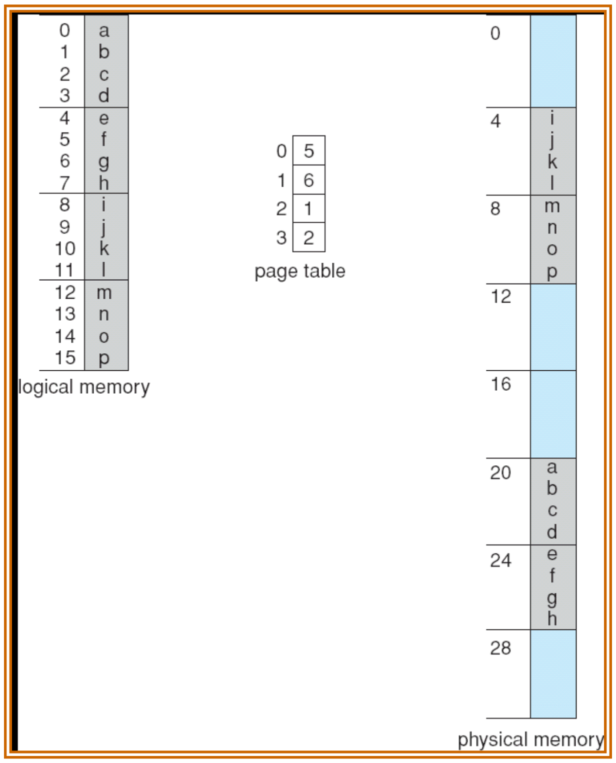


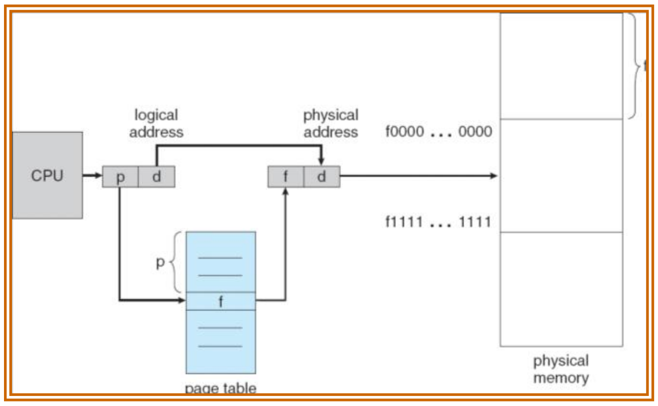
The page table must be kept in the OS's memory section of the RAM. If the user section had access to the page table, it could potentially modify the translation indexes for memory that a completely different program had allocated. This could lead to very undesirable behavior. Hence, these translations must be kept separate for use only by the operating system in order to accurately access a particular program's page in physical memory.

Paging Hardware

In this picture, we are using 32-bit addressing and 4 Bytes per page.

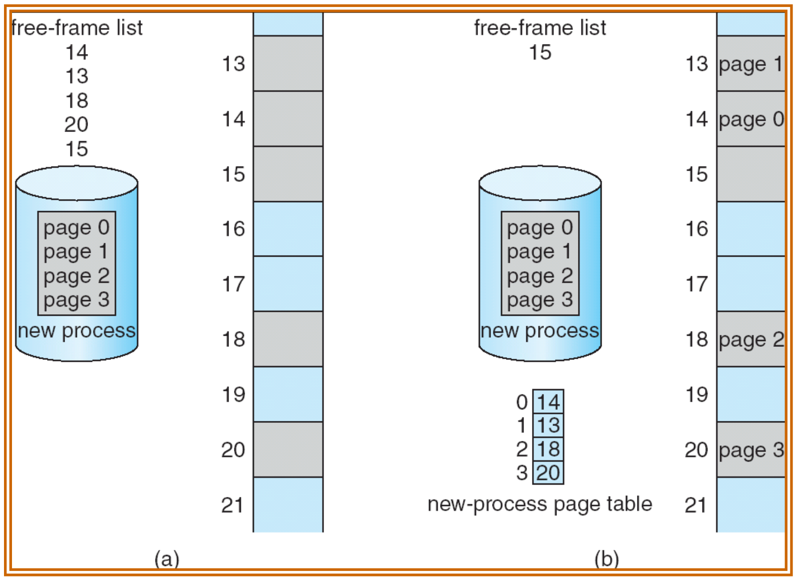
- Page 0 = {a,b,c,d}
- Page 1 = {e,f,g,h}
- Page 2 = {i,j,k,l}
- Page 3 = {m,n,o,p}





Translating:

- There is the same prefix for virtual that corresponds to physical
- Remaining bits are for the offset
- MATHCRAFT (32 bit addressing)
  - #addresses = 2^{32}
  - page size = 4 KB = 2^{12}
  - #pages = $(2^{32})/(2^{12}) = 2^{20}$
  - offset = 2^{12} bits



Page Table Implementation

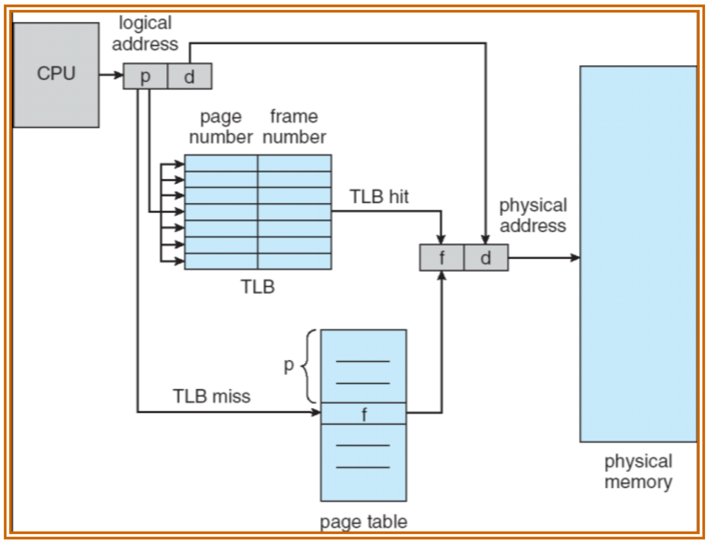
- Page table is kept in main memory (usually)
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses.
    - One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

TLB is associative memory:

Associative memory – parallel search

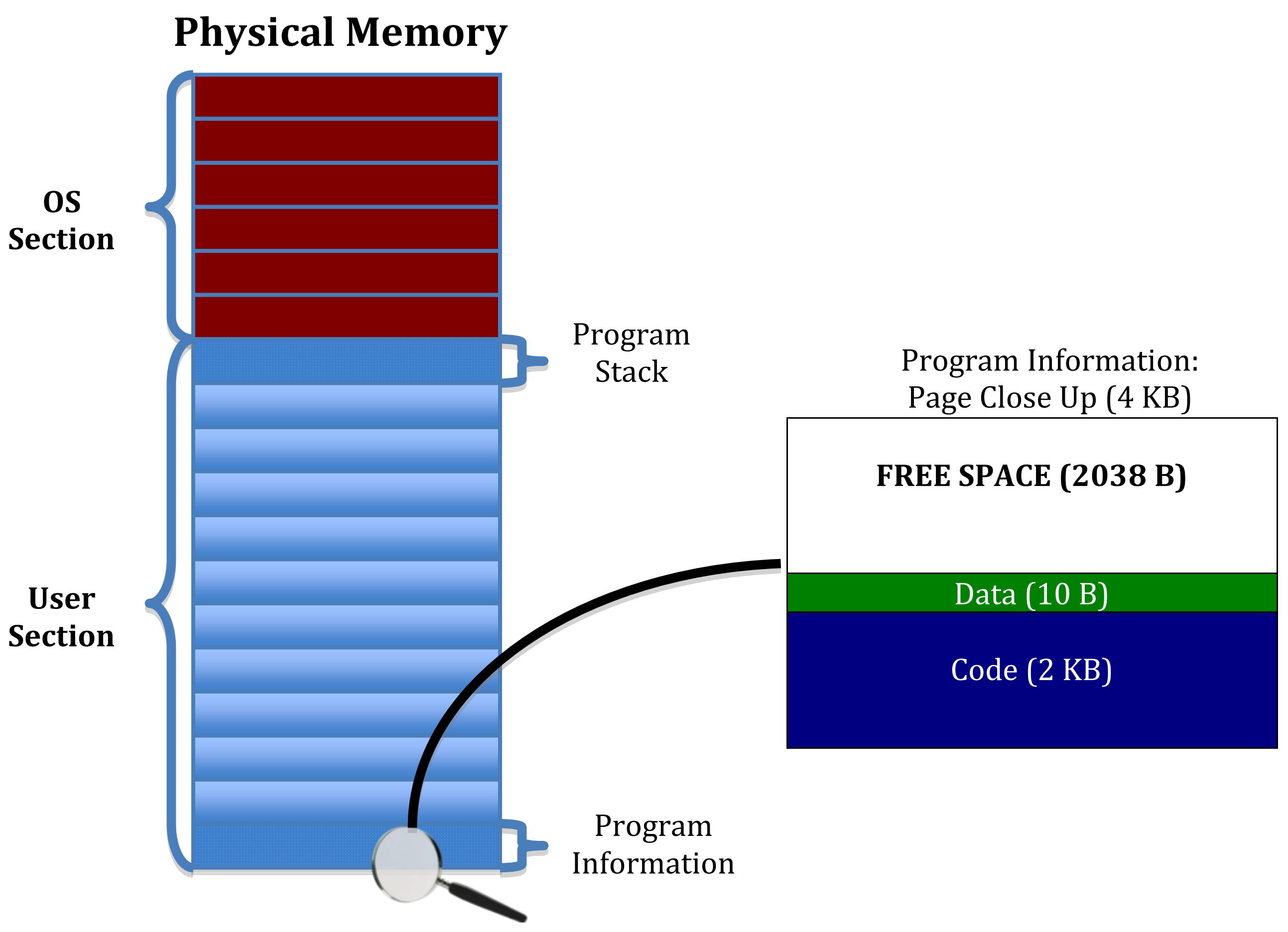
Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory



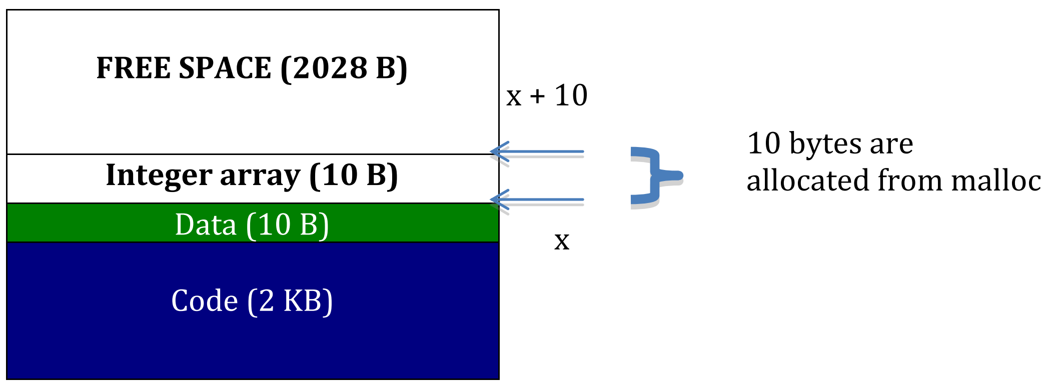
Consider the following:

Page size = 4 KB
Program's code = 2 B
Program's data = 10 B



What does the program's information page look like after we call

```
    int a* = malloc(10);
```



After `int a* = malloc(10);` is called, consider the following code:

```
int i;
i = a[11];
```

We were asked if this buggy code would cause a segmentation fault, since it attempts to access memory that has not yet been allocated. The answer is no, but only for this specific situation. **A segmentation fault occurs when a program tries to access memory from a page that that specific program has not allocated**. In our situation, a[11] tries to access unallocated memory, but the additional byte it reaches for is still within the boundaries of the page that was designated for the program it is running in. Though poor programming, this code would not lead to a segmentation fault in this situation.