COMP1001

Computer Systems

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk

Website:
https://www.plymouth.ac.uk/staff/vasilios-kelefouras

Outline

- Memory management
- Memory layout of programs
- Stack
- Heap
- Virtual Memory
- Fragmentation
- Swapping
- Paging System
- TLBs

What is Memory Management ?

- **Is the task carried out by the OS and hardware to accommodate multiple processes in main memory**
  - This makes sure that a program does not conflict with memory currently being used by another program
    - Keep track of which parts of memory are currently being used & by what
  - Map processes to memory locations
  - Allocate/deallocate memory space as requested/required

- The size of main memory is limited
- In most schemes, the kernel occupies some fixed portion of main memory and the rest is shared by multiple processes

Memory Management
Introduction

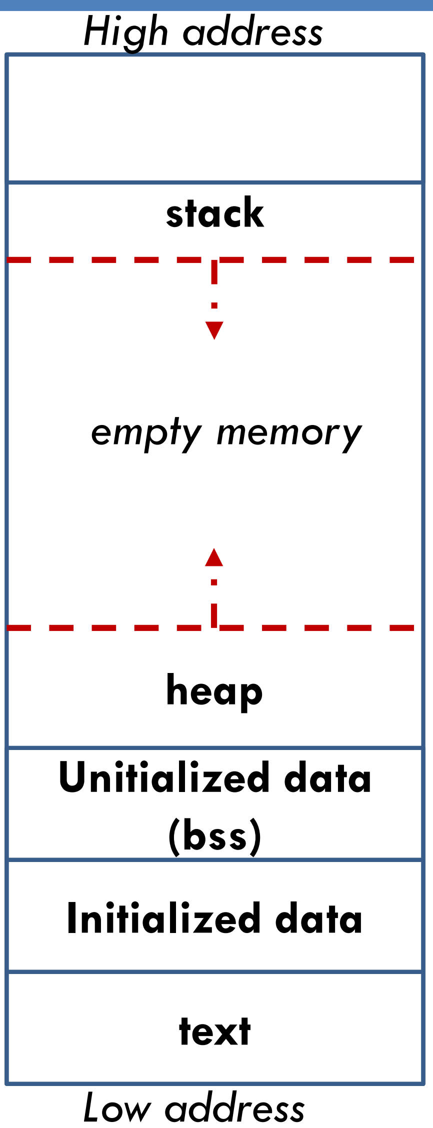
- Memory management is a form of resource management applied to computer memory
- **Each process has its**
  - **source code**
  - **Data**
  - Both must be stored into memory. The CPU reads instructions from memory as well as reads/writes data from/to memory
- For a program to execute as a process, it has to be loaded into memory via a program loader
  - The executable file that contains the program consists of several sections / regions of the program and where they should be loaded into memory, e.g.,
    - regions for executable code, initialized variables, zero-filled variables, dynamically allocated data, etc

Linking

- Complete programs are rarely compiled from a single file of code. Normally, the program uses
  - functions that are in separate files
  - Libraries
- The separately compiled files are linked together to create the final executable file
- The process above is called linking

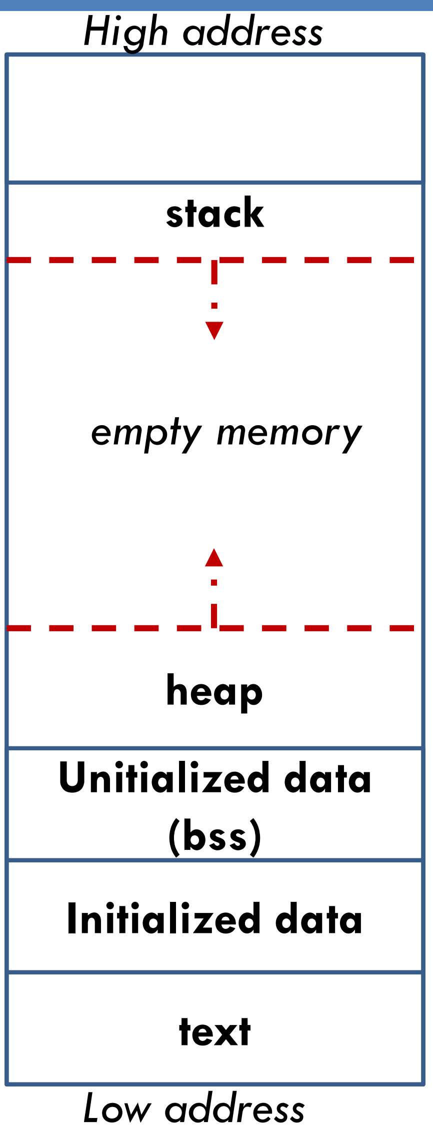
Memory Layout of C Programs

- A program refers to the code that is stored in a file.
- A process is a program plus its execution context. This includes
  - The process's memory map, which identifies the various regions of memory that have been allocated to the process
- Before the program starts its execution it needs memory for its code and (un)-initialized data.
- When the program is loaded into memory, additional space is needed for dynamically allocated data and the stack



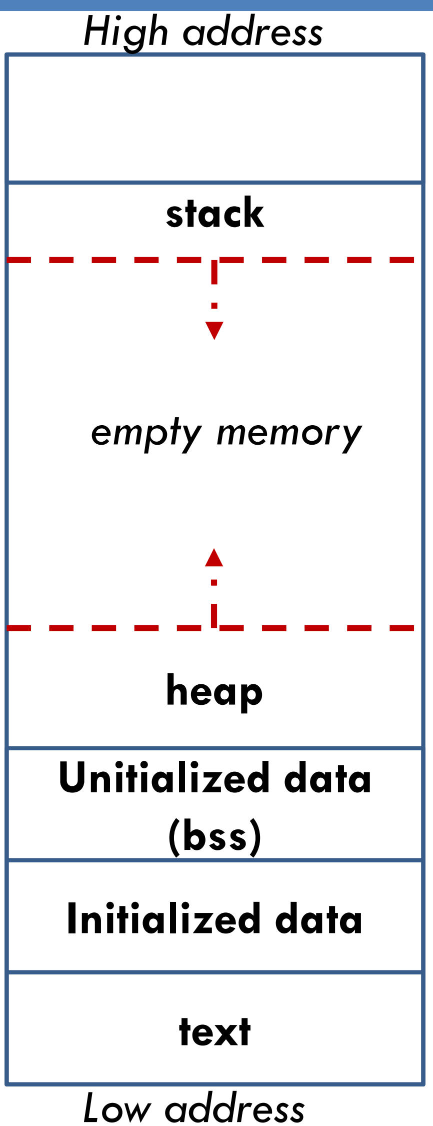
Memory Layout of C Programs

- **Text** : contains the compiled code (binary) – read only
- **Initialized data** : contains the global and static variables that are initialized by the programmer
- **Uninitialized data** : contains the global and static variables that are initialized to zero or they do not have explicit initialization
- **Stack:** it is a last in first out (LIFO) structure that stores temporary variables created by each function (including main).
  - Every time a function is called, ***its local data, function arguments and return values*** are pushed into the stack
  - Every time a function exits, all its local data are freed (popped from the stack).
  - This is done automatically
  - Limited size – normally its default value is 1Mbyte.



Memory Layout of C Programs (2)

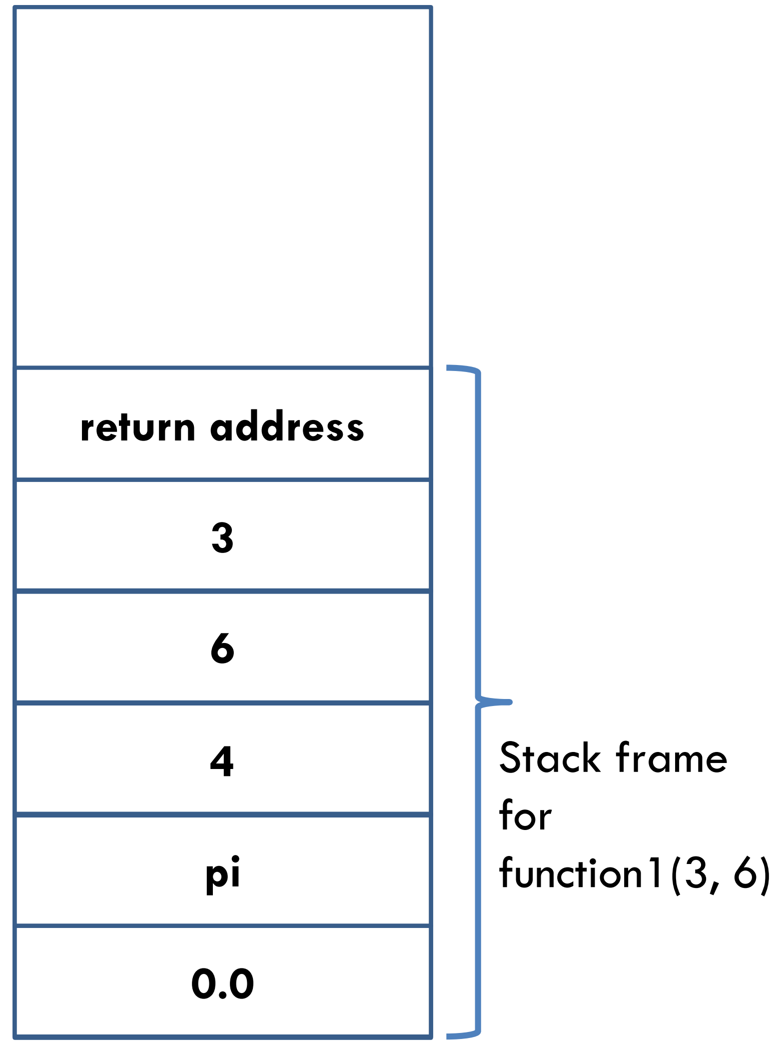
- **Heap** : contains all the data allocated dynamically
  - This is done by the programmer using functions like ***malloc, calloc, realloc*** etc
  - Once we have allocated memory in the heap, we are responsible for deallocating this amount of memory too using ***free()*** function
    - Otherwise, a ***memory leak*** occurs (why?)
  - Unlike stack, its contents are accessible by any function
  - No size restrictions – depends on your DDR memory size



Stack - Example


```
int function1(int i, int t){
int g=4;
float pi=3.14159265;
float res=0.0;
//...
return res;
}

int main(){
//...
int out=function1(3, 6);
//...
return 0;
}
```

When function1() exits, its stack frame will be deallocated



Why Stack is important?

- The stack allows our system to use
  - Nested subroutines
  - Function local variables

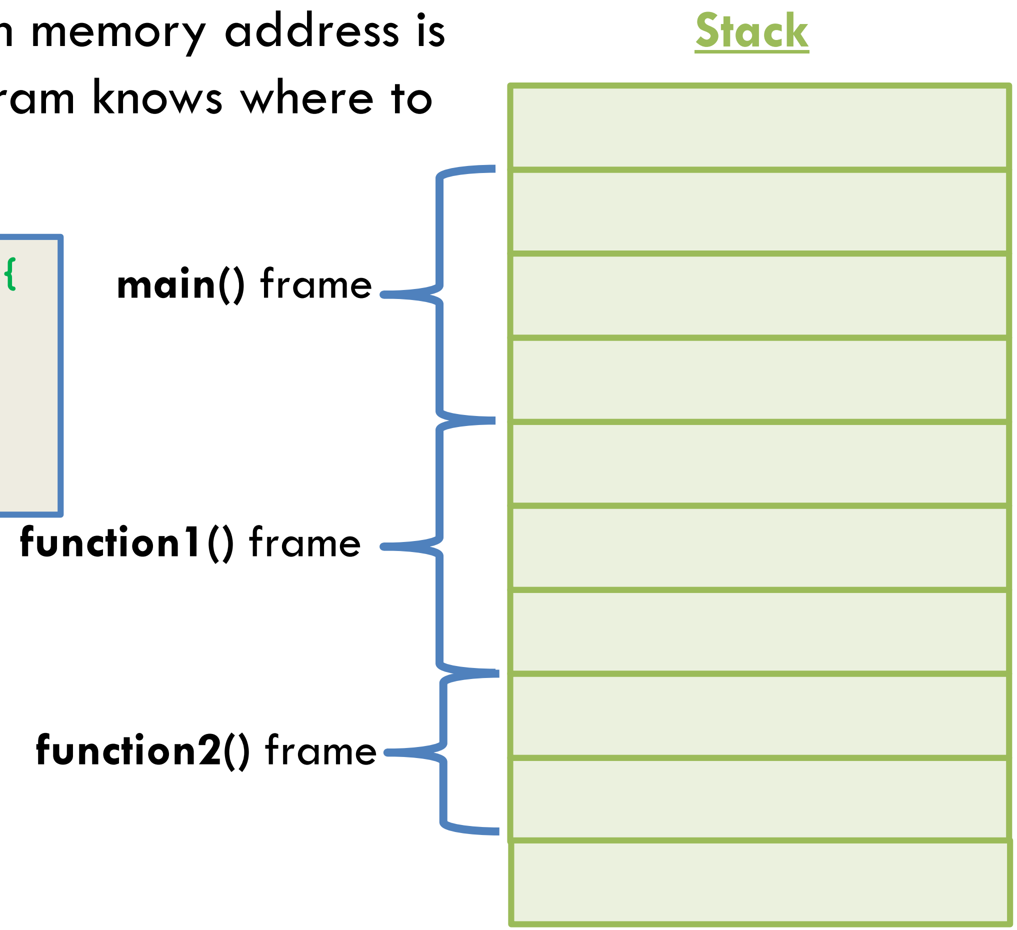
Stack – Nested Subroutines

- The CPU uses the program counter register to know where to find the next instruction
- When a function is called, the return memory address is stored into the stack so as the program knows where to go when the function ends

```
int main (){

...
function1();
...
}
```

```
void function2(){

...
}
```

```
void function1(){

function2();
...
}
```



Why should a programmer care about the stack?

- Stack memory is limited
- We cannot define large data structures inside functions

```
int main (){

...
function1();
...
}
```

```
void function1(){

int A[1000000];
...
}
```

This program will crash
Why?

Heap related functions
malloc

- *void * malloc (size_t size);*
  - It allocates a block of memory on the heap
  - Its input is the amount of memory to be allocated in bytes
    - *size_t is an unsigned int data type which is defined in several header files*
  - '*void **' is a pointer to everything (its data type will be given later using a type cast)
  - *int * array = (int *) malloc (4 * sizeof(int));*

*void * malloc (size_t size);*

Heap related functions
calloc

- ***void * calloc (size_t num, size_t size);***
  - It allocates a block of memory on the heap and **initializes to zero**
  - Its 1st input is the number of elements to allocate
  - Its 2nd input is the size of each element
  - ***int * array = (int *) calloc (4 , sizeof(int));***

Heap related functions
Preventing bugs

- **Malloc/calloc cannot always to allocate the memory we asked**
  - Thus, the following code is needed to prevent bugs

```
int * array = (int *) malloc (4 * sizeof(int) );
If  (array == NULL){
  printf("Memory is not allocated");
  return -1;  //inform the function that something bad happed
}
```

Heap related functions
realloc

- ***void * realloc (void *ptr, size_t size);***
  - **It reallocates** a given area of memory
  - It must previously allocated by malloc, calloc or realloc and not yet freed. Otherwise the results are undefined.
  - **Its 1st input is a pointer to the memory area to be reallocated**
  - **Its 2nd input is the new size of the array in bytes**
  - It maintains the already present values and the new blocks are initialized with garbage value.
  - ***int * array = (int *) calloc (4 , sizeof(int));***

    int * array = (int *) realloc (array, 8 * sizeof(int));

Heap related functions
free()

- When we do no longer need the memory dynamically allocated, then we should deallocate it
- ***void free (void * ptr)***
  - The input argument is the pointer to the memory block previously dynamically allocated
    - Failure to deallocate the memory leads to memory leaks
    - Memory leaks waste memory resources and can lead to allocation failures and bugs
    - A tool to debug memory leaks is valgrind

Activity – Task1

- See notes

Stack and Heap Crash

- How can we make the heap crash?
- How can we make the stack crash?

Activity – Task2-3

- See notes

Positioning code and addressing memory

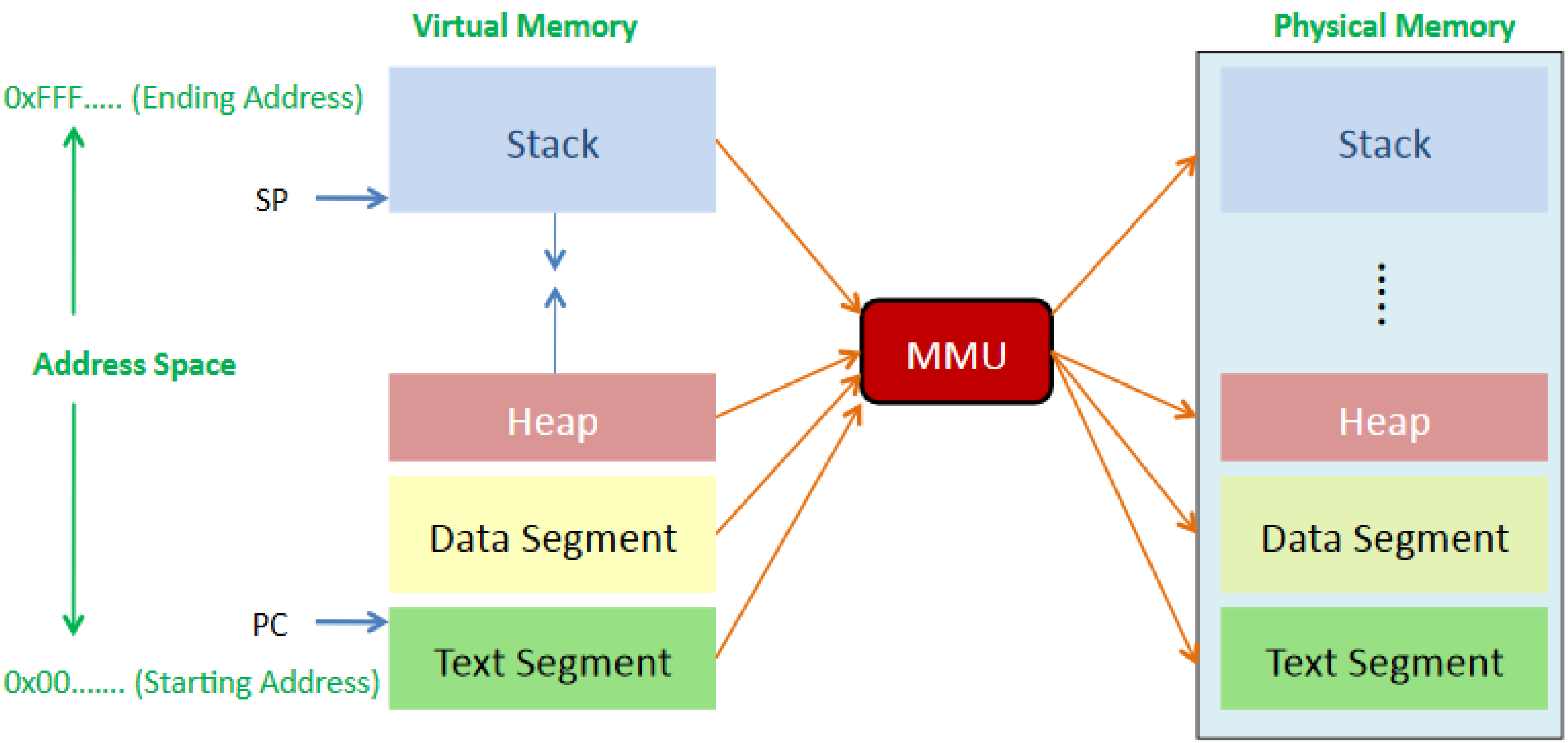
- Multiple programs are loaded into memory at the same time
- The OS periodically switches the processor’s execution state from one process to another

- ***Since programs use actual memory addresses, they need to make sure that the aforementioned memory address will be used, no matter where the program is positioned into memory***
  - How will this happen?
    - By using Logical addressing

Logical Addressing

- Recall from assembly that in order to load/store we need to refer to a memory location but
  - Physical memory is limited, normally 4-16Gbytes
  - A computer must be able to address more memory (hard disc)
- This is why **programs use virtual memory addresses and not physical**
  - The ***OS*** and the ***Memory Management Unit (MMU)*** are responsible for translating a virtual address to a physical
  - Virtual memory is divided into pages, normally 4kbytes each
  - Moreover, using virtual addresses allows to have **memory protection**
    - Virtual memory allows the system to give every process its own memory space isolated from other processes - processes cannot easily interfere with each other

Virtual and Physical Memory



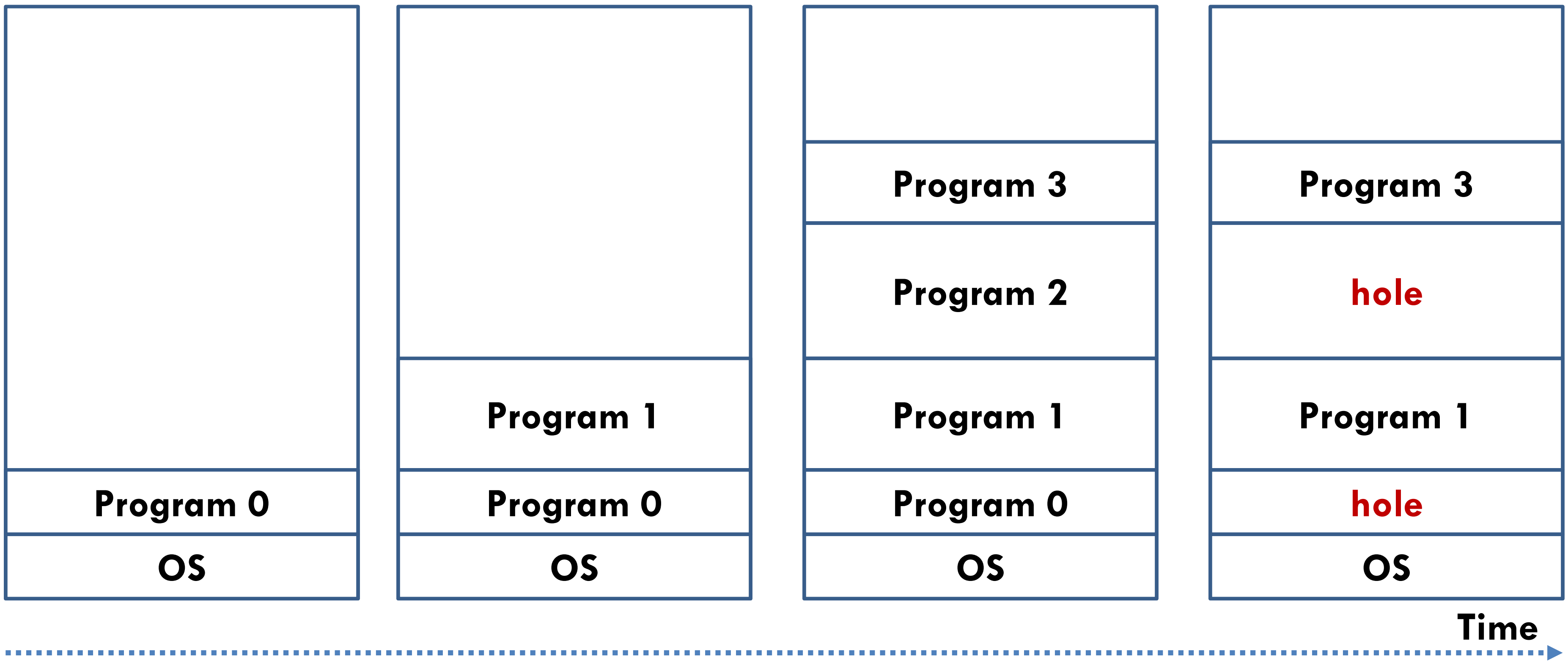
The memory addresses we see in assembly or C refer to virtual memory addresses

Virtual Memory

- Virtual memory enables programs to execute without requiring their entire address space reside in physical memory. This provides ***benefits***:
  - **Utilizes main memory size**
    - Many programs do not need all of their memory at once (or ever), so there is no need to allocate memory for it
  - **Flexibility**
    - Programs can loaded/evicted from/to memory in a dynamic way
  - **Security**
    - A process cannot access the memory of other processes (there are some exceptions)
- ***Challenges*** also arise such as the overhead of
  - context switch (page swapping is very slow as the slow disc is accessed)
  - Address translation from virtual to physical
  - **This is why hardware support is needed - MMU**

Fragmentation (1) (older systems)

- Multiple processes are stored into memory at the same time
- Memory partitions are created dynamically
- The OS allocates to each process the amount of memory required
- When processes exit, they create memory holes

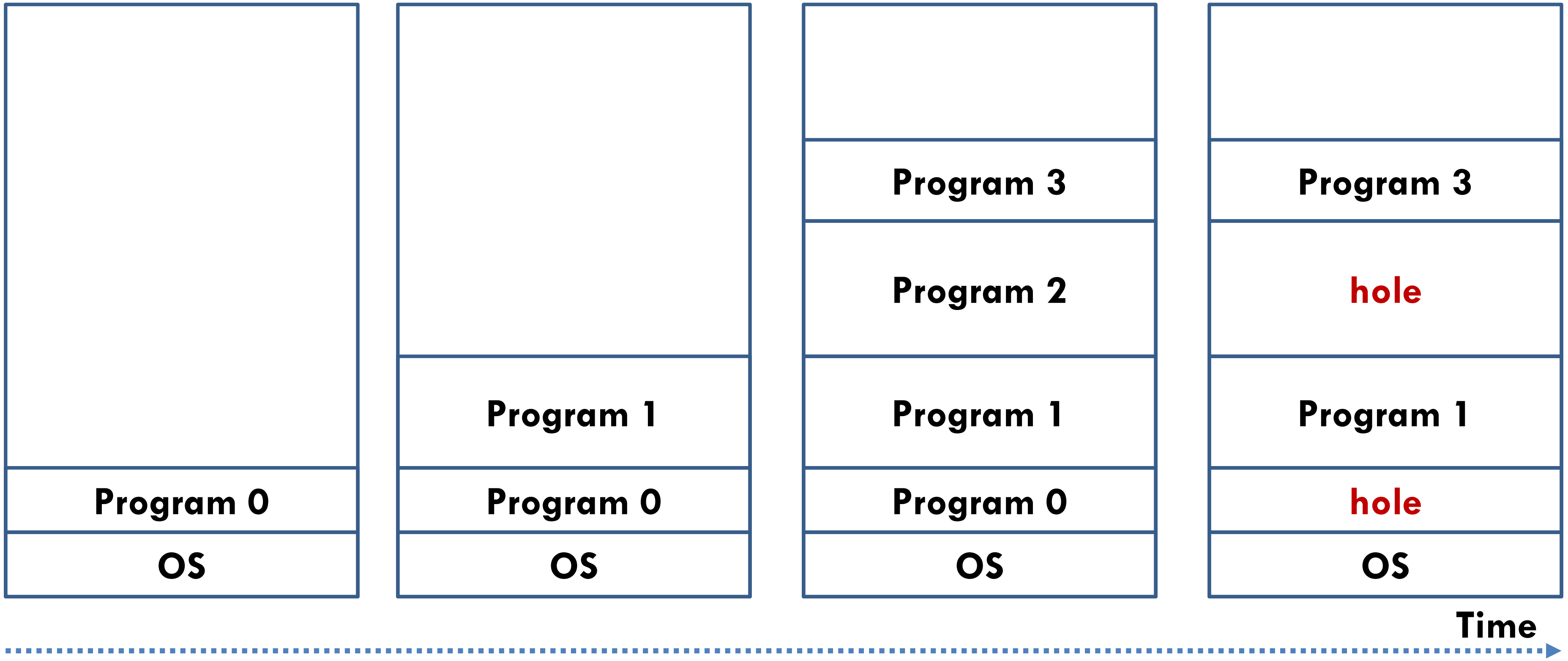


User Space and System Space

- When a CPU reads/writes from/to memory, it uses virtual addresses
- The memory management unit will translate the virtual addresses to physical addresses
  - This gives many advantages
- **Each user mode process, has its own private virtual address space**
- All the processes that run under kernel mode, share a single virtual address space, called system space
- Each process feels like it owns the full address space

Fragmentation (2) (older systems)

- When processes exit, they create memory holes
  - The OS tries to fit new processes to the available holes
  - However, a new smaller hole will be created
    - this problem is also known as ***fragmentation***

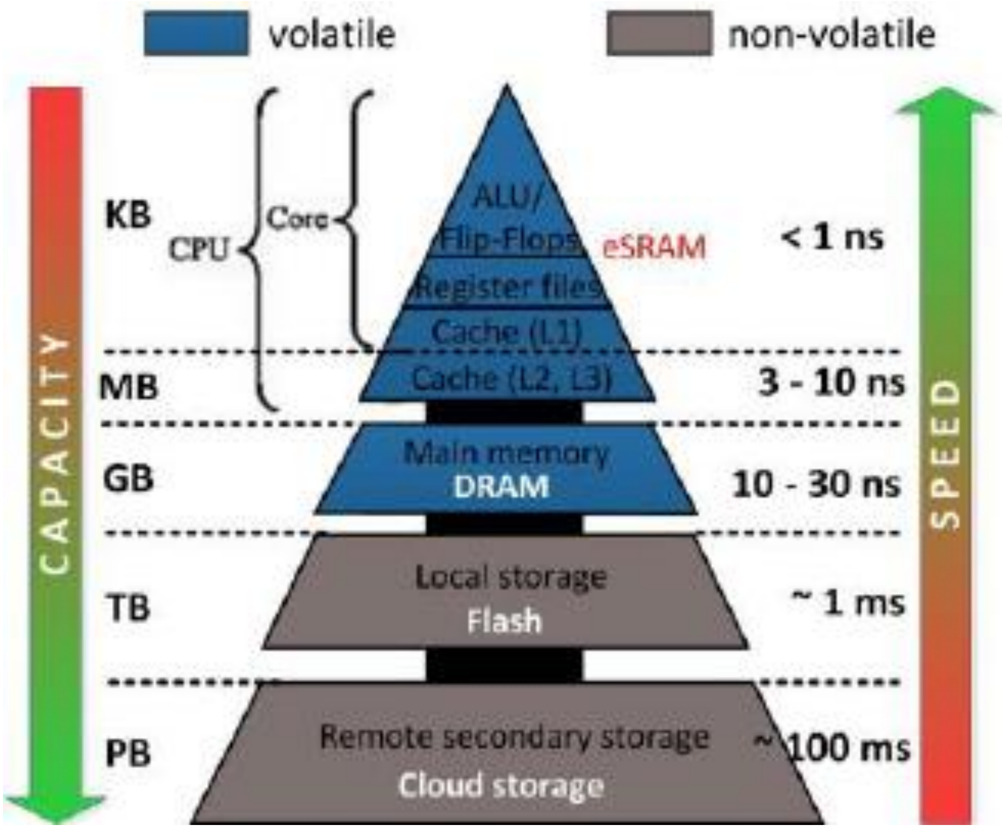


Growing processes

- Processes may increase their memory at runtime.
- If a process is located next to a hole (unusued memory), then the memory manager will use that
- **What if there is no unused memory next to the growing process?**
  1. If there is a hole with the desired memory space then we can relocate the process into that hole
     - This is known as **memory compaction** and it introduces an overhead
  2. If there is no hole with the desired memory space then some processes will be swapped out onto the disk to create space into memory
  3. If there is no swap space left on the disk and there is no large enough hole, then the process will be terminated
  - For this reason, the OS normally allocates extra space in memory for each process

Swapping

- **Swapping**
  - A process needs to be in main memory in order to be executed
  - But sometimes there is not enough main memory to hold all the currently active processes
  - So, process are kept on disk and brought in to run dynamically.
  - **Swapping is the process of bringing in each process in main memory, running it for a while and then putting it back to the disk.**
  - **This is a very slow process**



Page-based Virtual Memory (modern systems)

- **Assigning a memory partition to each process is problematic because**
  - of the fragmentation problem discussed in the previous slides
  - Processes are limited to the amount of physical memory available
  - The number of processes in memory cannot exceed the size of the available memory
  - If we need to run more processes, we must swap the contents of some processes out to the disk

- The solution to the above problem is ***page-based memory management***

Page-based Virtual Memory

- Physical and Virtual memory are divided into equal-sized memory blocks.
  - The blocks of physical memory are called ***frames***
  - The blocks of virtual memory are called ***pages***
  - Their size is always a power of 2
  - In Linux, the default size is 4kbytes

- A page may be placed to any available frame
- **When a process is swapped in, only the pages that are expected to be needed right away are loaded into memory.**

Page-based Virtual Memory

- **The Paging technique**:
  - Allows for **keeping just the parts of a process that we are using in memory and the rest on the disk**
  - Does not limit the size of the process to the size of the physical memory
  - Allows memory allocation to be non-contiguous (frames are not written into consecutive memory locations) and thus **no fragmentation occurs,** simplifying memory management and obviating any need for compaction
  - Allows to keep more processes in memory than the sum of their memory requirements

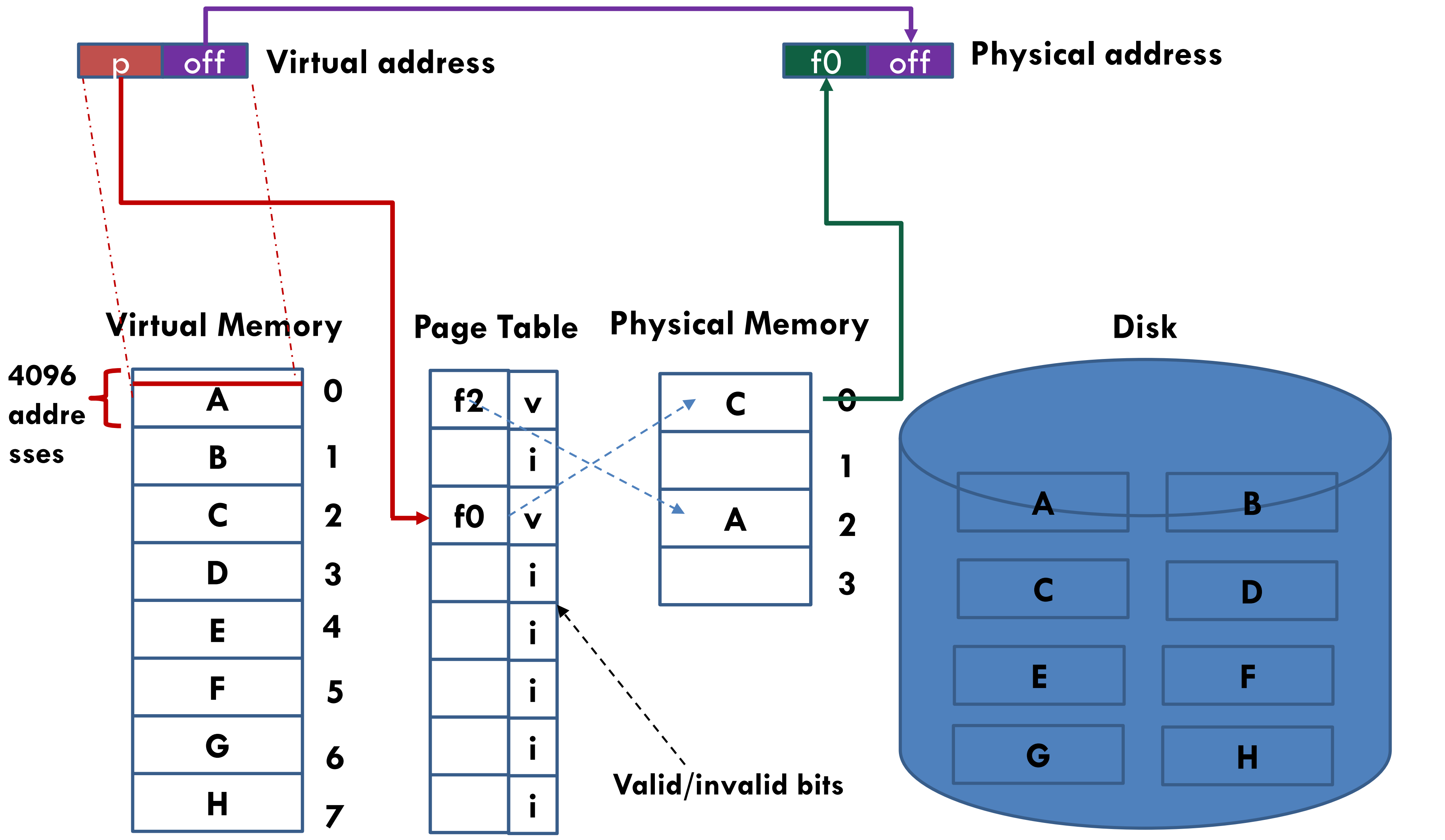
Page-based Virtual Memory

- **The memory management unit is responsible to map a page to a frame**
- A fixed number of bits in the memory address identifies the **page number** and a fixed number of bits identifies the **offset** (the specific block inside the page).
  - Consider that the page/frame size is 4Kbytes, i.e., 12 bits are needed as $\log_2 4096=12$.
  - In a 32bit OS, the virtual memory addresses are of 32 bits
  - The 20 most significant bits are used to find the page needed, while the 12 least significant bits are used as an offset
  - The address 0x3CC5 will be stored into page 3. Why?

Virtual address decomposition

00000000000000000011	110011000101
*Page number, **p** - 20bits*	*Offset, **off** - 12bits*

How Page-based Virtual Memory Works? (1)
Simple approach – older systems



How Page-based Virtual Memory Works? (2) Simple approach – older systems

- Let's say the CPU needs the datum/instruction in virtual address ***v=(p, off)***
- The memory management unit (MMU) will look in the page table using 'p' as an index
  - The **page table** is an array of **page table entries (PTE)**
- The PTE also contains a valid/invalid bit indicating whether the page is currently mapped onto a frame.
- The physical location of the data/instruction we are looking for, is at offset 'off' in frame 'f0' – the address that is the concatenation of f0 and off
- If the data needed are in a page that is not mapped to a frame in memory, a **page fault** occurs and the OS is responsible for either swapping in the page needed or killing the process (this is not very likely)
  - **Page faults introduce a high overhead in performance**
- Any page can be mapped into any frame and the page table keeps the mapping, providing the illusion of one contiguous block of memory

How Page-based Virtual Memory Works? (3) Simple approach – older systems

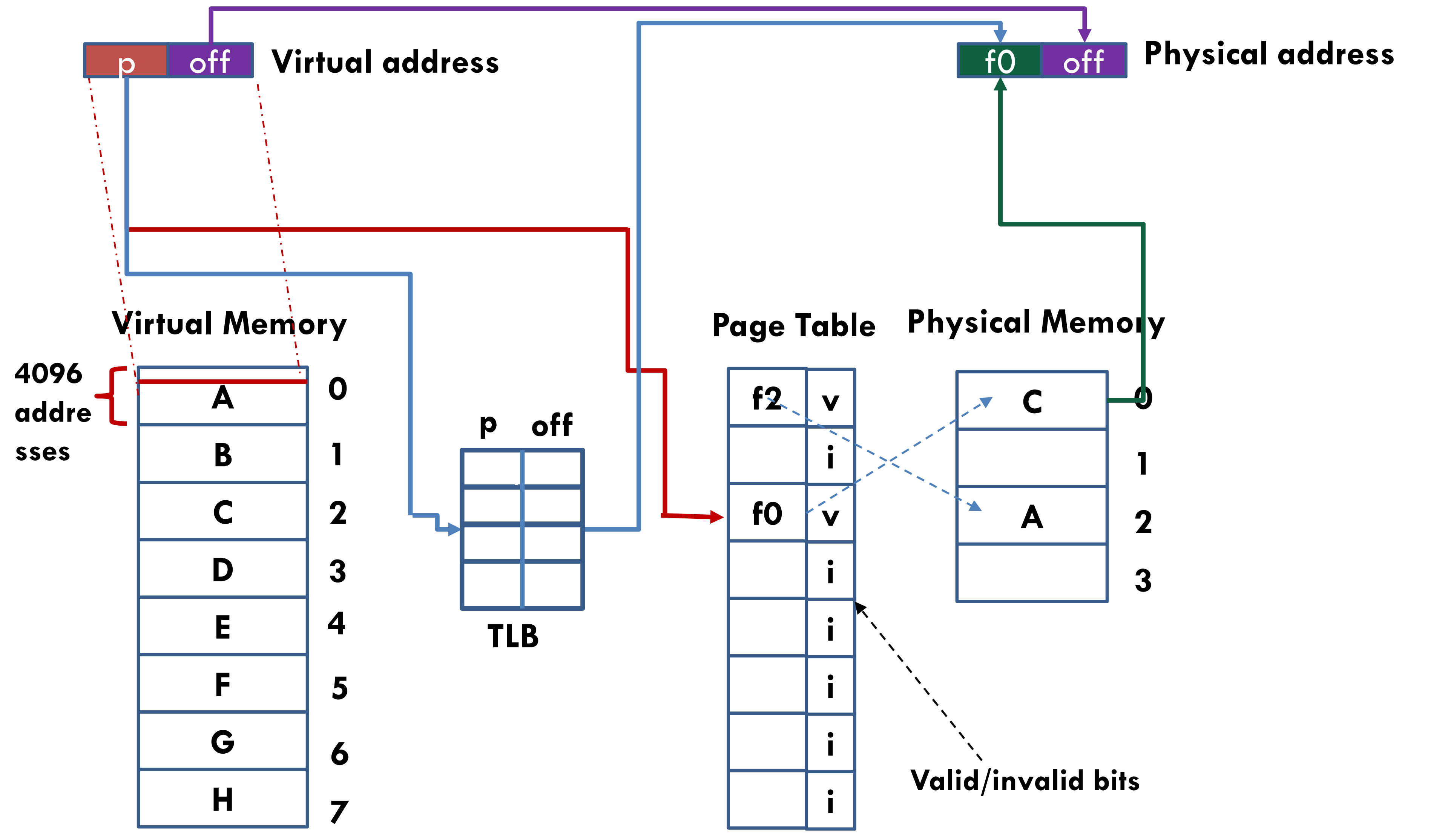
- The approach described contains an entry for every page.

- Unfortunately, this approach has the **following drawbacks**
  - **the page table is very large – 1 million entries** are needed for an address space of 32Gbytes and 4Kbyte pages
  - We need a page table per process
  - Thus, we need to store the page table in physical memory
    - So, for a single load instruction, we access the physical memory twice, once for accessing the datum itself and another for accessing the page table
    - This approach is very slow

How Page-based Virtual Memory Works? **Modern Systems** – TLBs (1)

- The answer to the previous problem is to use a set associative cache memory called ***Translation Lookaside Buffer (TLB)***
- TLBs (caches) are high speed small memories, but they are expensive
- The **TLB contains just the frequently accessed page table entries only**
- TLB is a cache but instead of storing data, it stores page table entries
- **If the CPU needs to access a datum/instruction from memory, it first look for its address in the TLB**
  - Then it is either a cache hit or a cache miss
  - In a cache miss, one TLB entry is removed and replaced by the entry that was just looked up so that the next reference to that page will result into a hit

How Page-based Virtual Memory Works? Modern Systems – TLBs (2)



Questions

- **What if the page size is very large?**
  - Internal fragmentation: the memory space of each frame is likely to be wasted
- **What if the page size is too small?**
  - Many page table entries are needed, consuming more memory
  - Not efficient TLB usage
- **Can frames be shared among two or more processes?**
  - Of course they do
    - A virtual memory system allows each process to have its own private address space
    - Having a page table for each process, the OS can share frames between two or more processes
    - This reduces the overall memory space used

Multiple Levels of TLBs

- Modern processors have multiple Levels of TLBs
  - For example, Intel Core i7-6700 has
    - 1^{st} level of TLB for data
    - 1^{st} level of TLB for instructions
    - 2^{nd} level of TLB for both instructions and data

Further Reading

- Chapter 7 and Chapter 8 in Operating Systems book, available at https://dinus.ac.id/repository/docs/ajar/Operating_System.pdf

Thank you