CSC 230 - Summer 2018

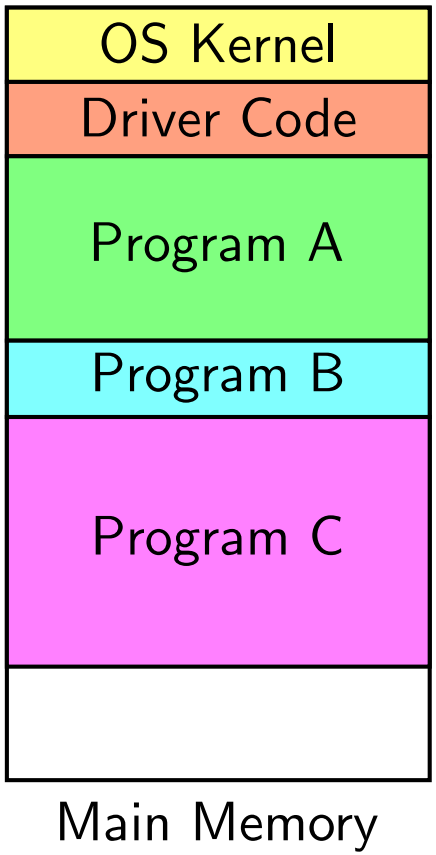
Virtual Memory

Bill Bird

Department of Computer Science
University of Victoria

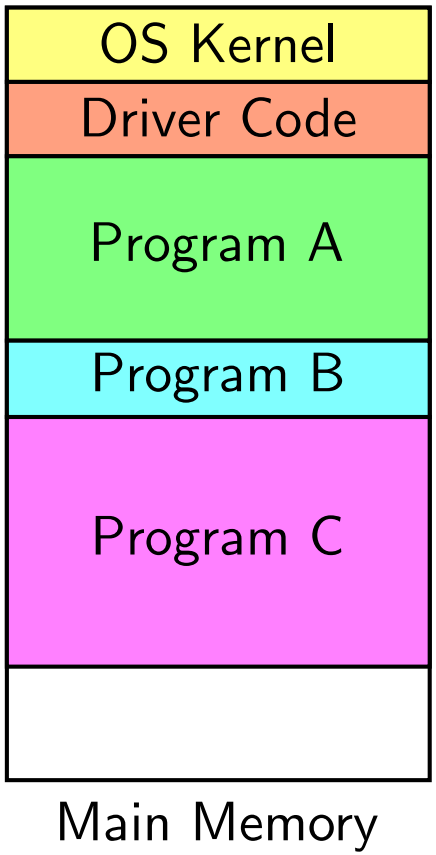
July 22, 2018

Multitasking (1)



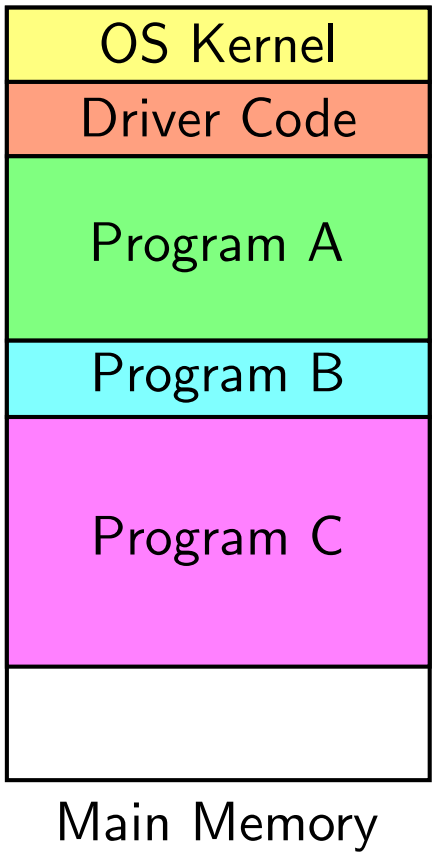
In a **multitasking** operating system, several programs can be active simultaneously, with the operating system mediating the resource requirements of the various programs.

Multitasking (2)



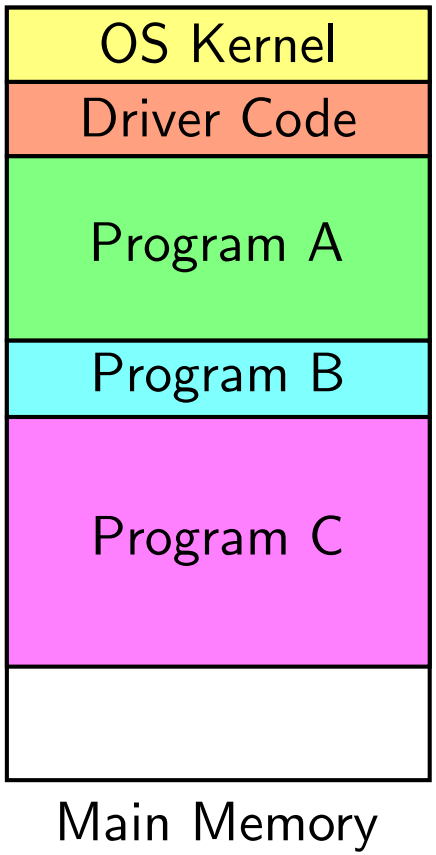
When a particular program is compiled, it is impossible to know exactly where in memory the program will be positioned when it is run.

Multitasking (3)



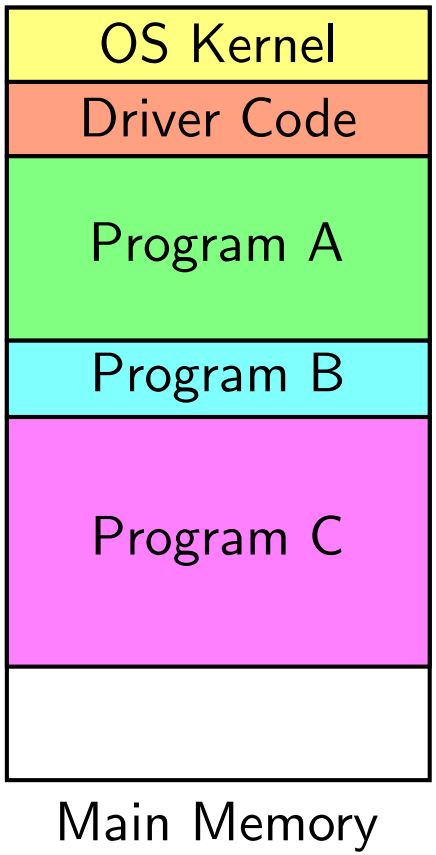
If each program occupies a contiguous block of memory, it is relatively easy to generate **relocatable code** which allows the program to be positioned anywhere. For example, a base address can be added to every memory address in the code to reposition it.

Multitasking (4)



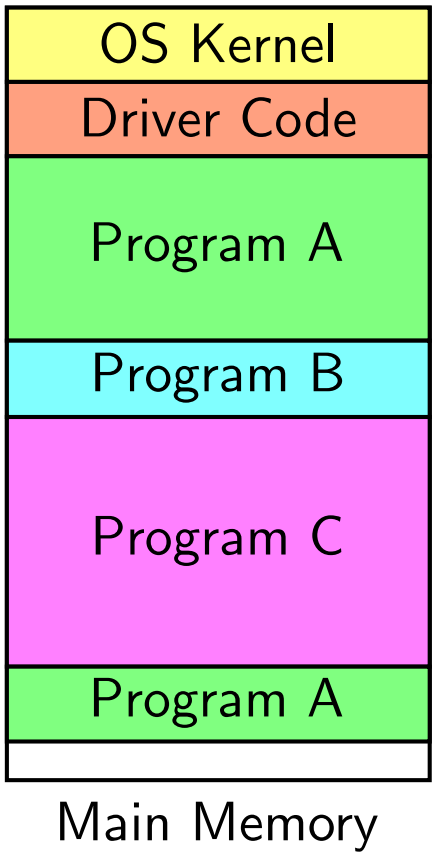
Question: What happens if Program A requests more memory?

Multitasking (5)



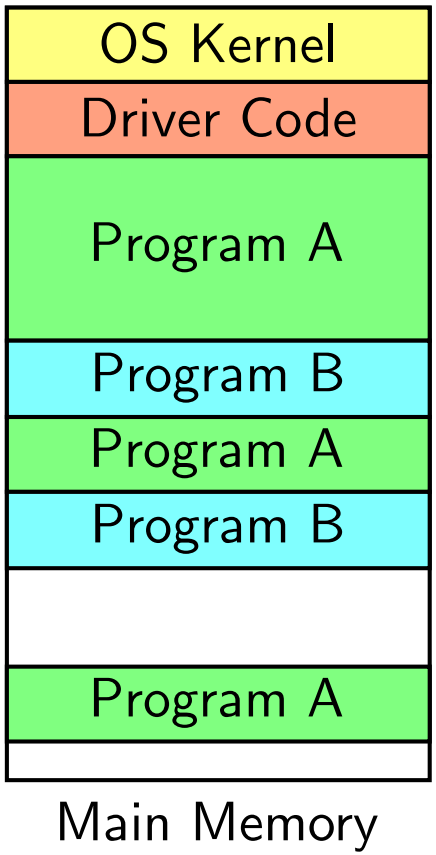
Question: What happens if Program A requests more memory?

Multitasking (6)



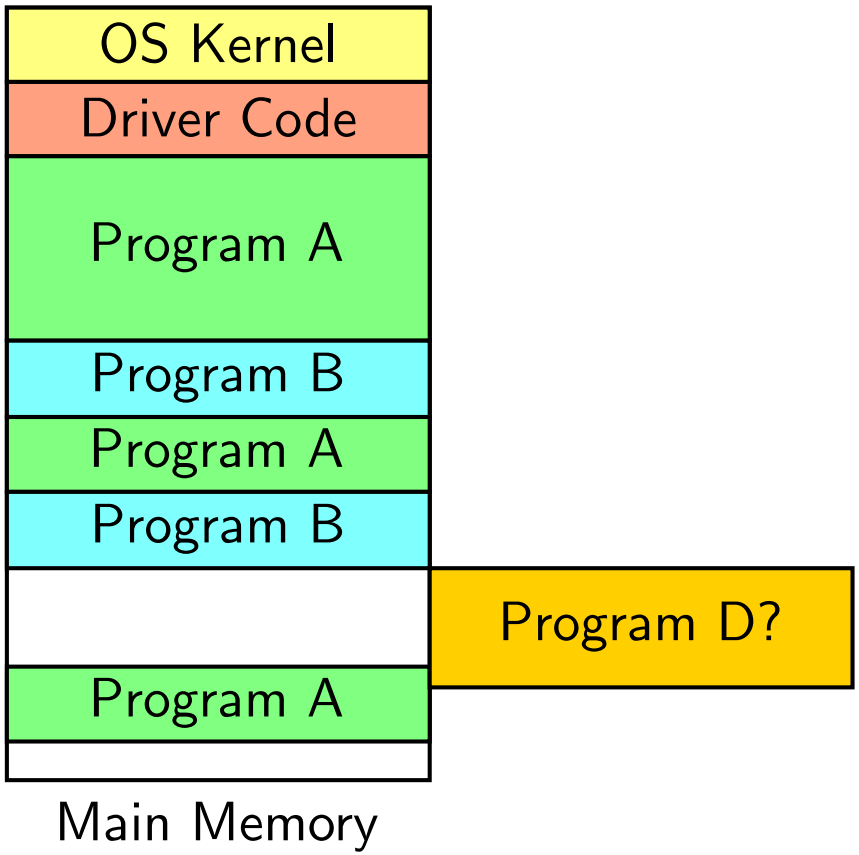
The only available memory is not contiguous to Program A's existing allocation, but a new allocation can be created.

Multitasking (7)



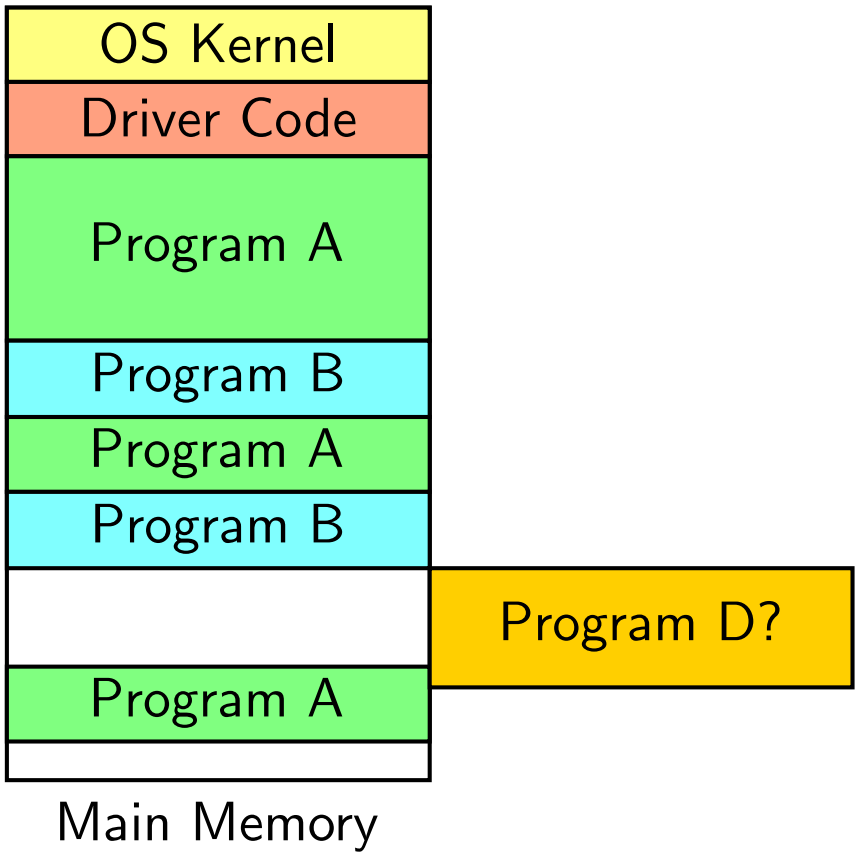
Over time, as programs finish and other programs ask for new storage, the memory can become increasingly fragmented.

Multitasking (8)



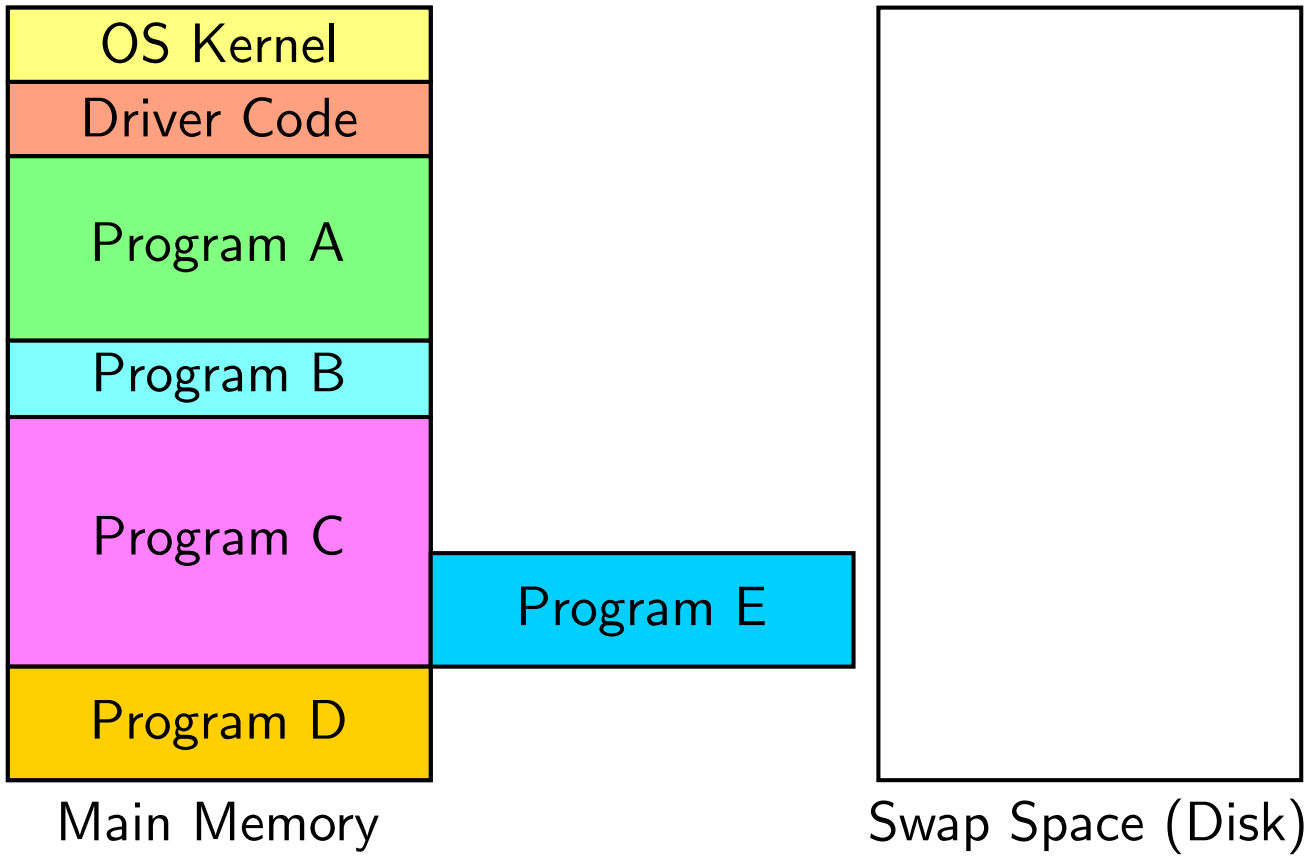
Problem: Suppose Program D is run, with the initial allocation shown. Although there is enough memory to hold Program D, the fragmentation prevents Program D from being contiguous.

Multitasking (9)



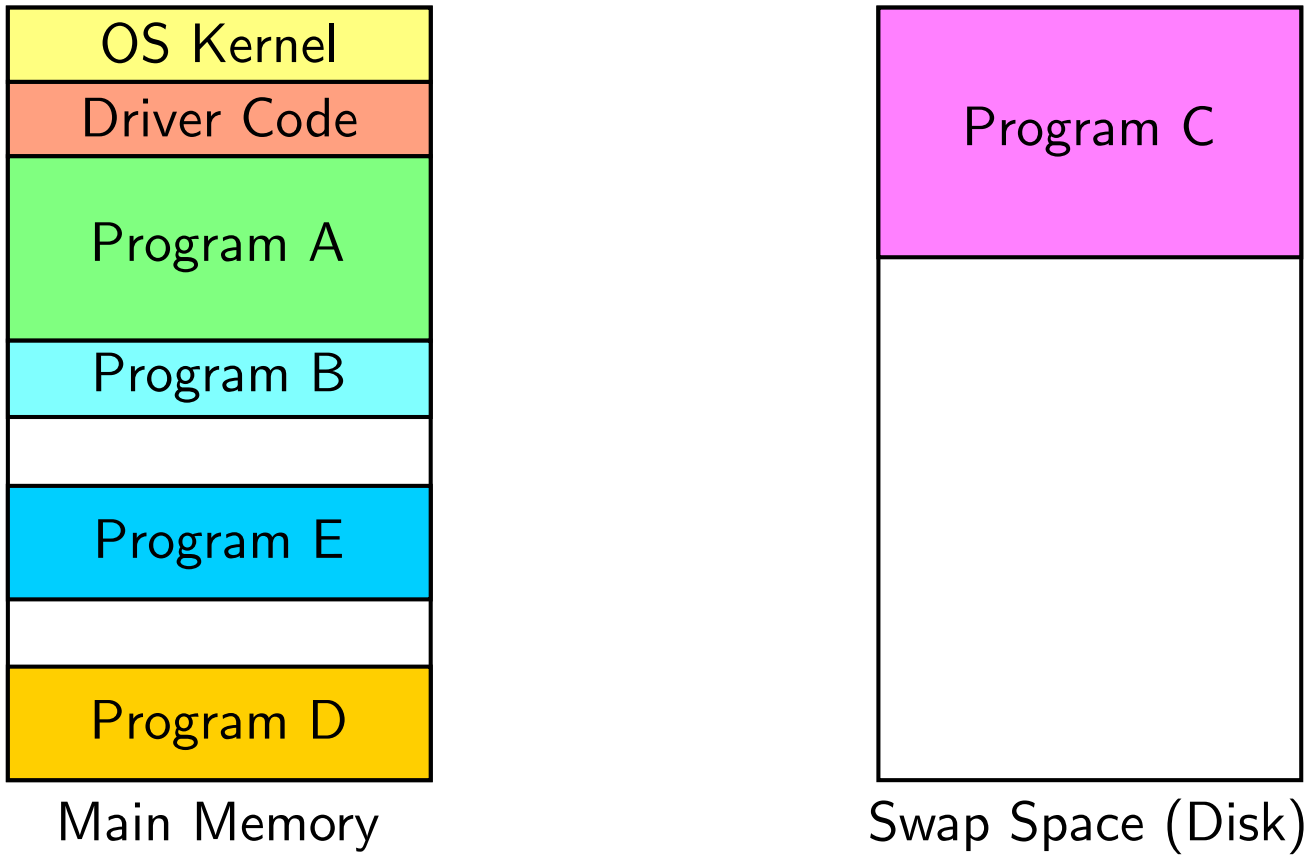
Often, the actual layout of memory is not amenable to the internal layout needed by programs. It can be argued that the 'geography' of the memory space is irrelevant to each program, since each program is only allowed to use its own space.

Multitasking (10)



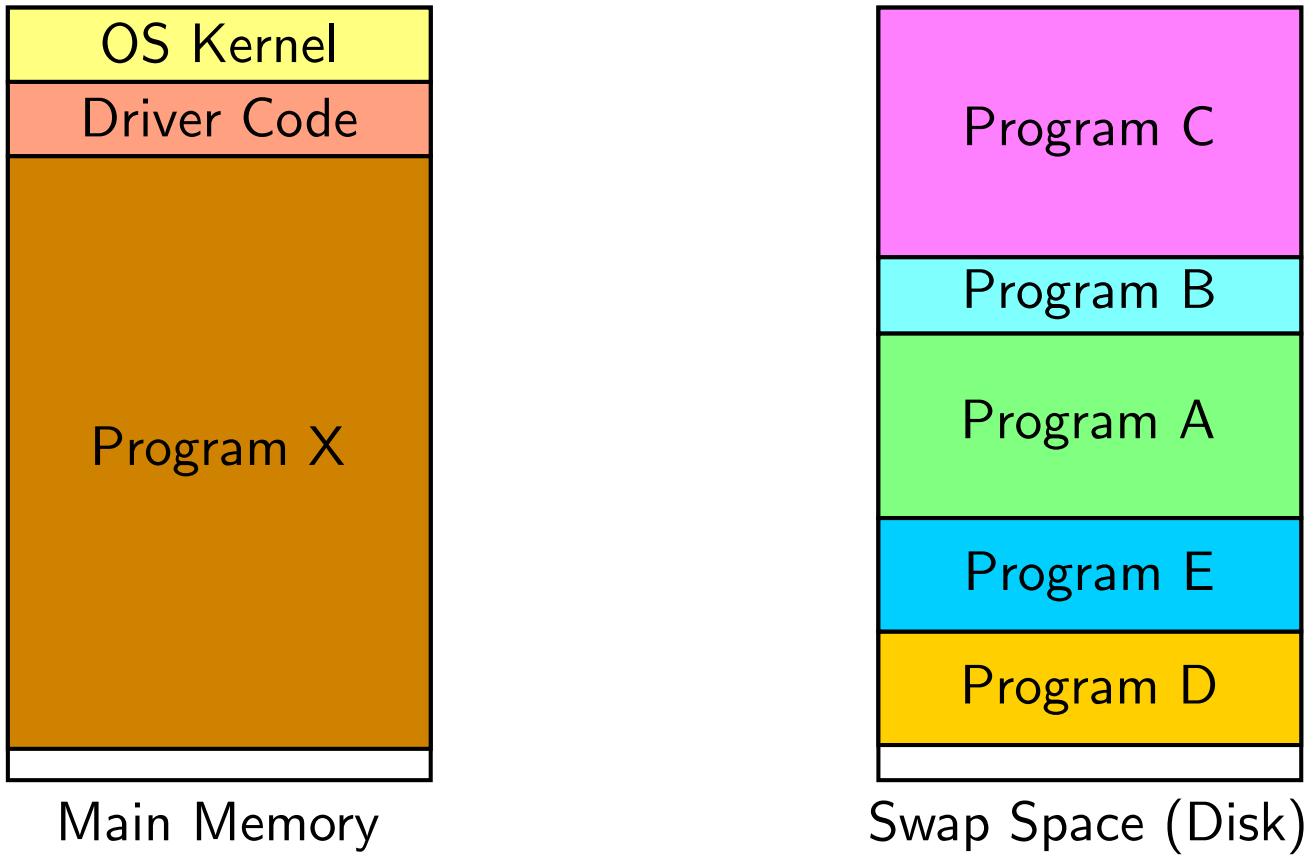
Another Problem: Suppose Program E is run in the system above. Even though the size of the main memory is sufficient to accommodate Program E, other programs are occupying all available space.

Multitasking (11)



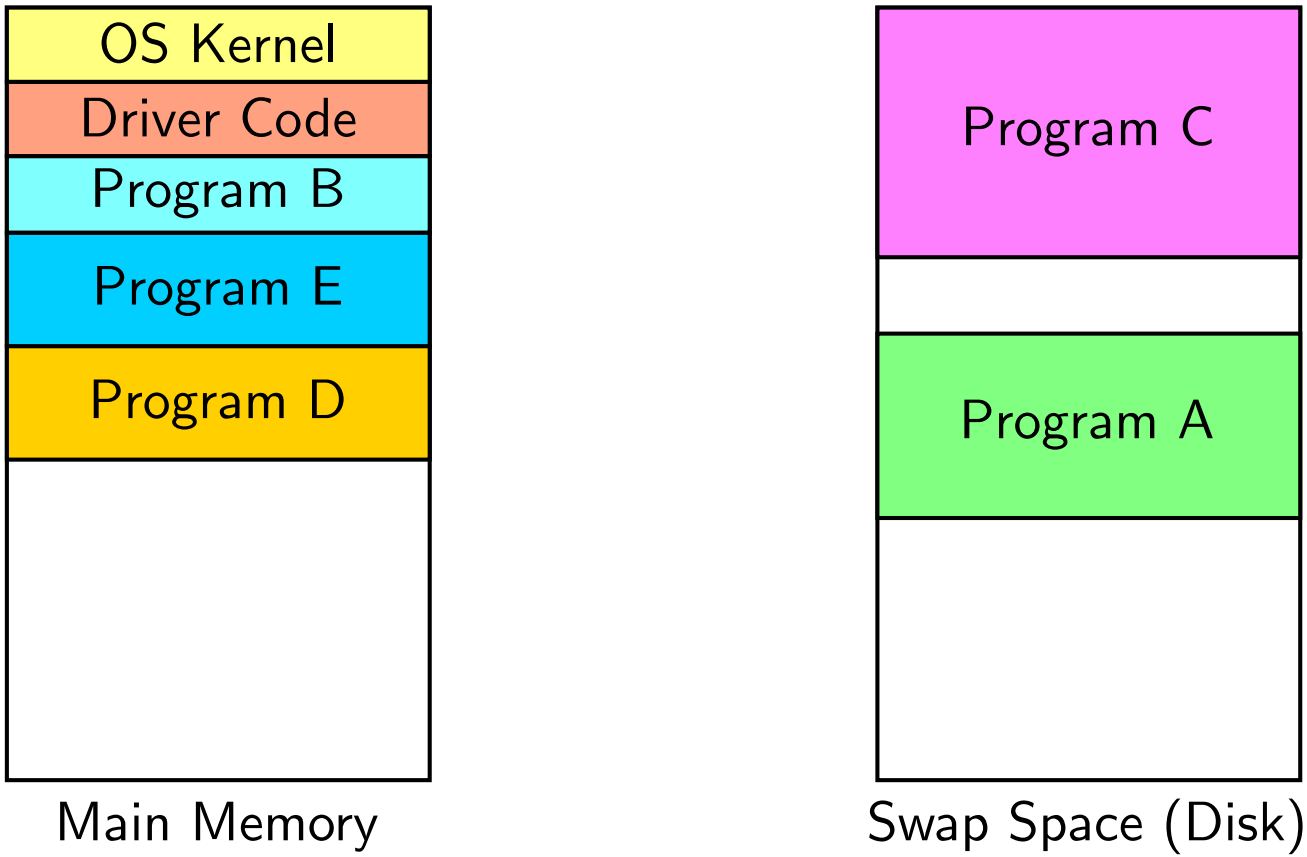
In this case, the operating system can **swap** one of the other running programs to a temporary area on disk (or other secondary storage) to make room for Program E. When the swapped process (Program C) is given control again, it can be returned to main memory.

Multitasking (12)



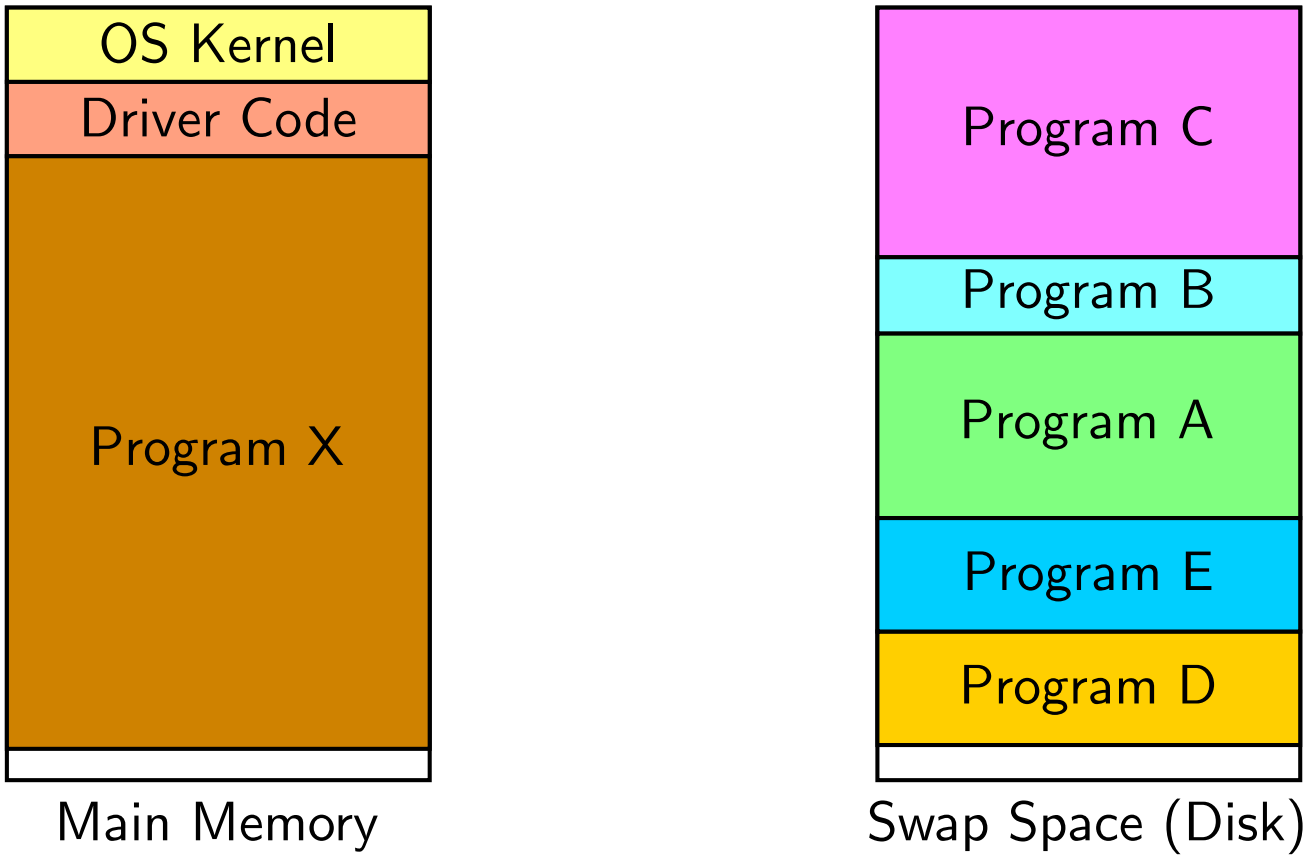
By swapping some programs to disk, enough main memory can be freed to run even massive programs, like Program X above.

Multitasking (13)



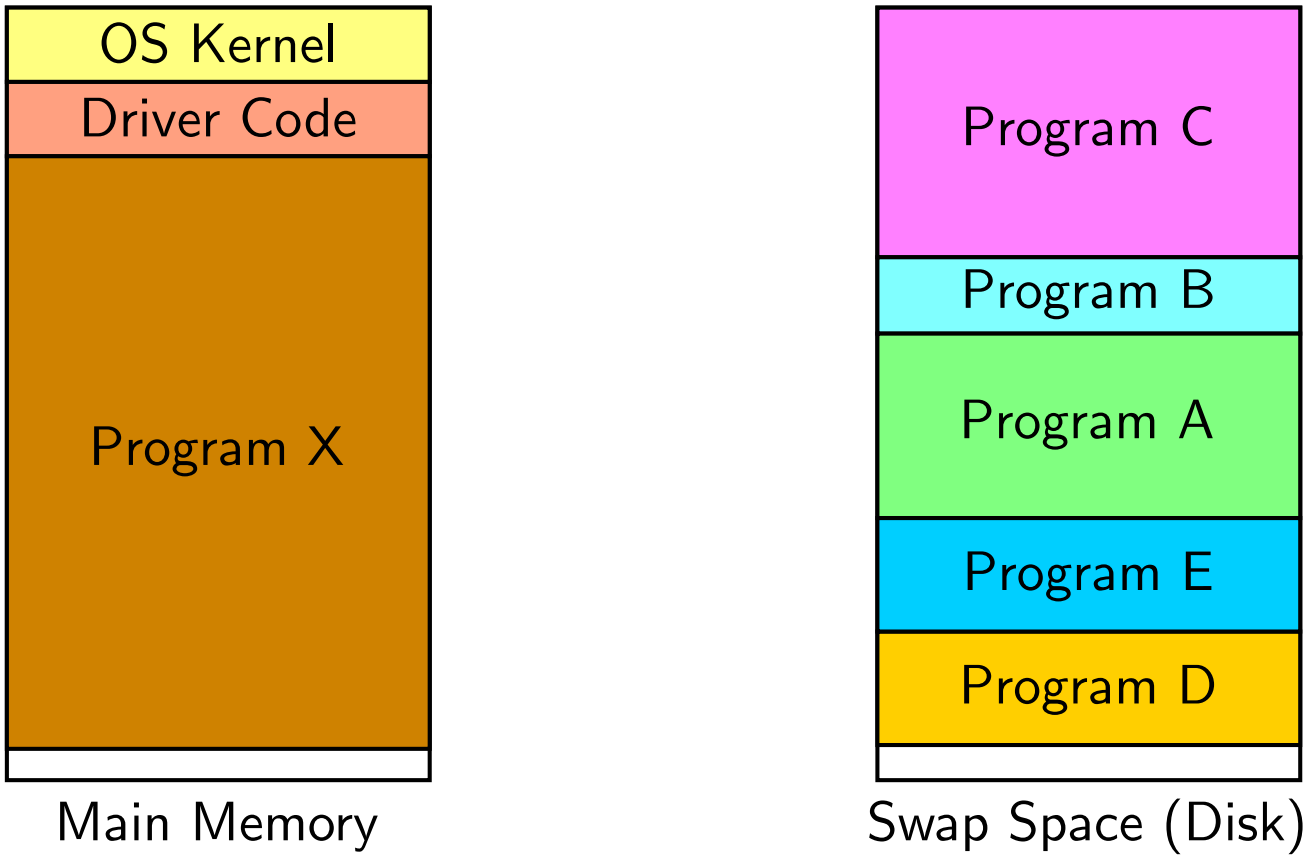
When Program X finishes, the other programs are selectively moved back to main memory as they are needed. Note that the OS may not be able to return each program's data to the same location it occupied previously.

Multitasking (14)



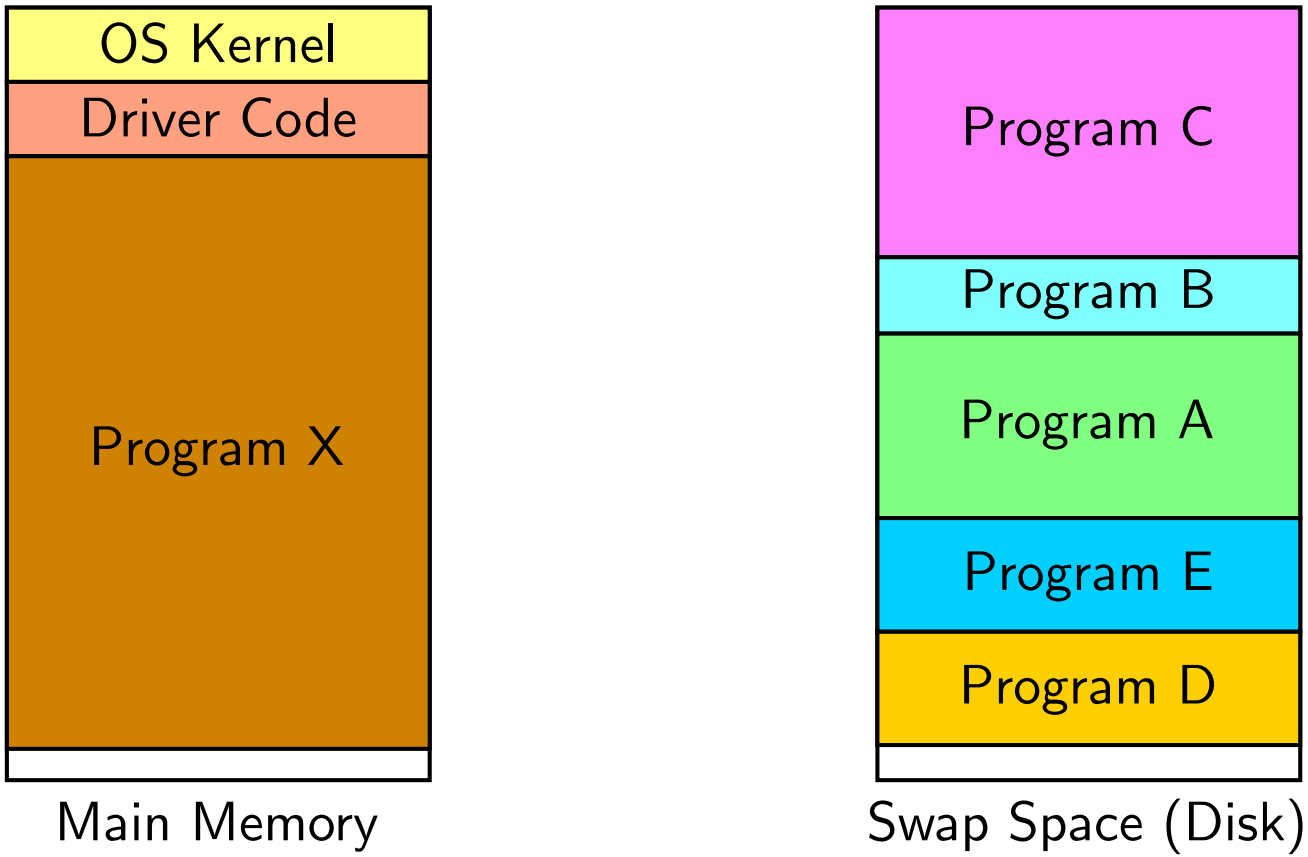
Question: What if the OS needs to give control to Program A again while Program X is running?

Multitasking (15)



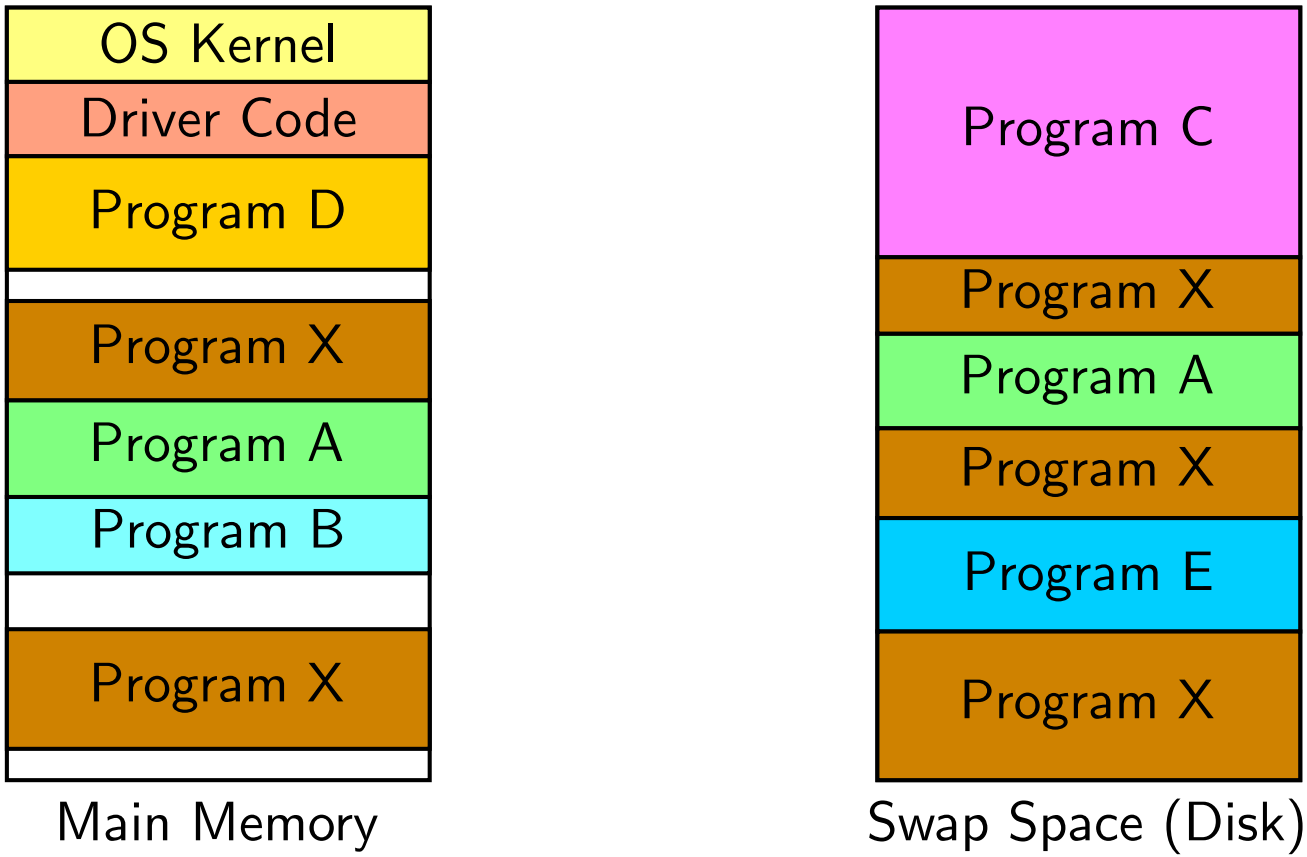
In the example above, where the swap space is the same size as main memory (which may not be the case in practice), there is no way to move Program X as a monolithic block.

Multitasking (16)



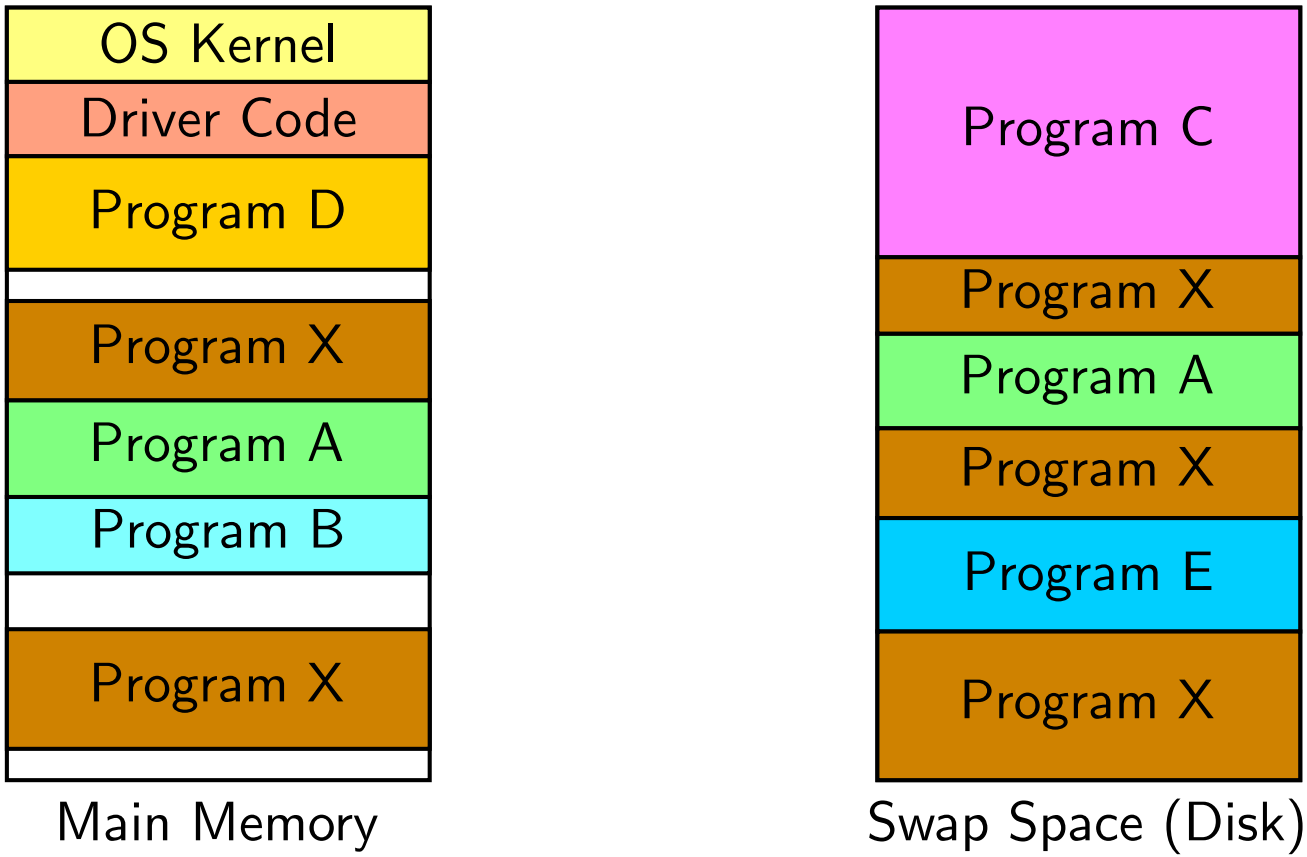
Idea: A particular program may not need all of its memory data at once. Why not swap unused parts of each program's data to disk?

Multitasking (17)



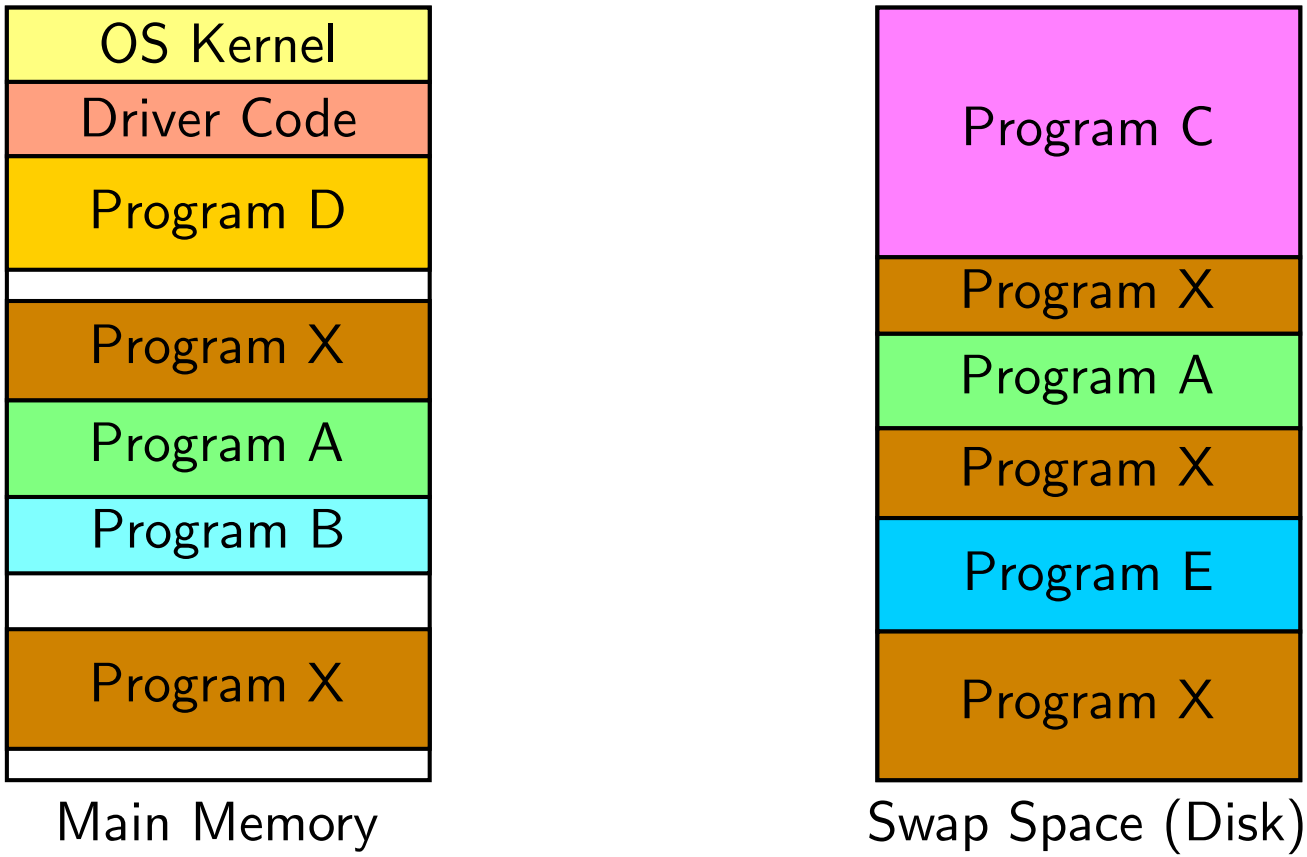
By selectively swapping parts of each program to disk, several programs can maintain a presence in main memory. Some programs may be split between main memory and the swap space.

Multitasking (18)



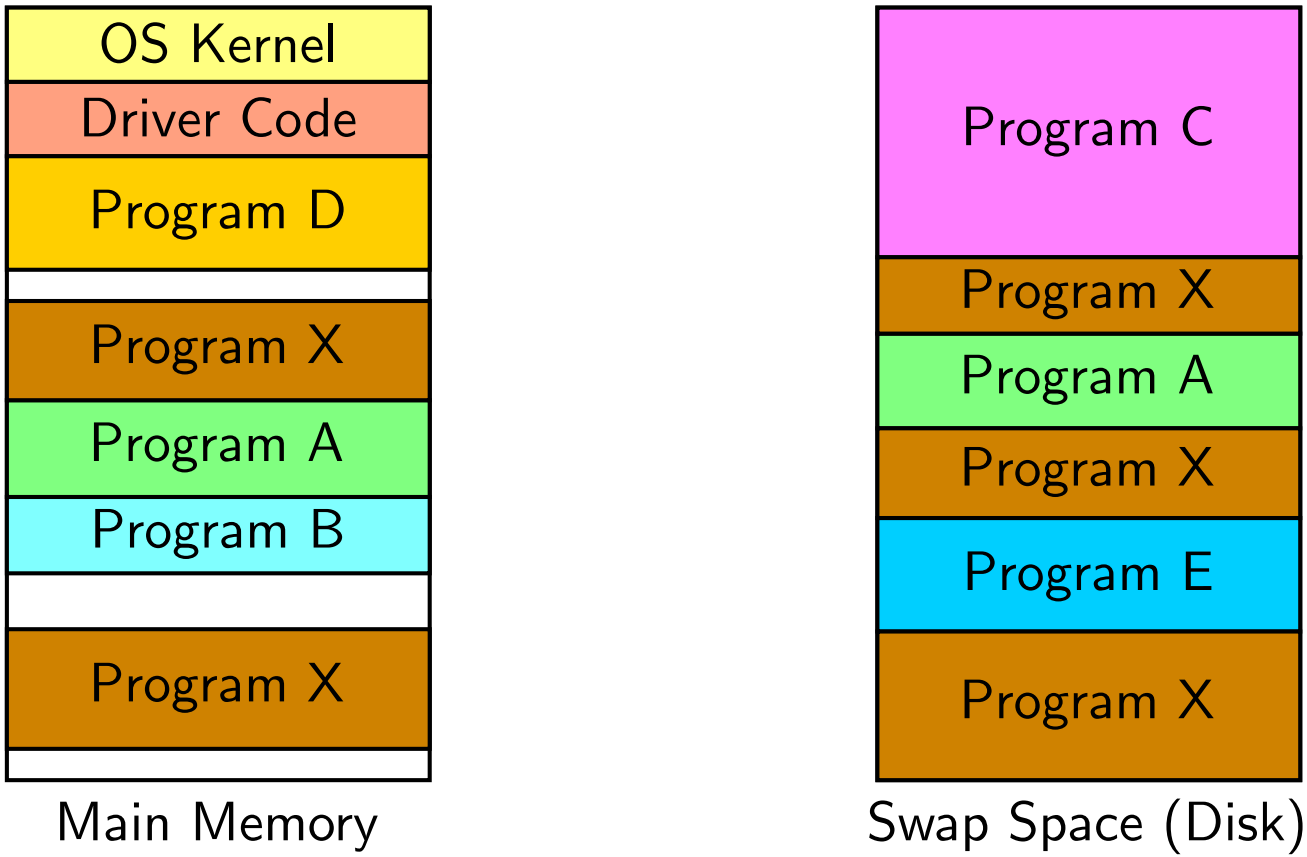
Operating systems need this flexibility since modern multitasked environments often have hundreds of running processes, of which some may only need to run (that is, have control over the processor) very rarely.

Multitasking (19)



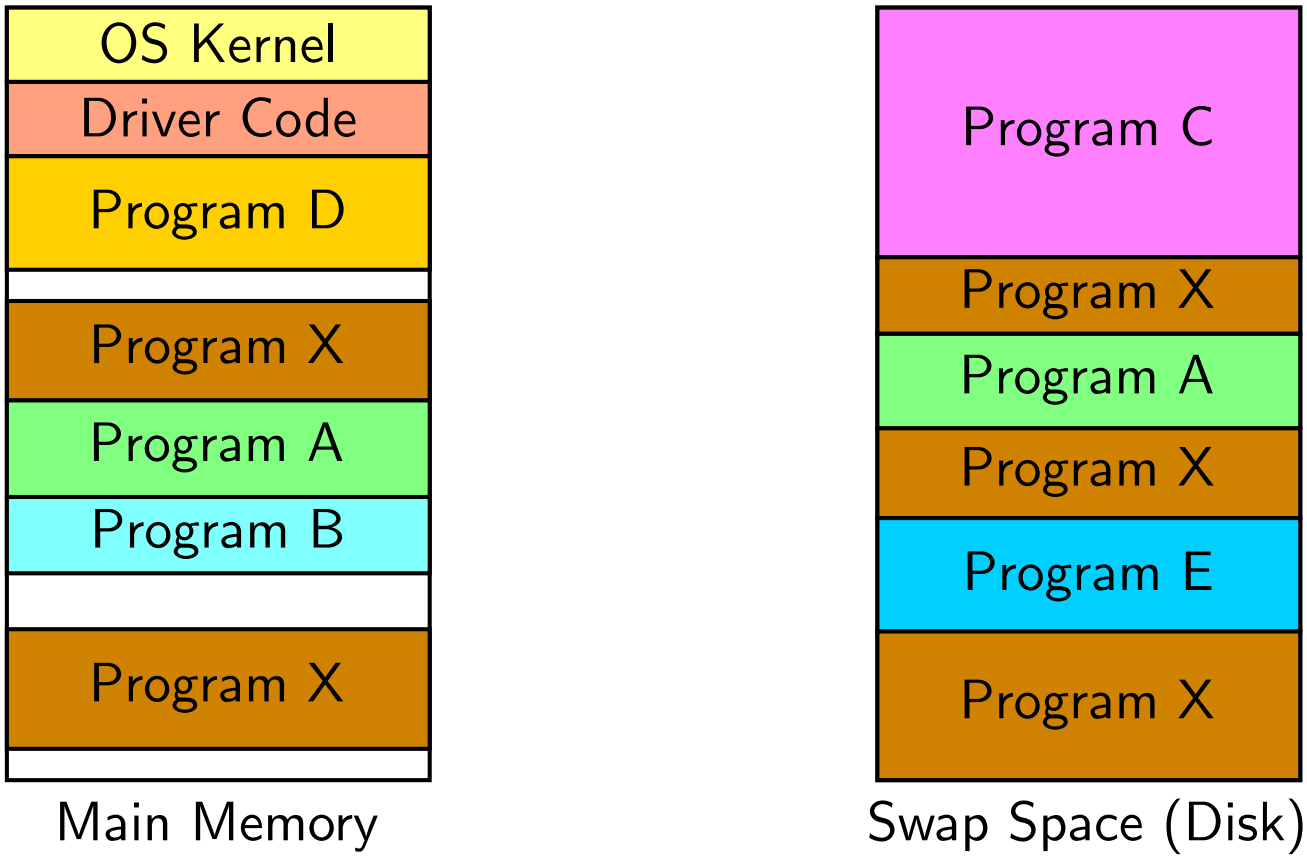
However, the code for each program is not generally aware of this scheme, for two reasons.

Multitasking (20)



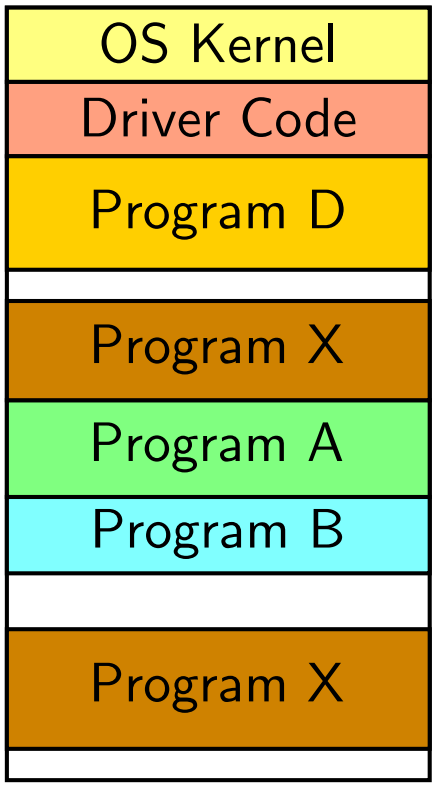
Reason One: The broader memory layout of the system is none of the program's business.

Multitasking (21)

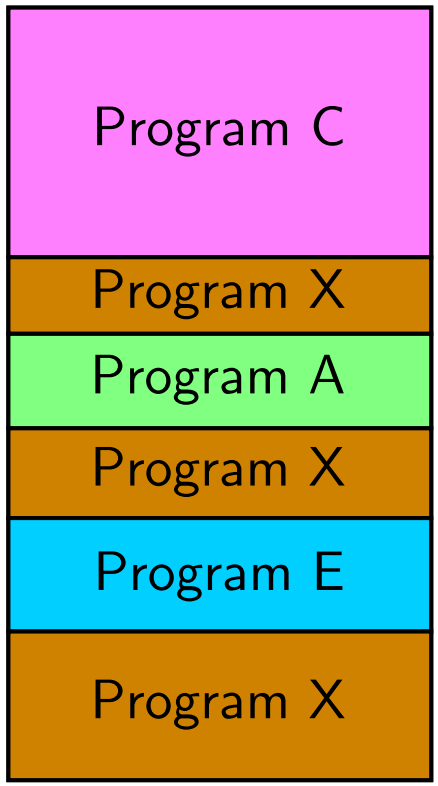


Reason Two: The program is written using normal instructions and addresses, which assume that all data is in (main) memory and that it is located at particular addresses which do not change.

Multitasking (22)



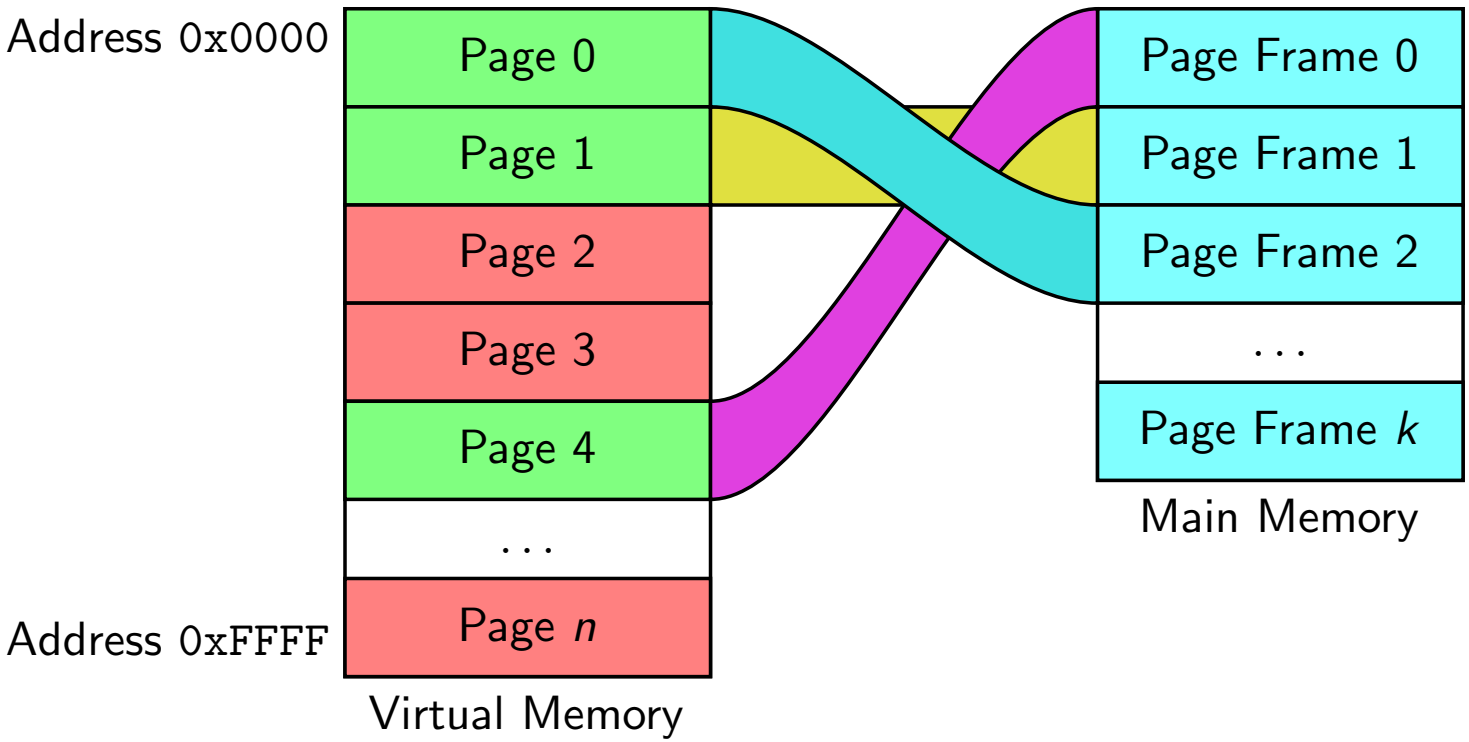
Main Memory



Swap Space (Disk)

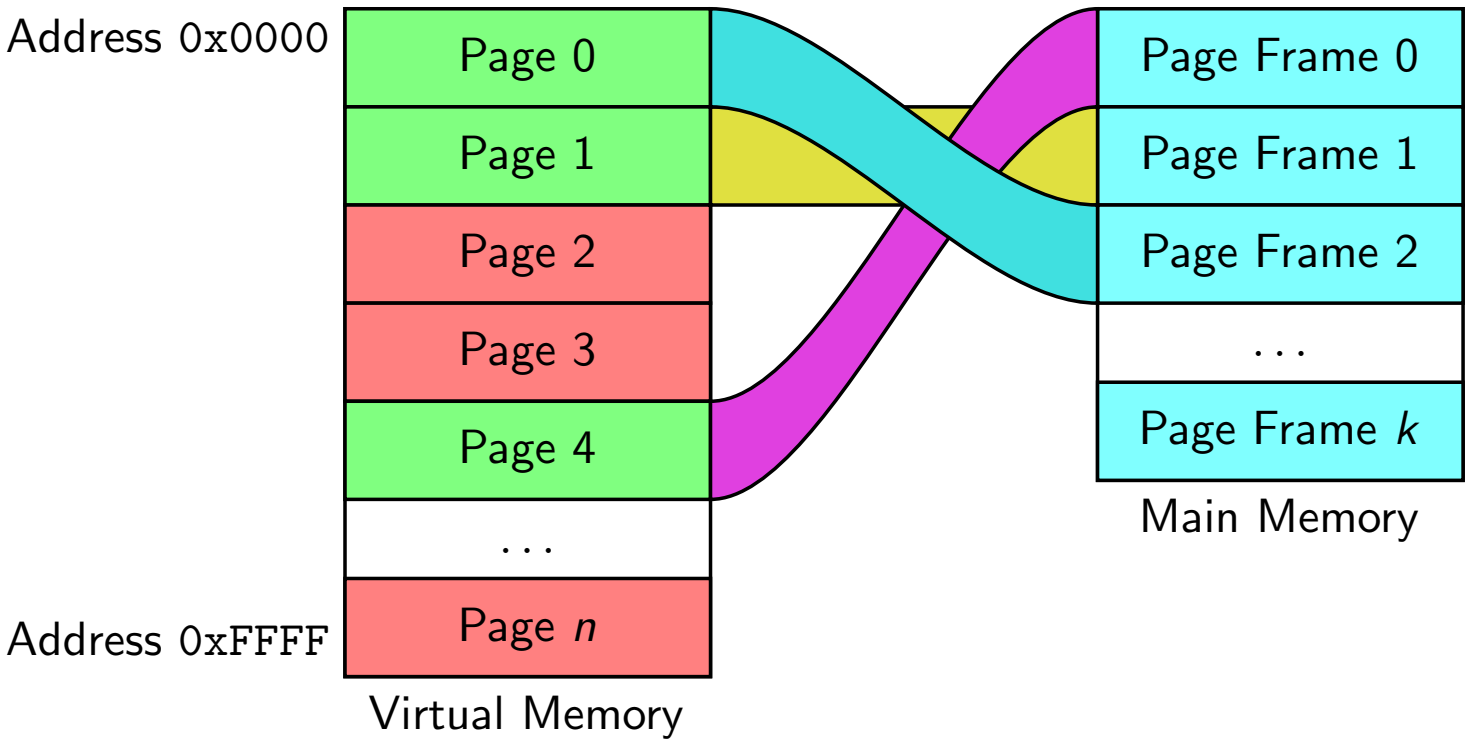
(For example, an `LDS` instruction uses a fixed address in data memory, and has no provision for the address to be easily changed or redirected to disk).

Virtual Memory (1)



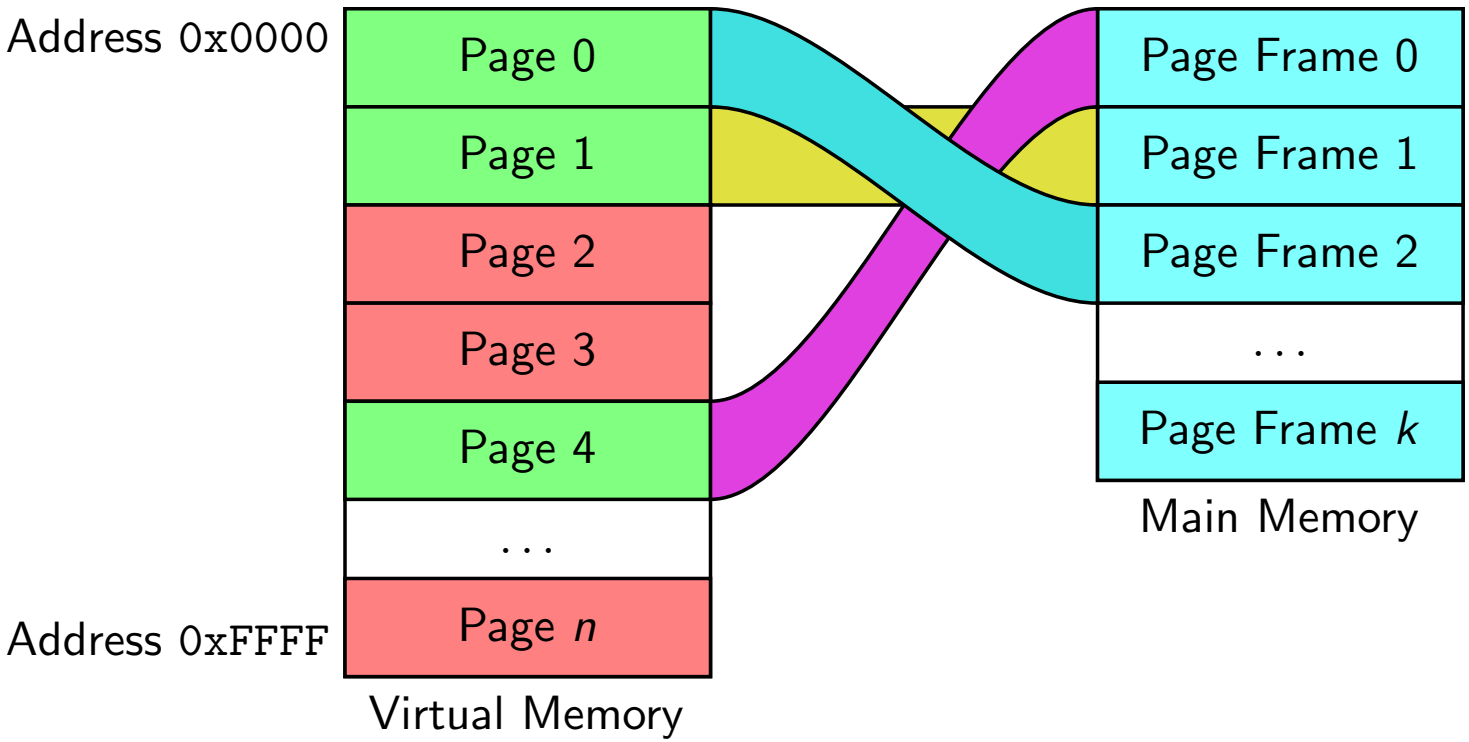
A **virtual memory** system adds a layer of abstraction on top of the main memory. Instead of memory addresses directly referring to main memory locations, memory addresses are 'virtual' and correspond to a much larger address space (possibly with more address space than the system could conceivably contain).

Virtual Memory (2)



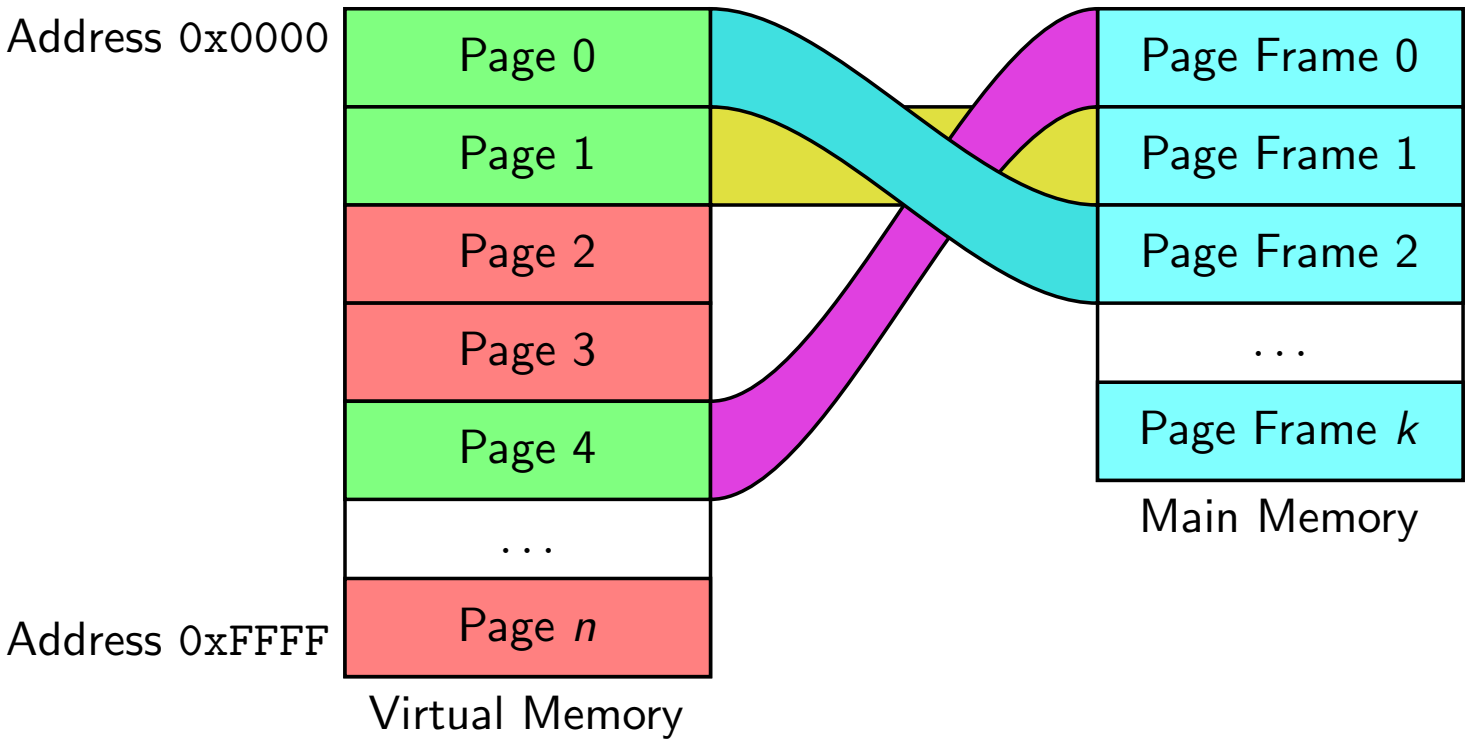
When a load or store request is made with respect to a particular virtual address, the hardware maps that address to a real address in main memory.

Virtual Memory (3)



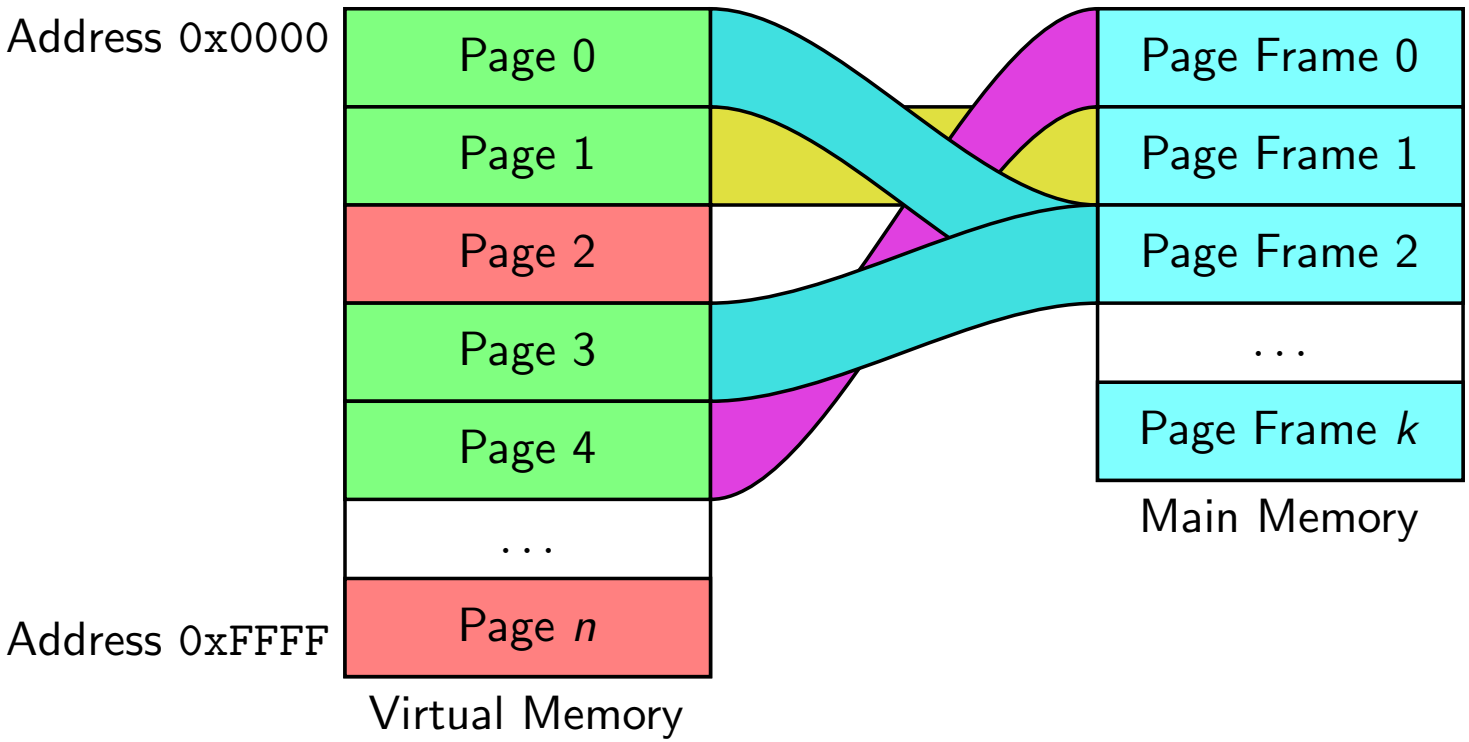
At any given time, only a small portion of the virtual memory space corresponds to real memory space. The mapping between virtual memory blocks and real memory blocks is also variable: a virtual block may be moved seamlessly in real memory.

Virtual Memory (4)



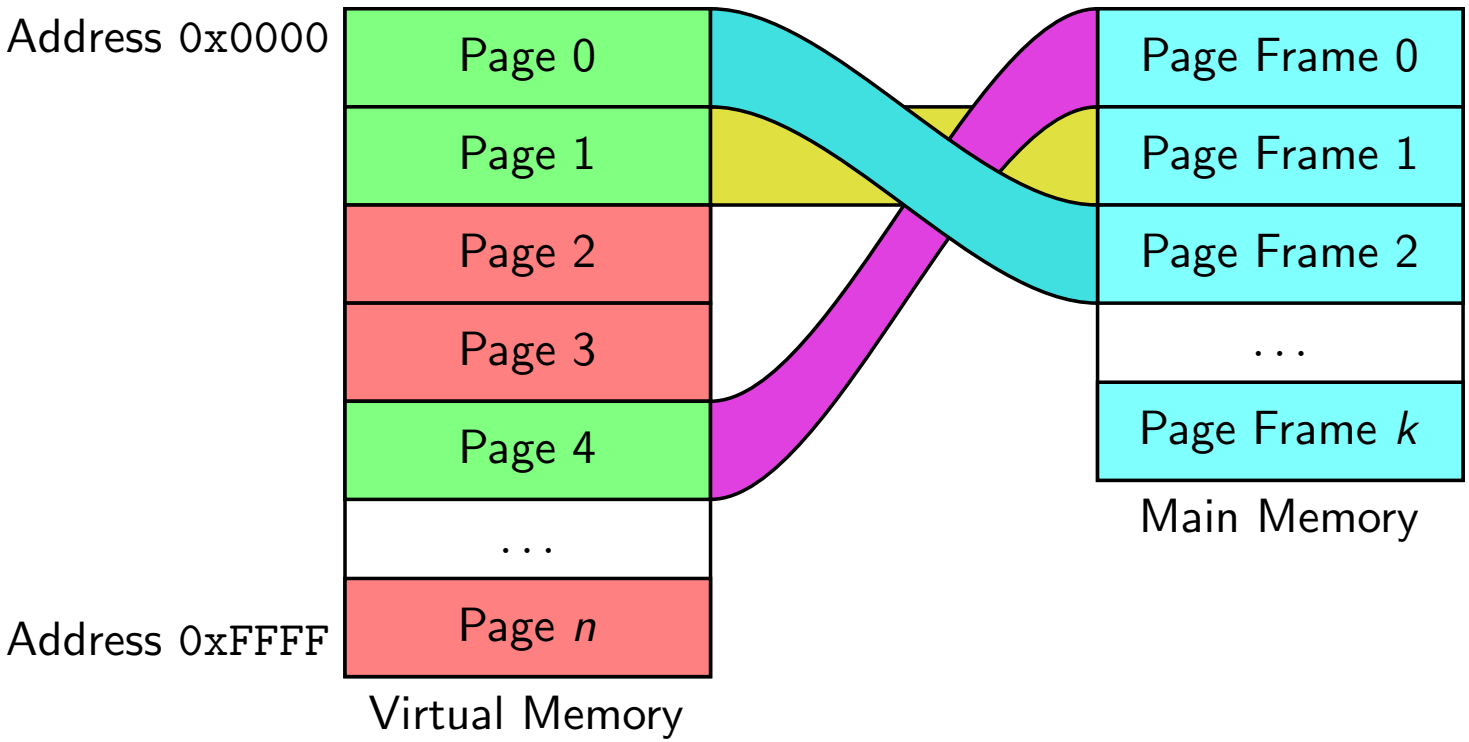
As a result, the operating system can choose which blocks to maintain in main memory and which blocks to store on disk (or simply ignore altogether).

Virtual Memory (5)



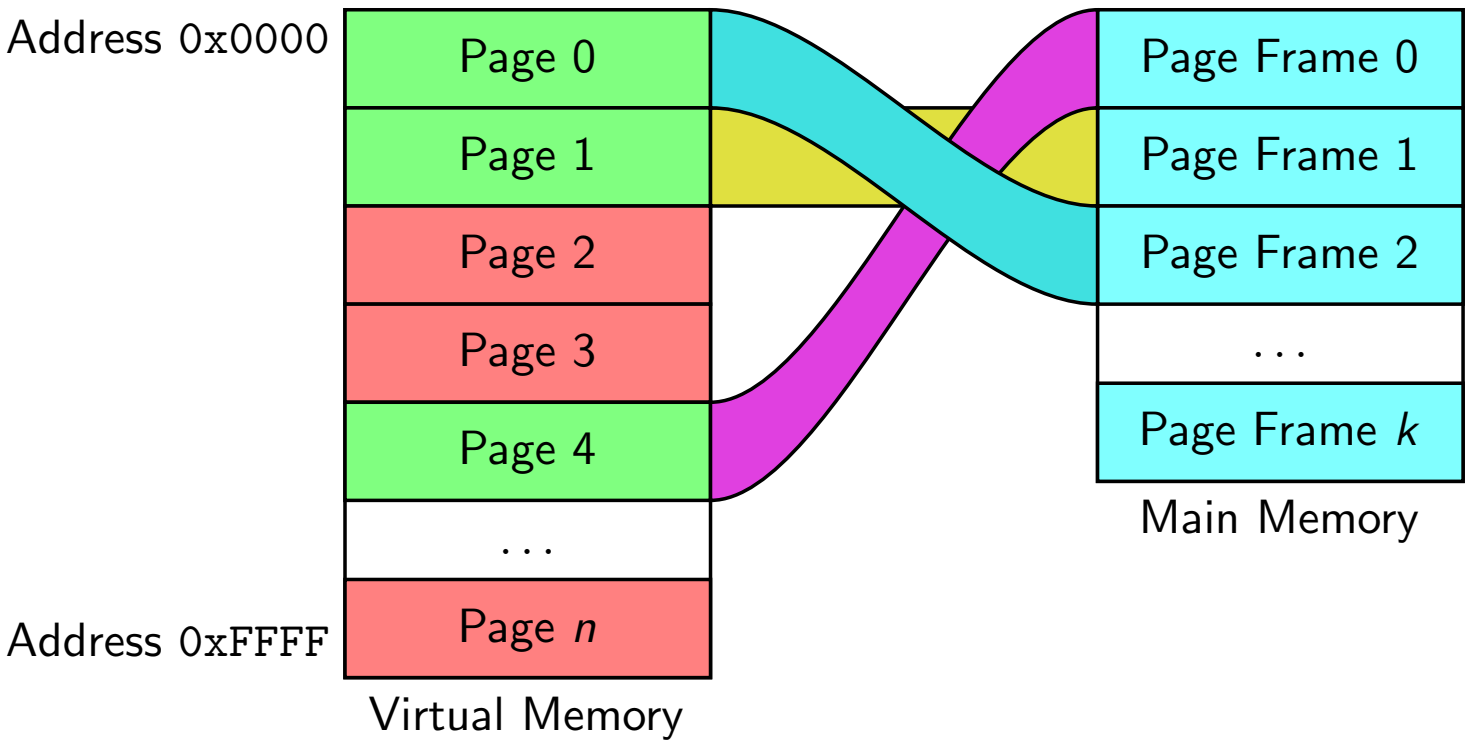
Additionally, if a common piece of data (for example, a software library) is needed by two processes, it can be 'loaded' into two different virtual memory locations while being stored in only one real memory location.

Virtual Memory (6)



The virtual address space is divided into blocks called **pages**. A subset of virtual pages will be mapped to real blocks of main memory. The remaining pages will either be ignored (for example, if they aren't used) or stored elsewhere (e.g. on disk).

Virtual Memory (7)



The operating system may reconfigure the mapping between virtual pages and real memory frames at any time. For example, if the program needs to access an unmapped page, the OS may map that page into main memory for use.

Virtual Memory Implementation (1)

In a virtual memory system, running programs (except the OS itself) are assigned a virtual address space. The programs use ordinary load and store instructions to access data, using virtual addresses. Each load and store instruction is intercepted by hardware **address translation** logic.

Some parts of the program's memory space will be stored in main memory, while other parts might be stored on disk (or elsewhere). If a requested virtual address is in main memory, the processor translates the address to a real memory address and continues. Otherwise, a **page fault** occurs, since the requested data is not in memory.

Virtual Memory Implementation (2)

When a page fault occurs, the operating system will determine how to remedy the problem. Normally, it will load the requested data into a frame in main memory so that the data can be used by the program.

It is important to understand that virtual memory systems **do not** allow disks to be used as primary memory. All actual processing (load and store instructions) will occur on main memory; virtual memory systems allow main memory to be reorganized automatically by the OS to expand the amount of useable memory.

Virtual Memory Implementation (3)

In some cases, a page fault may occur when a process attempts to access a page of memory which is not valid (for example, because of a bad pointer). In this case, the OS will not be able (or willing) to map any valid data into main memory, since no page is defined. Most operating systems will create an error condition in this case.

On Unix-based operating systems (including OS X) and Windows, attempting to access an invalid page will produce a segmentation fault (named for an older abstraction mechanism called segmentation, which has been displaced by modern virtual memory systems) and cause the immediate termination of the offending program.

Virtual Memory Implementation (4)

Since operating systems have control over the mapping between virtual memory and real memory, the mapping can also be used to impose **protection** on memory (on a coarse-grained per-page level) by marking some pages as read-only or write-only, or by marking pages as executable or non-executable. Attempts to misuse protected memory (such as executing code in a non-executable page) can be caught by the operating system and handled (again, usually with a segmentation fault).

The ability to mark pages as non-executable or read-only is useful for protecting against certain types of security exploits.
Non-executable pages can protect against attacks which attempt to execute arbitrary data as code. Read-only pages are useful for preventing self-modifying code (which is not generally needed for legitimate reasons on modern architectures).

Virtual Memory Implementation (5)

Question: On modern general purpose machines, when are there legitimate reasons to execute pages which are also modified like arbitrary data at run time?

Virtual Memory Implementation (6)

Virtual memory systems are normally implemented with a combination of hardware and software.

- When a running program uses a load or store instruction, hardware **address translation** is used to find the real address of the requested data.
- If the real address does not exist (e.g. if the requested virtual address is currently in a page stored on disk), the CPU invokes a handler routine (essentially an ISR) in the operating system, which handles the page fault.
- The operating system software is also responsible for setting up the virtual memory translation scheme (using special registers in the MMU).

Virtual Memory Implementation (7)

The mapping of virtual addresses to real addresses is performed with a **page table**. The page table stores the list of all virtual pages, along with their mappings to real memory pages (if applicable). Each process receives its own private page table.

The page table is usually stored somewhere in memory (in a protected area accessible only by the operating system), but it may also be stored directly in the MMU (although this requires extra storage circuitry). The operating system normally sets a control register in the MMU to the base address of the page table (and changes this address as it switches between processes).

Page Tables (1)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

Consider a virtual memory system with 18 bit virtual addresses, 17 bit real addresses and blocks of 2^{15} bytes (32KB). The size of main memory is then 2^{17} bytes (128KB).

Page Tables (2)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

Question: How many pages of virtual memory are there?

Page Tables (3)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

The number of pages can be obtained by dividing the total address space size (2^{18}) by the size of each block (2^{15}) to get $2^3 = 8$ pages.

Page Tables (4)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

The number of frames of real memory is similarly determined to be $2^2 = 4$.

Page Tables (5)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

The page table contains one entry for each virtual memory page. The page number of a particular virtual address is obtained by taking the high bits of the address (in the example above, the three highest bits of the 18 bit address are the page number).

Page Tables (6)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

Questions: Which page contains virtual address `0x17C5C`? What range of addresses is contained in page 6?

Page Tables (7)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

The highest three bits of `0x17C5C` (when it is interpreted as an 18-bit value) are 010, so the address lies in page 2. Page 6 spans the range `0x30000` through `0x37fff`.

Page Tables (8)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

Each entry of the page table contains a present bit (P), which indicates whether the page is currently stored in main memory, as well as a disk address and the index of the frame containing the page in main memory (if $P = 1$).

Page Tables (9)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

Question: How many bits are needed in the frame field?

Page Tables (10)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

The number of frames in main memory is $2^{17}/2^{15} = 2^2 = 4$, so two bits are necessary in this case. In general, virtual memory spaces are usually much larger than the actual size of main memory (and the size of main memory may not always be a power of two).

Page Tables (11)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

For example, on an architecture with 64 bit addresses and 32GB (2^{35} bytes) of main memory, virtual addresses might be 48 or 64 bits long.

Page Tables (12)

	P	Disk Address	Frame
Page 0			
Page 1			
Page 2			
Page 3			
Page 4			
Page 5			
Page 6			
Page 7			

The size and structure of the disk address is architecture (and OS) dependent and is determined by the storage scheme used by the OS.

Page Tables (13)

	P	Disk Address	Frame
Page 0	0	11001111011	-
Page 1	0	00111000001	-
Page 2	1	00000000111	01
Page 3	0	10110101110	-
Page 4	1	00111100011	00
Page 5	1	00011111100	10
Page 6	1	01101101111	11
Page 7	0	01101110111	-

In the example above, pages 2, 4, 5 and 6 are stored in main memory. The other pages are all swapped to disk.

Page Tables (14)

	P	Disk Address	Frame
Page 0	0	11001111011	-
Page 1	0	00111000001	-
Page 2	1	00000000111	01
Page 3	0	10110101110	-
Page 4	1	00111100011	00
Page 5	1	00011111100	10
Page 6	1	01101101111	11
Page 7	0	01101110111	-

Exercise: Give the physical (main memory) address accessed by a load from virtual address `0x36465`.

Page Tables (15)

	P	Disk Address	Frame
Page 0	0	11001111011	-
Page 1	0	00111000001	-
Page 2	1	00000000111	01
Page 3	0	10110101110	-
Page 4	1	00111100011	00
Page 5	1	00011111100	10
Page 6	1	01101101111	11
Page 7	0	01101110111	-

The value `0x36465` expands to $(11011001000110010 1)_2$ in 18 bit binary. The leading digits are 110, so the address lies in page 6.

Page Tables (16)

	P	Disk Address	Frame
Page 0	0	11001111011	-
Page 1	0	00111000001	-
Page 2	1	00000000111	01
Page 3	0	10110101110	-
Page 4	1	00111100011	00
Page 5	1	00011111100	10
Page 6	1	01101101111	11
Page 7	0	01101110111	-

Page 6 is mapped to physical frame 11. The frame index is combined with the lower 15 bits of the original address to form the physical address $(1111001000110010 1)_2 =$ `0x1e465`.

Page Tables (17)

	P	Disk Address	Frame
Page 0	0	11001111011	-
Page 1	0	00111000001	-
Page 2	1	00000000111	01
Page 3	0	10110101110	-
Page 4	1	00111100011	00
Page 5	1	00011111100	10
Page 6	1	01101101111	11
Page 7	0	01101110111	-

Conversely, if the program attempts to access address `0x0ebdf`, which lies in page 1, a page fault will occur.

Page Tables (18)

	P	Disk Address	Frame
Page 0	0	11001111011	-
Page 1	0	00111000001	-
Page 2	1	00000000111	01
Page 3	0	10110101110	-
Page 4	1	00111100011	00
Page 5	1	00011111100	10
Page 6	1	01101101111	11
Page 7	0	01101110111	-

When a page fault occurs, the operating system handles it in software. If a new page is loaded, one of the existing frames will be swapped back to disk. The exact algorithm for choosing the page to eject may vary (and since it's implemented by software, any algorithm can be used).

Page Tables (19)

	P	Disk Address	Frame
Page 0	0	11001111011	-
Page 1	0	00111000001	-
Page 2	1	00000000111	01
Page 3	0	10110101110	-
Page 4	1	00111100011	00
Page 5	1	00011111100	10
Page 6	1	01101101111	11
Page 7	0	01101110111	-

Generally, virtual memory is fully associative (a page can be loaded into any frame) since the cost of loading a page from disk is extremely high, so the replacement algorithm's speed is not as much of a concern.

Cache and Virtual Memory (1)

Modern high performance procesors contain both hardware caches and virtual memory systems. The cache is considered part of the physical memory system (even if it is located physically inside the processor), so it works **only** with physical main memory addresses.

As a result, virtual memory translation must occur **before** the memory access is sent through the hardware cache. This can lead to performance issues, since the translation may require a main memory access (if the page table is in memory).

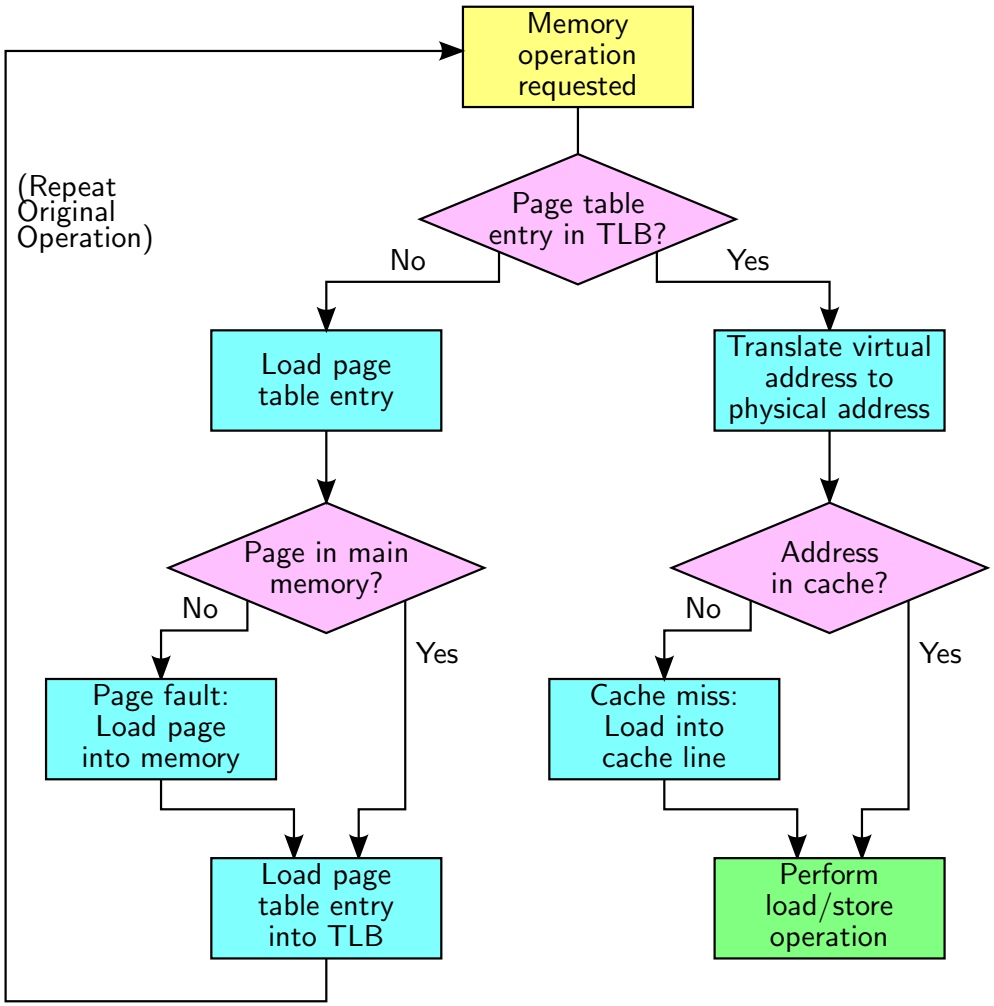
All accesses to main memory go through the cache, so if the CPU must load from the page table, it is possible that the page table is cached. However, in general the translation process can at least double the access time of memory requests.

Cache and Virtual Memory (2)

One solution is a **translation lookaside buffer** (TLB), which is an extra cache structure (which is physically located inside the CPU, like the regular cache) that caches the recently used page table entries. Address translation operations first check the TLB before looking up the address in the page table, and if the TLB contains the desired page table entry, no memory (or regular cache) access is needed to translate the address.

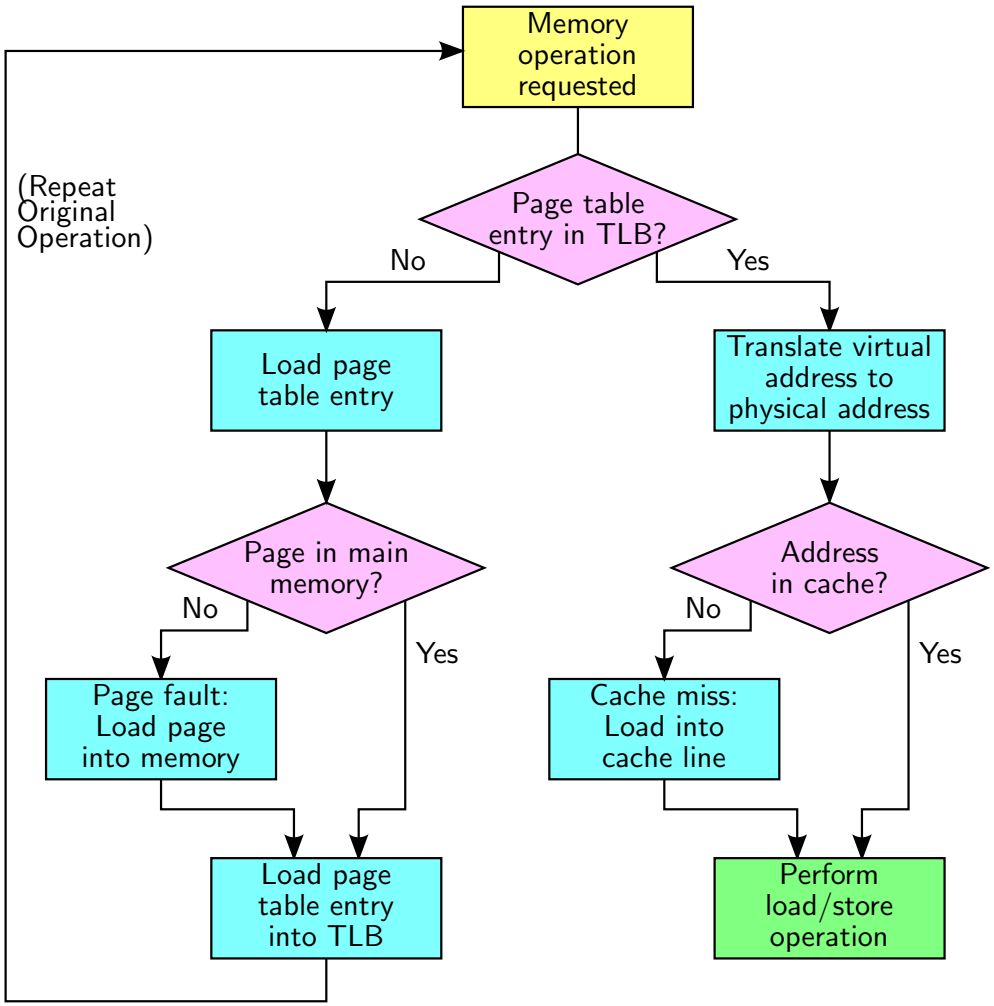
The TLB maintains a small subset of the full page table, but by the usual assumptions about memory locality, the TLB can be very effective at eliminating accesses to the page table in memory. When a TLB is present, memory accesses are normally only performed on pages that are present in the TLB. If the page table entry for a particular access is not found in the TLB (which is analogous to a cache miss), the TLB is updated by loading the relevant page table entry and the entire operation is repeated (and will then produce a hit in the TLB cache).

Cache and Virtual Memory (3)



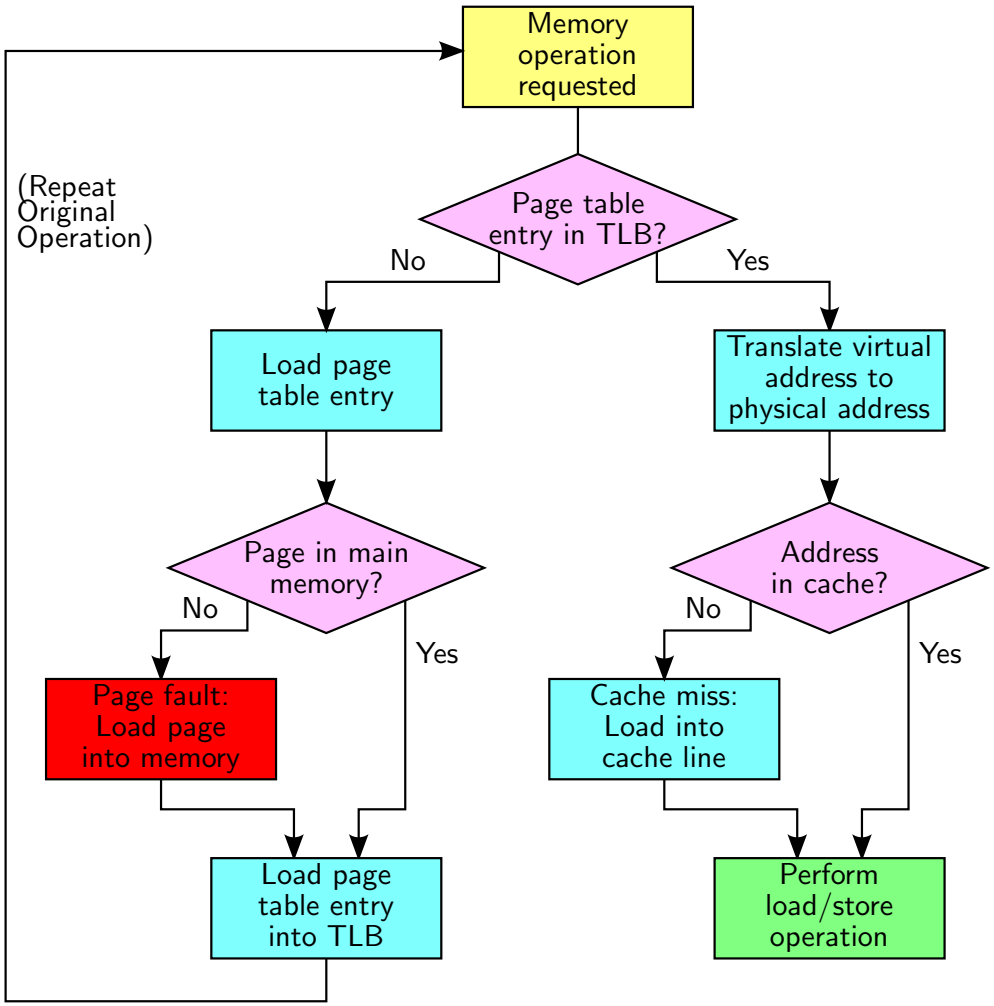
The flow chart above illustrates the full memory access cycle used in modern general purpose architectures like the x86.

Cache and Virtual Memory (4)



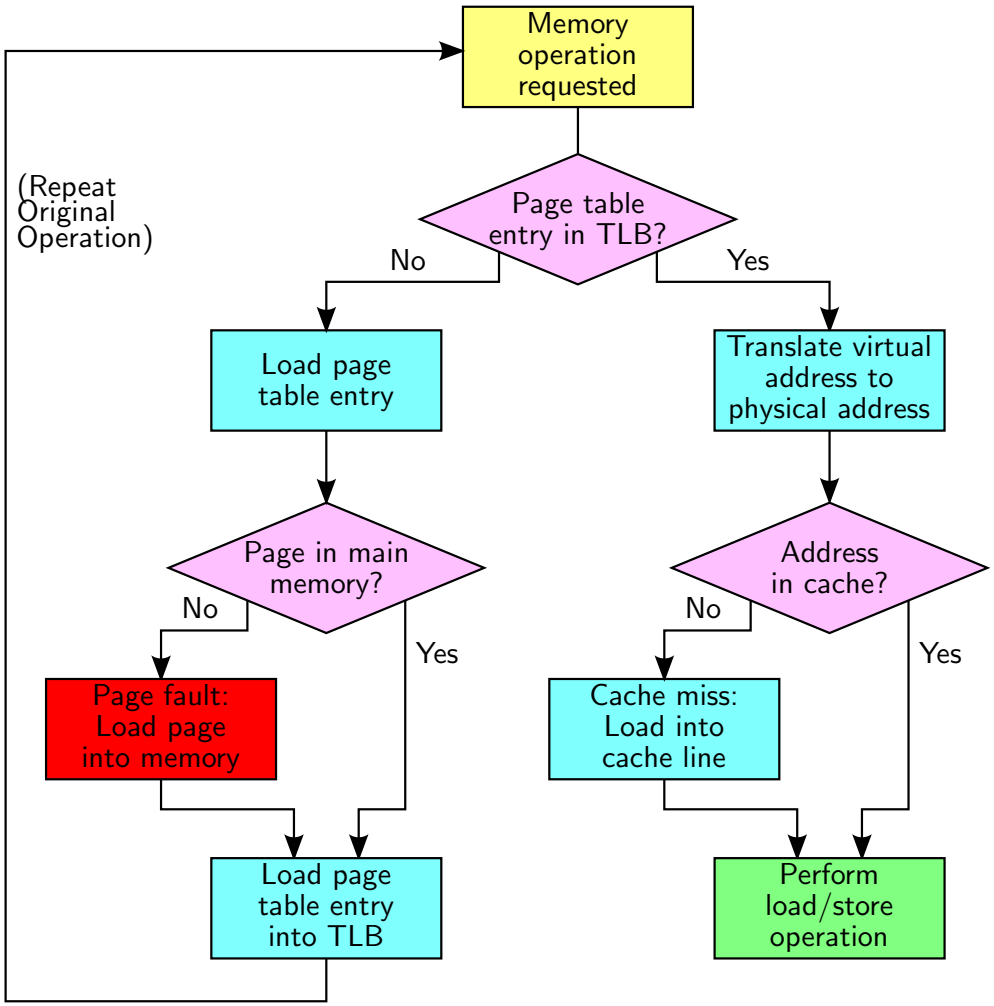
Natural Question: Why must a TLB miss cause the entire operation to be repeated? Why not just load the page table entry into the TLB and continue as if a TLB hit occurred?

Cache and Virtual Memory (5)



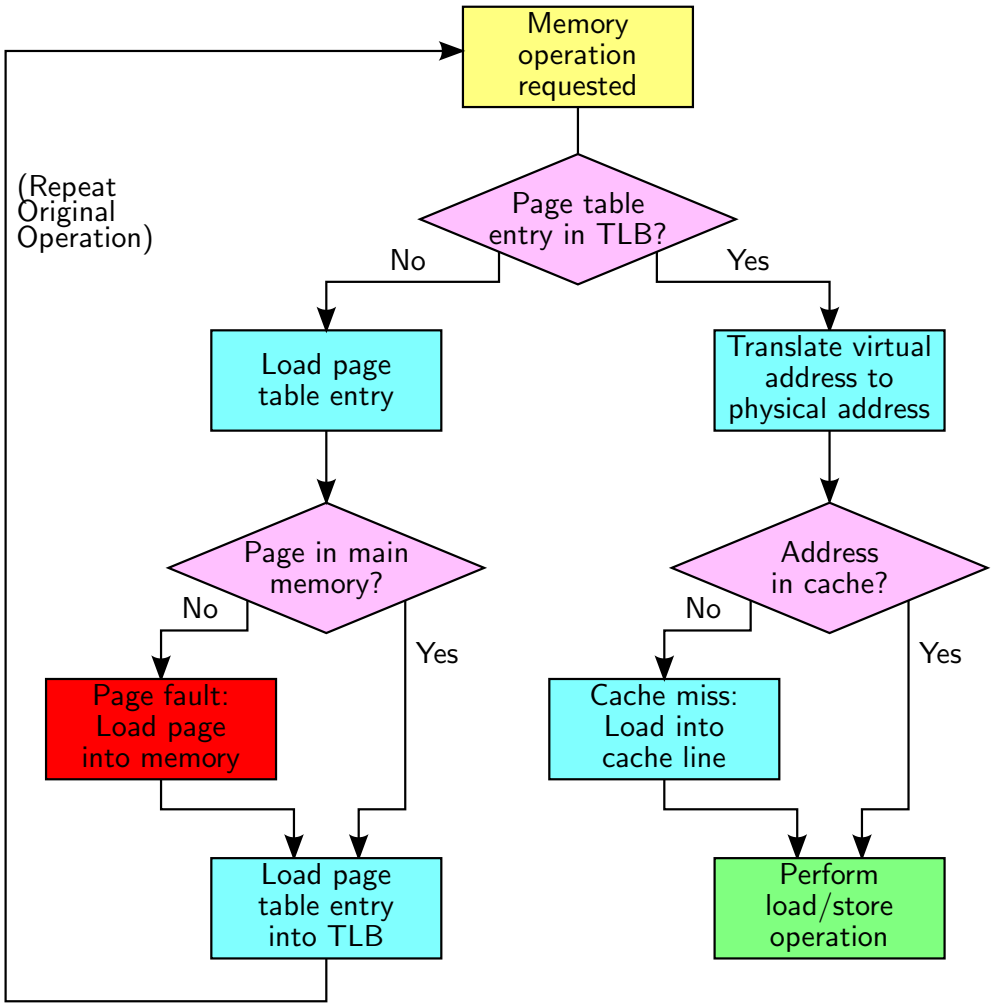
A TLB miss occurs during the execution of a single instruction, but the handler for page faults is a software interrupt handler (which might be millions of instructions long).

Cache and Virtual Memory (6)



The odd TLB miss behavior is due to the complexity of the highlighted component above.

Cache and Virtual Memory (7)



When the OS page fault handler finishes, the instruction that produced the TLB miss cannot pick up where it left off, so the entire instruction must be restarted.