5. Game Engine Support System (part. 2)

Game Player Experience Design

Prof. H. Kang

Keywords of prior knowledge

Hope these keywords help your study:

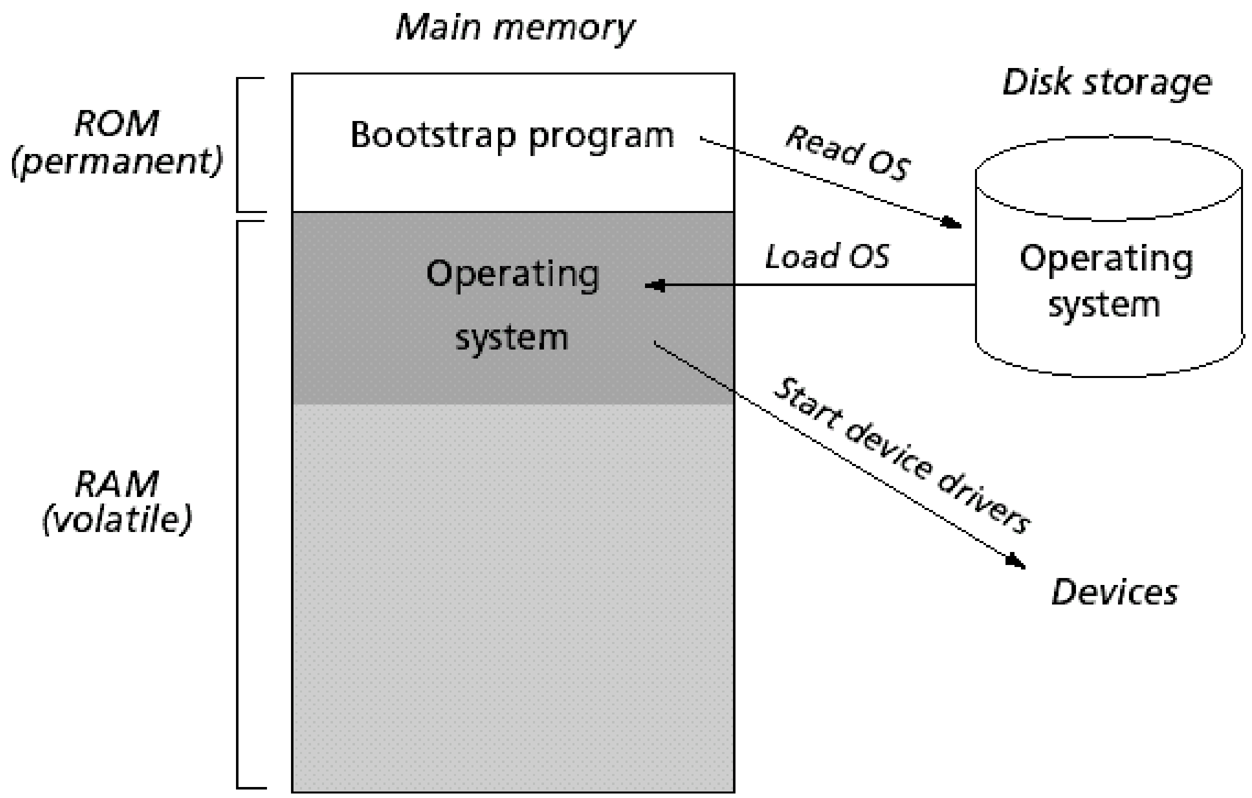
- Kernel (Operating System)
- Virtual Memory
- Page (Operating System)
- Physical Memory & Logical Memory

Kernel

What is a kernel?

- The kernel is the core computer program of the operating system that has a complete control over the system.

- On computer power-on, the kernel is the second program to run on the system preceded by the bootstrap program.
  - The **bootstrap** program is the first program to be loaded and executed when the computer system is started.
  - Subsequently, it locates the **kernel** from the storage and mounts it into the memory.
  - The kernel handles the rest of the startup.

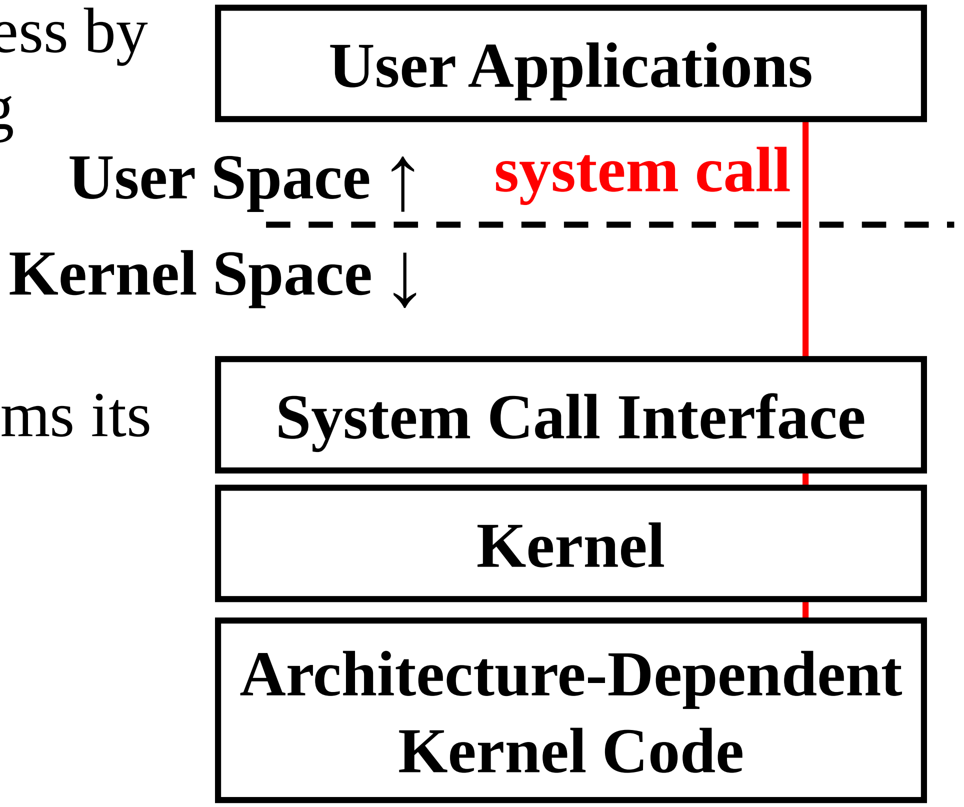


Kernel

What is a kernel?

- The **critical code of the kernel** is usually loaded into a separate area of memory, which is protected from access by applications or other less critical parts of the operating system.
  - We call this space the kernel space.

- In kernel space, the **critical code of the kernel** performs its tasks, such as running processes, managing hardware devices, handling interrupts, etc.

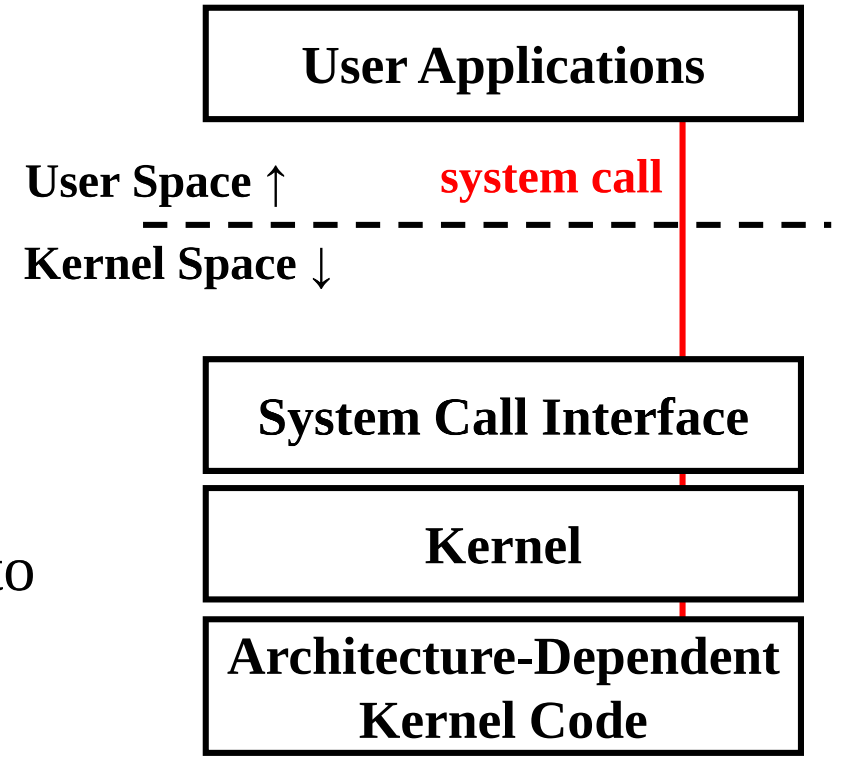


Kernel

What is a kernel?

- In user space, application programs like a game, chrome browser, or PowerPoint performs their own tasks.

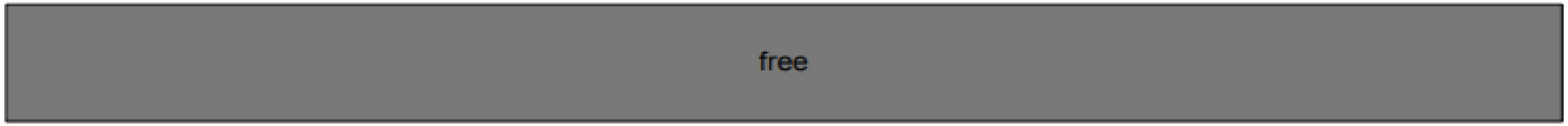
- When a process requests a service from the kernel, it must invoke a system call, usually through a wrapper function.
  - The **system call** is the programmatic way in which a computer program requests a service from the kernel.
  - The **wrapper function** is a subroutine in a software library or a computer program whose main purpose is to call a second subroutine.



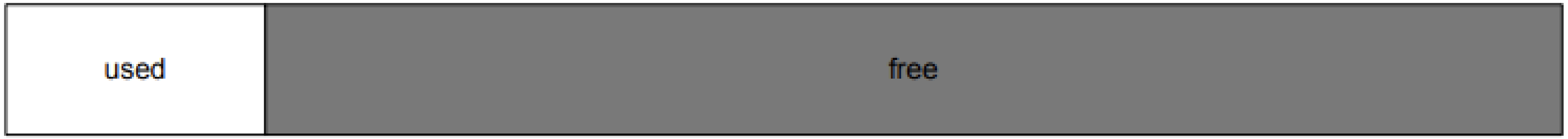
Memory Fragmentation

Fragmented memory

- One major problem caused by dynamic heap allocation is that memory can become *fragmented* over time.
  - When a program first runs, its heap memory is entirely free.



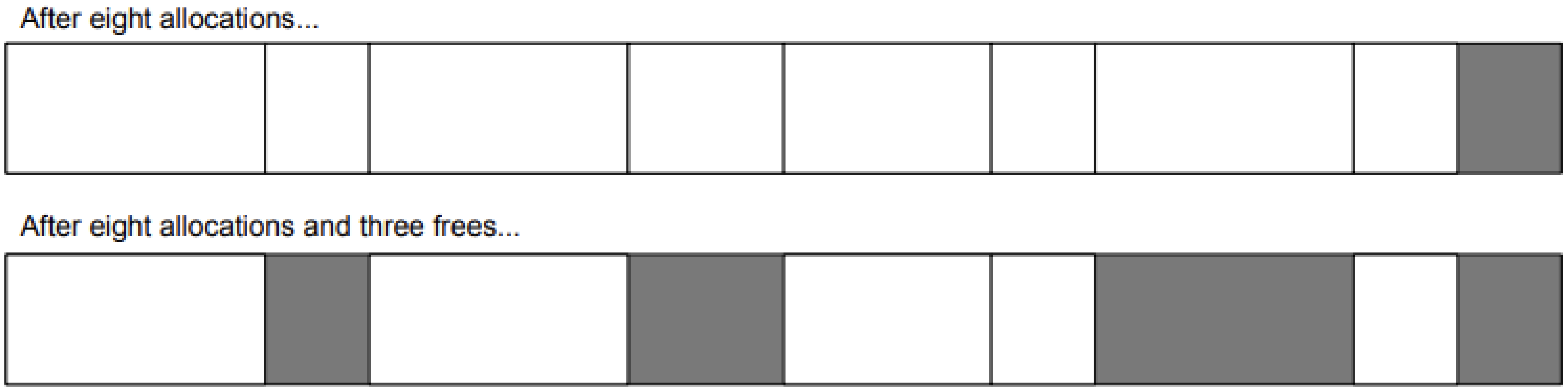
  - When a block is allocated, a contiguous region of heap memory of the appropriate size is marked as “in use,” and the remainder of the heap remains free.



Memory Fragmentation

Fragmented memory

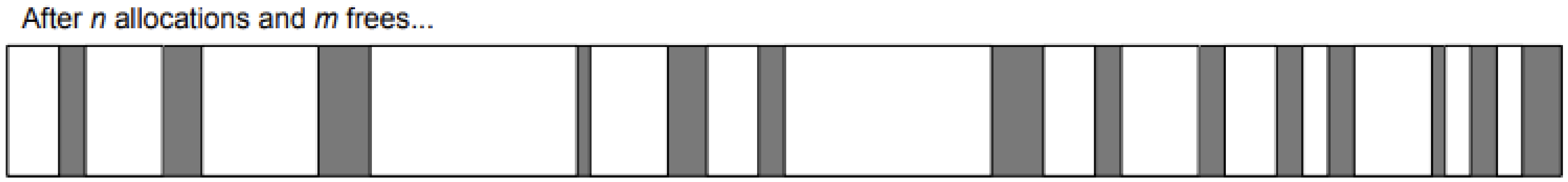
- One major problem .. (cont.)
  - When a block is freed, it is marked as such, and adjacent free blocks are merged into a single, larger free block.


After eight allocations...
After eight allocations and three frees...

Memory Fragmentation

Fragmented memory

- One major problem .. (cont.)
  - Over time, as allocations and deallocation of various sizes occur in random order, the heap memory begins to look like a patchwork of free and used blocks.
  - We can think of the free regions as “holes” in the fabric of used memory.
  - When the number of holes becomes large, and/or the holes are all relatively small, we say the memory has become fragmented.



Memory Fragmentation

Memory Fragmentation in PC games

- In PC games, memory fragmentation problems are handled nearly automatically by the OS with the concept of virtual memory.
- Virtual memory is the technique of utilizing a map to link a program’s memory addresses to actual physical memory addresses in RAM.
- Understanding this requires some prior knowledge.

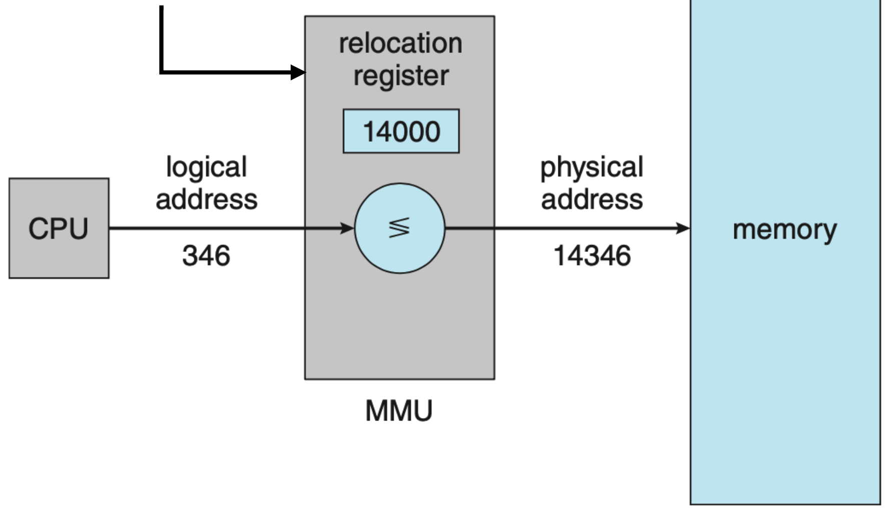
Physical & logical addresses

- Let us look at the **physical & logical addresses**.
  - Address uniquely identifies a location in the memory.

- **Logical address** (also known as virtual address)
  - This is used like a reference, to access the physical address.
  - This is generated by the CPU for a program.
  - This can be viewed by the user.

- **Physical address**
  - This refers to a location in the memory unit.
  - The user cannot view a physical address of program.
  - The user cannot directly access physical address.
  - Physical address is computed by memory management unit (MMU).

The **relocation register** holds a constant value that is determined by the location of the memory area assigned to the program.



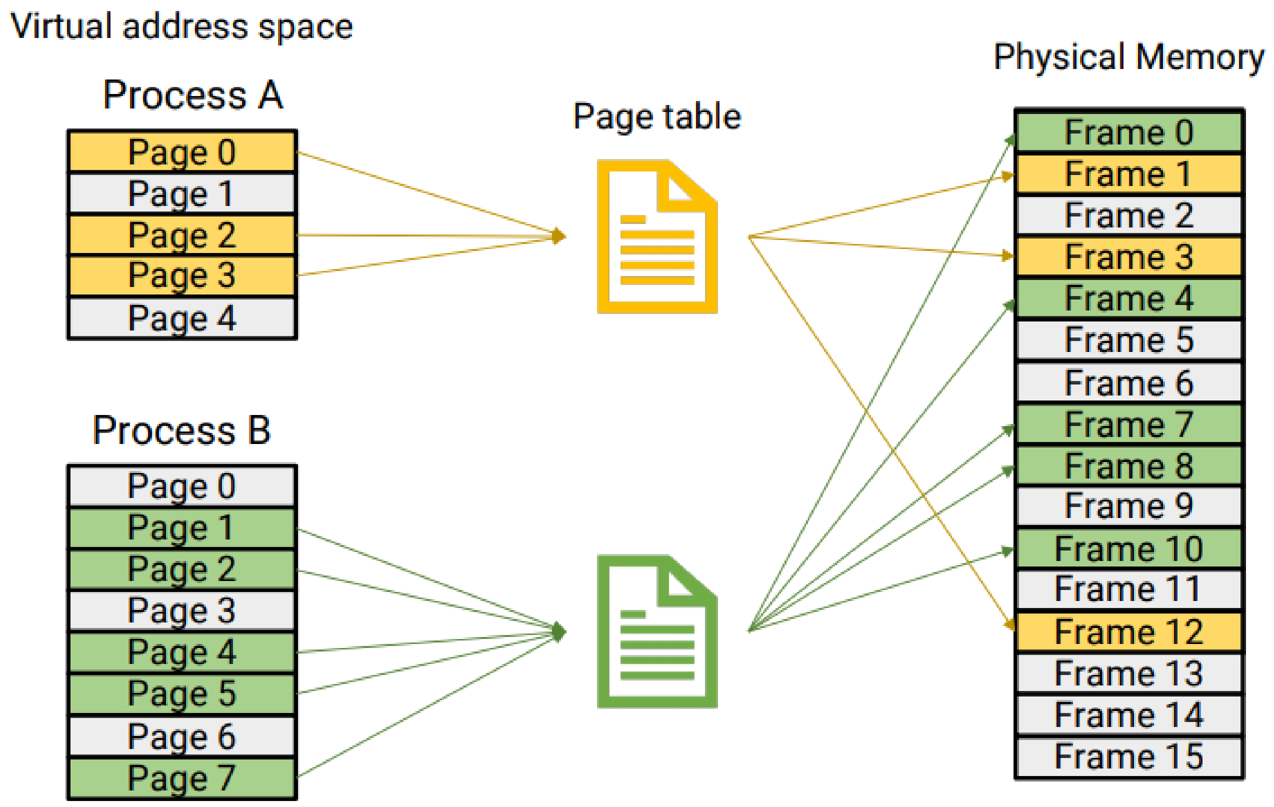
Memory Fragmentation

Page

- A page (also known as memory page or virtual page) is a **fixed-length** contiguous block of virtual memory.
- A page table is the data structure used by a virtual memory system to store the mapping between virtual addresses and physical addresses.
  - Physically, the memory of each process (or program) may be dispersed across different areas of physical memory, or may have been moved to another storage, typically to a hard disk drive.
  - When a process requests access to data in its memory, it is the responsibility of the operating system to map the virtual address provided by the process to the physical address of the actual memory where that data is stored.
  - The page table is where the operating system stores its mappings of virtual addresses to physical addresses, with each mapping also known as a page table entry.

Fame

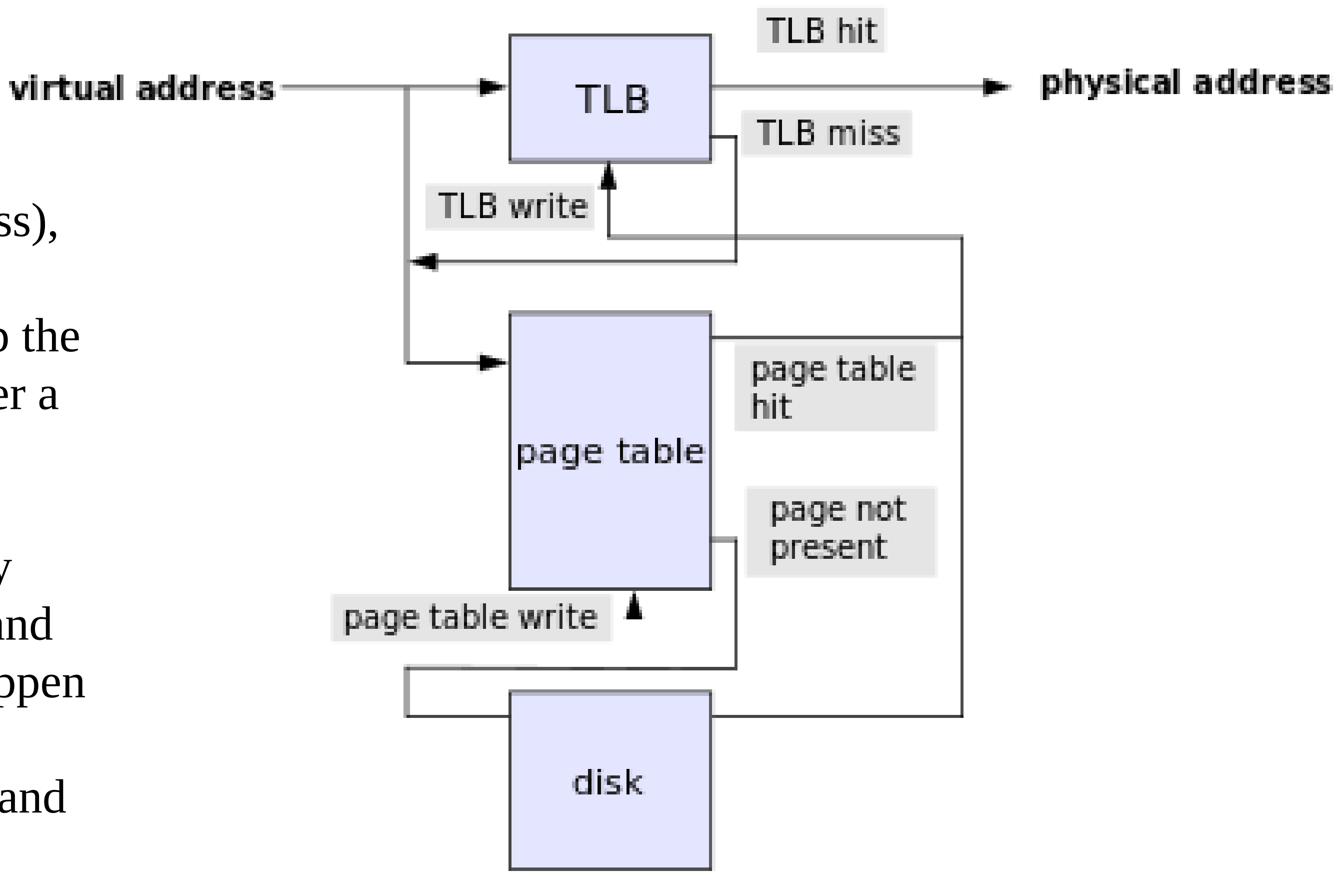
- A frame (or page frame) is a **fixed-length** block of physical memory (i.e. RAM)
- The frame may not be contiguous



Memory Fragmentation

Translational lookaside buffer (TLB)

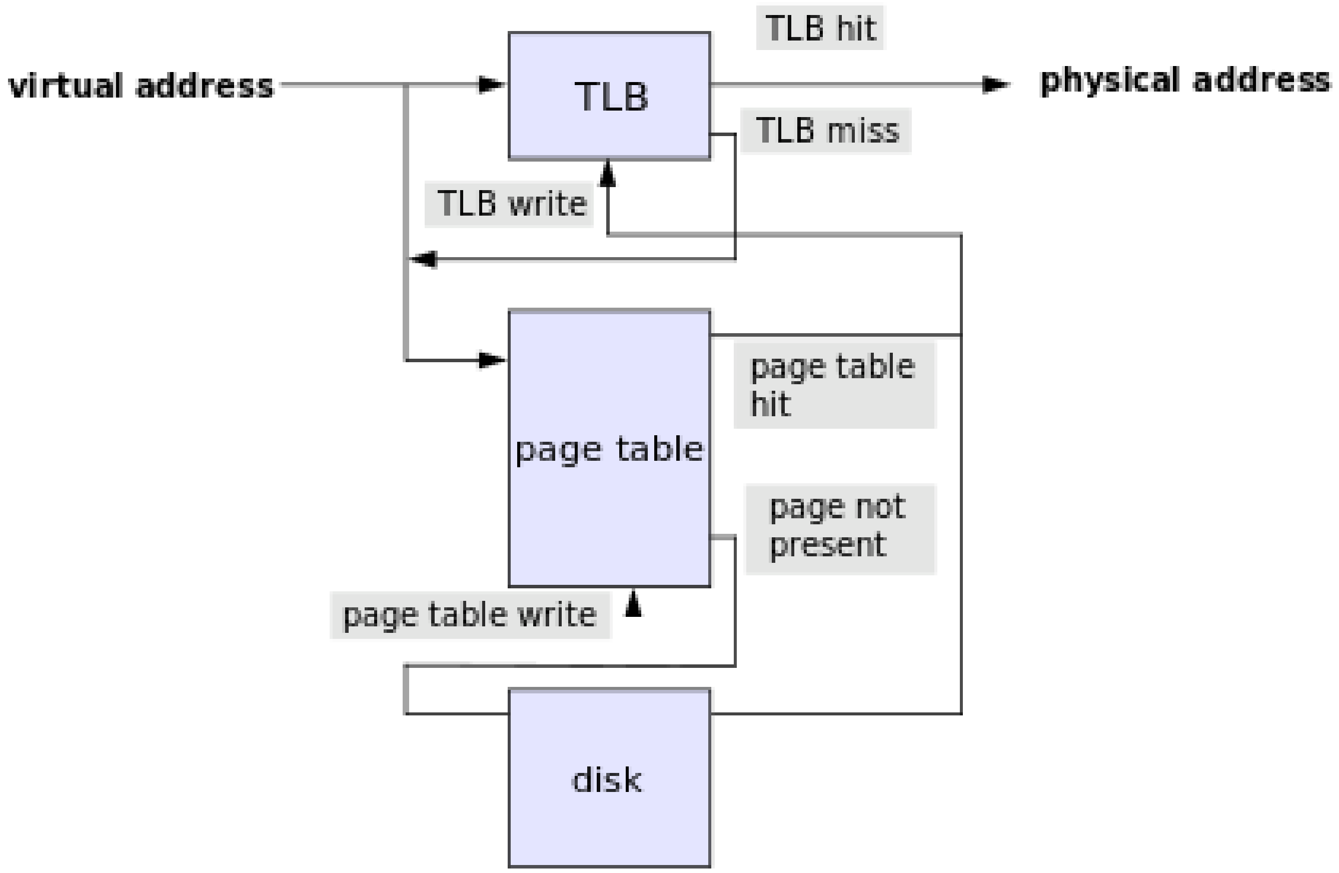
- The CPU's memory management unit (MMU) stores a cache of recently used mappings from the operating system's page table. This is called the translation lookaside buffer (TLB), which is an associative cache.
  - When a virtual address needs to be translated into a physical address, the TLB is searched first.
  - If a match is found (a TLB hit), the physical address is returned and memory access can continue.
  - However, if there is no match (called a TLB miss), the memory management unit, or the operating system TLB miss handler, will typically look up the address mapping in the page table to see whether a mapping exists (a page walk).
  - If one exists, it is written back to the TLB (this must be done, as the hardware accesses memory through the TLB in a virtual memory system), and the faulting instruction is restarted (this may happen in parallel as well).
  - The subsequent translation will find a TLB hit, and the memory access will continue.



Memory Fragmentation

Translational failure

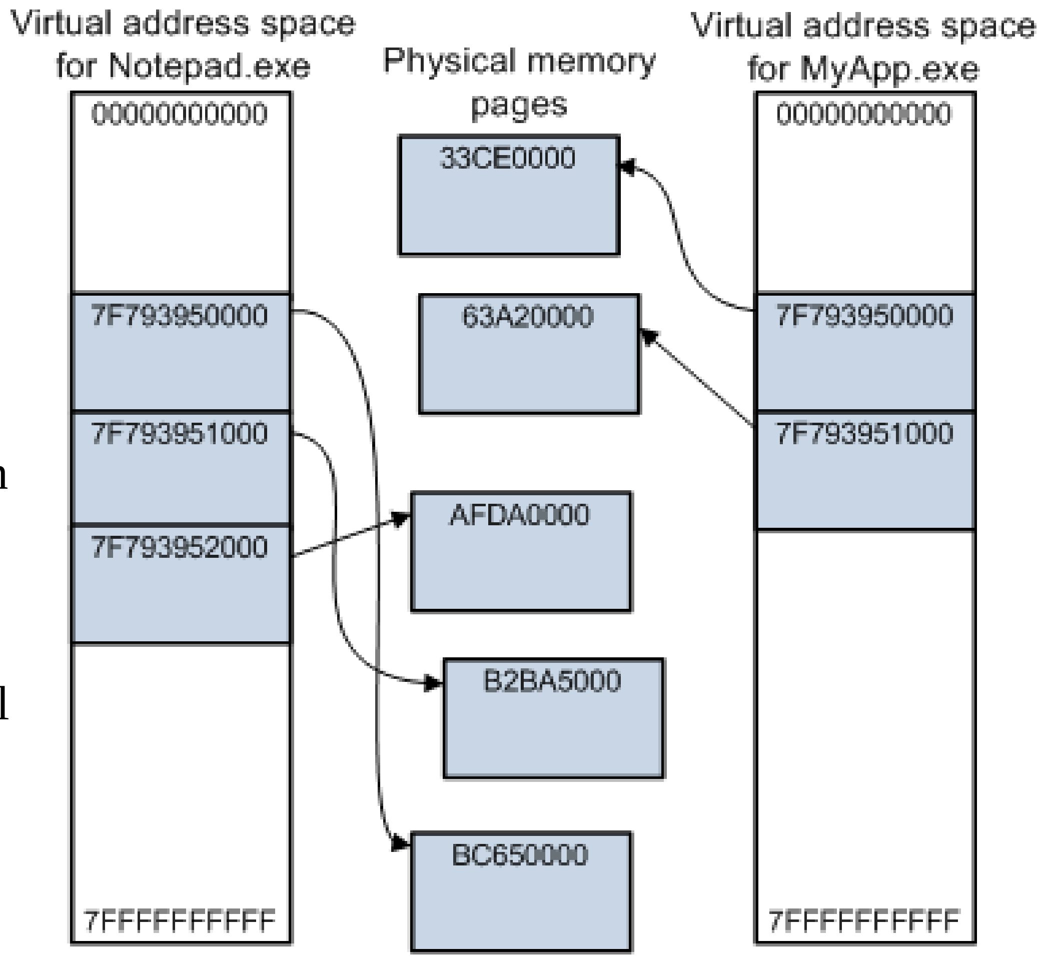
- The page table lookup may fail, triggering a page fault.
- The lookup may fail if there is no translation available for the virtual address, meaning that virtual address is invalid. This will typically occur because of a programming error, and the operating system must take some action to deal with the problem. On modern operating systems, it will cause a segmentation fault signal being sent to the offending program.
- The lookup may also fail if the page is currently not resident in physical memory. This will occur if the requested page has been moved out of physical memory to make room for another page. In this case the page is paged out to a secondary store located on a medium such as a hard disk drive (this secondary store, or "backing store", is often called a "swap partition" if it is a disk partition, or a swap file, "swapfile" or "page file" if it is a file). When this happens the page needs to be taken from disk and put back into physical memory. A similar mechanism is used for memory-mapped files, which are mapped to virtual memory and loaded to physical memory on demand.



Memory Fragmentation

Virtual memory

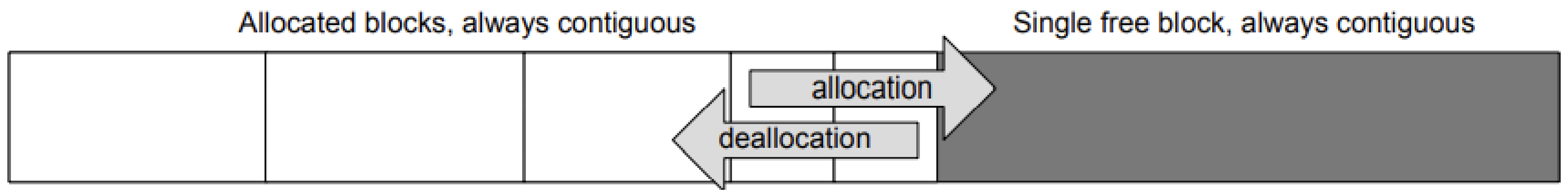
- Virtual memory makes software programming easier by hiding fragmentation of physical memory; by delegating to the kernel the burden of managing the memory hierarchy (eliminating the need for the program to handle overlays explicitly).
- Accessing memory through a virtual address has these advantages:
  1. A program can use a contiguous range of virtual addresses to access a large memory buffer that is not contiguous in physical memory.
  2. A program can use a range of virtual addresses to access a memory buffer that is larger than the available physical memory. As the supply of physical memory becomes small, the memory manager saves pages of physical memory (typically 4 kilobytes in size) to a disk file. Pages of data or code are moved between physical memory and the disk as needed.
  3. The virtual addresses used by different processes are isolated from each other. The code in one process cannot alter the physical memory that is being used by another process or the operating system.
- However, most console game engines still do not make use of virtual memory due to the inherent performance overhead.



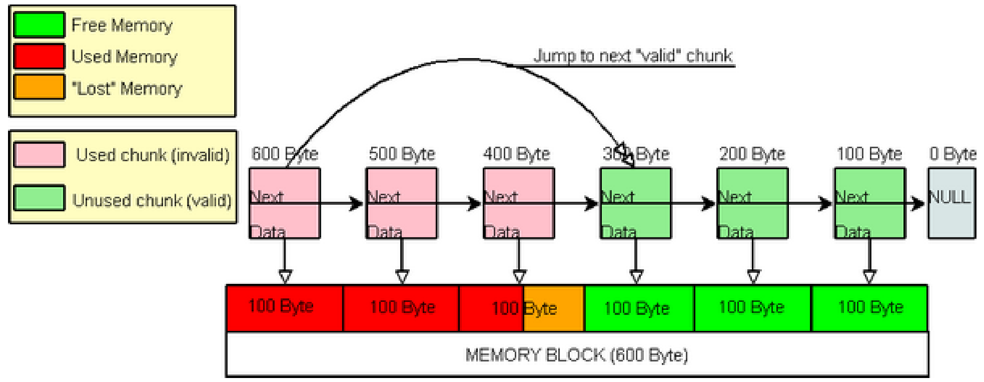
Memory Fragmentation

Stack and pool allocators

- The detrimental effects of memory fragmentation can be avoided by using stack and/or pool allocators.
  - A stack allocator is impervious to fragmentation because allocations are always contiguous, and blocks must be freed in an order opposite to that in which they were allocated.


Allocated blocks, always contiguous
Single free block, always contiguous
allocation
deallocation

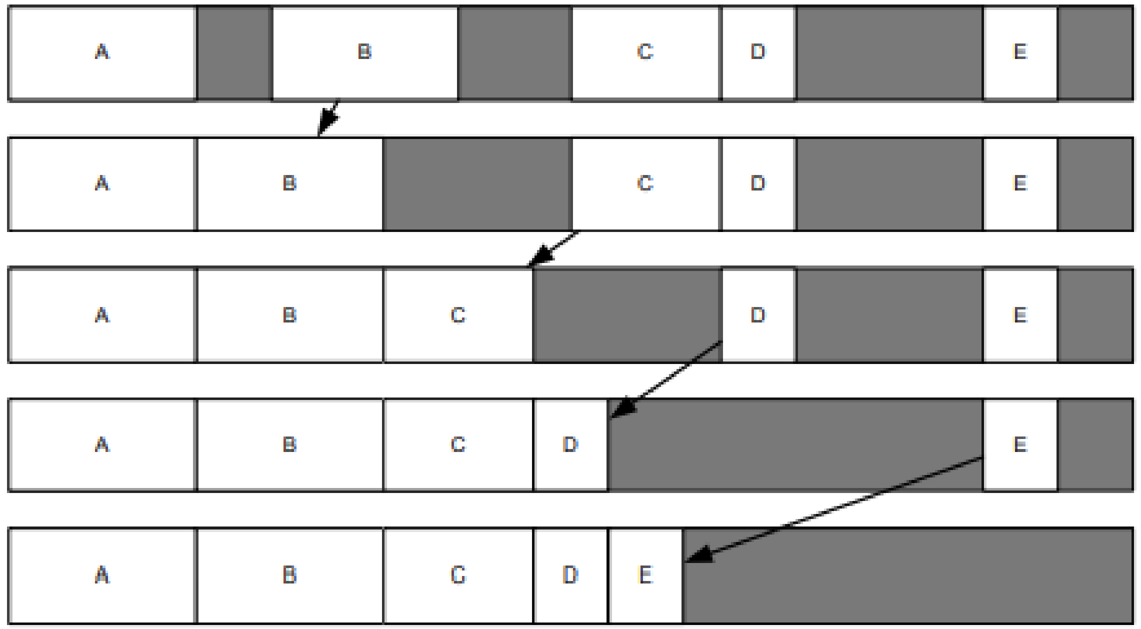
  - A pool allocator is also free from fragmentation problems. Pools do become fragmented, but the fragmentation never causes **premature out-of-memory conditions** as it does in a general-purpose heap. Pool allocation requests can never fail due to a lack of a large enough contiguous free block, because all of the blocks are exactly the same size.



Memory Fragmentation

Defragmentation

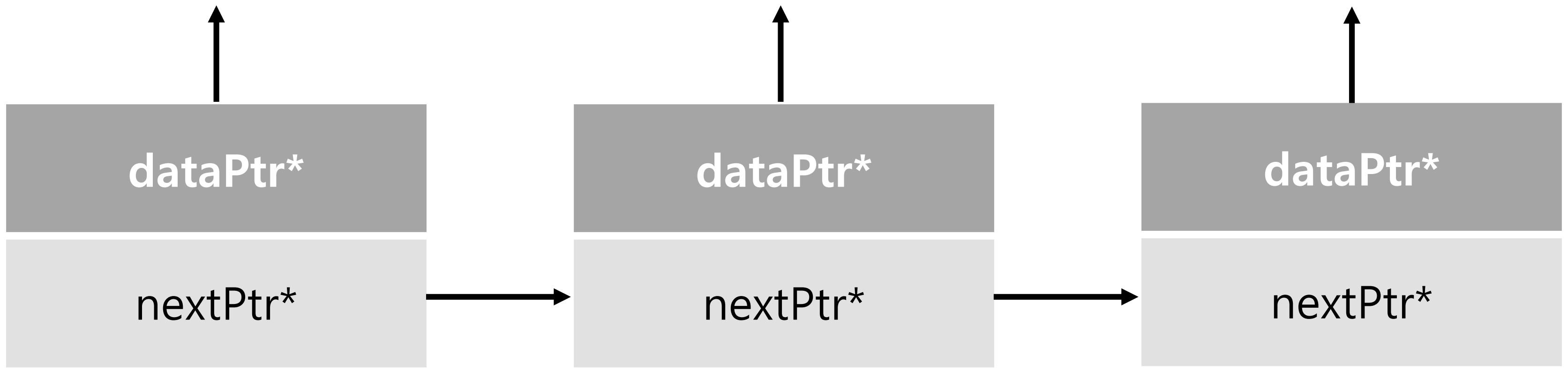
- When differently sized objects are being allocated and freed in a random order, neither a stack-based allocator nor a pool-allocator can be used.
- In such cases, fragmentation can be avoided by periodically *defragmenting* the heap.
- Defragmentation coalesce all of the free *holes* in the heap by shifting allocated blocks from higher memory address down to lower address.
    - For example, one algorithm first searches for a hole, takes the allocated block above the hole, and shifts it down to the start of the hole.
    - If this process is repeated, eventually all the allocated blocks will occupy a contiguous region of memory at the low end of the heap's address space, and all the holes will have bubbled up into one big hole at the high end of the heap.



Memory Fragmentation

Relocation

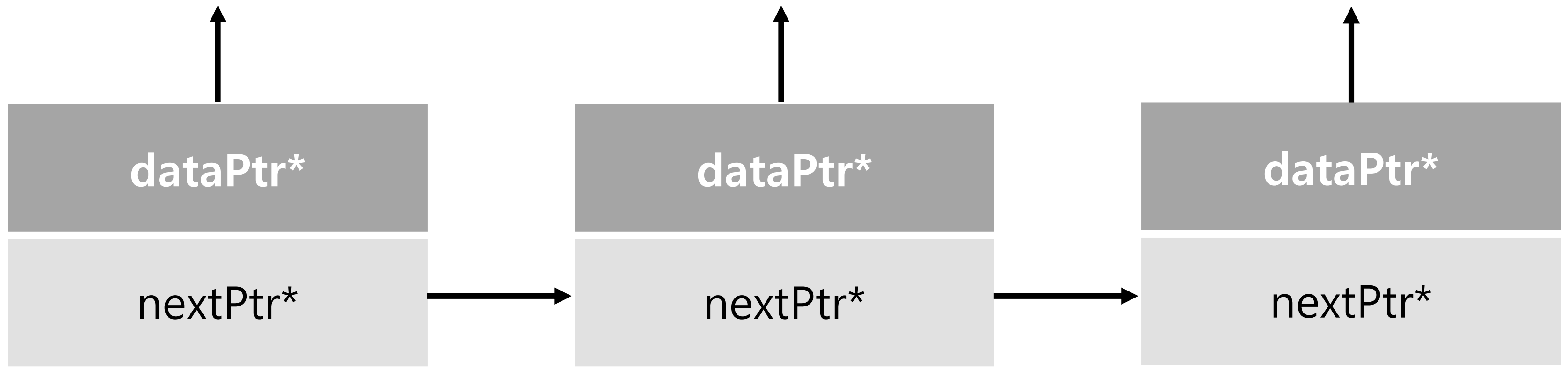
- The shifting of the allocated block will invalidate the corresponding pointer.
- The solution to this problem is to patch any and all pointers into a shifted memory block so that they point to the correct new address after the shift. This procedure is known as pointer *relocation*.
- Unfortunately, there is no general-purpose way to find all the pointers that point into a particular region of memory.
- So if we are going to support memory defragmentation in our game engine, programmers must either carefully keep track of all the pointers manually or use smart pointers or handles.
- Smart pointers:
  - A smart pointer is a small class that contains a pointer and acts as a pointer for most intents and purposes. But because a smart pointer is a class, it can be coded to handle memory relocation properly.
  - One approach is to arrange for all smart pointers to add themselves to a global linked list. Whenever a block of memory is shifted within the heap, the linked list of all smart pointers can be scanned, and each pointer that points into the shifted block of memory can be adjusted appropriately.



Memory Fragmentation

Relocation

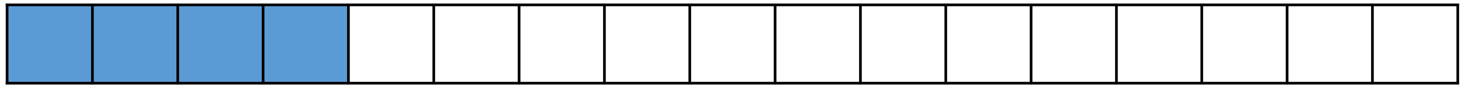
- Handles:
  - A handle usually implemented as an index into a non-relocatable table, which itself contains the pointers.
  - When an allocated block is shifted in memory, the handle table can be scanned and all relevant pointers found and updated automatically.
  - Because the handles are just indices into the pointer table, their values never change no matter how the memory blocks are shifted, so the objects that use the handles are never affected by memory relocation.



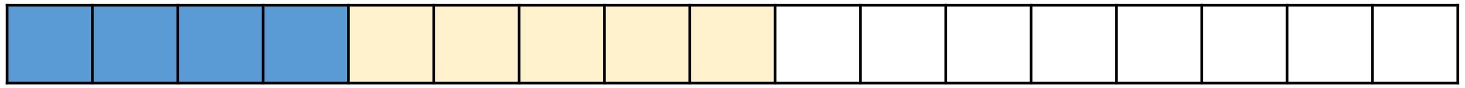
Memory management

Dynamic memory allocation

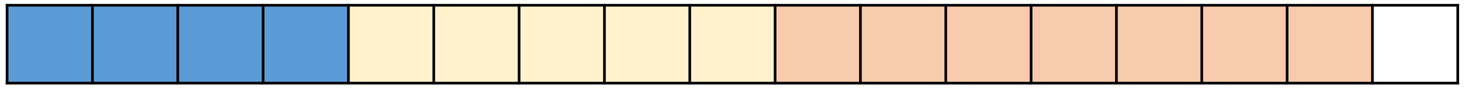
- Allocation examples:
  - p1 = malloc(4)



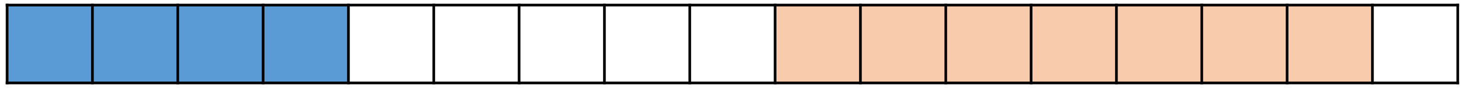
  - p2 = malloc(5)



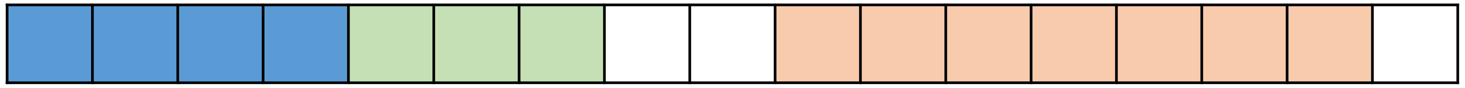
  - p3 = malloc(7)



  - free(p2)



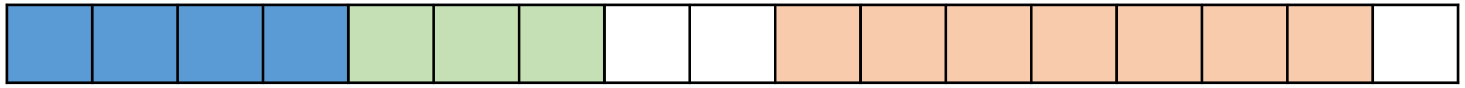
  - p4 = malloc(3)



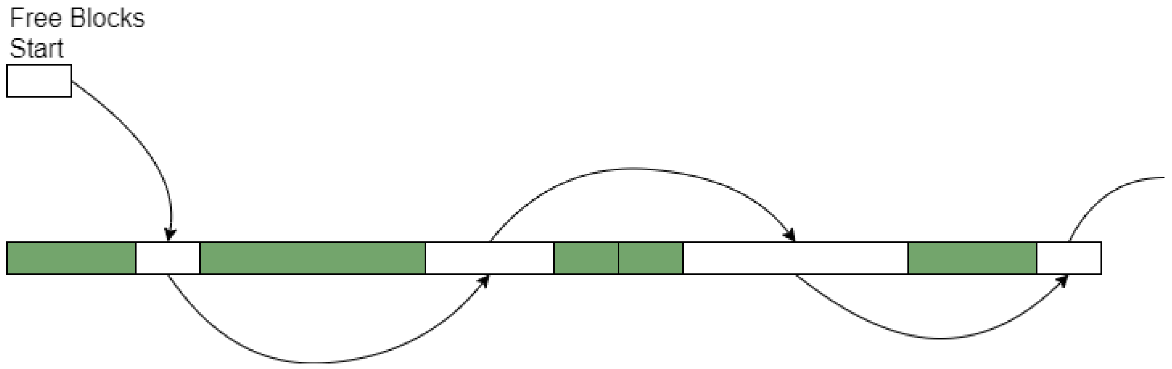
Memory management

Dynamic memory allocation

- External Fragmentation example:
  - p5 = malloc(3)



- External fragmentation arises when free memory is separated into small blocks and is interspersed by allocated memory.
- There are many algorithms on how to mange smaller chunks out of a large block and they differ in speed and memory consumption.
- However, such algorithms inevitably consume computational power.



an example of fragmentation handling[frag]

Memory management

Dynamic memory allocation

- A custom allocator can have better performance characteristics than the operating system’s heap allocator for two reasons:
  - A custom allocator can satisfy requests from a pre-allocated memory block. This allows it to run in user mode and entirely avoid the cost of context-switching into the operating system.
  - By making various assumptions about its usage patterns, a custom allocator can be much more efficient than a general-purpose heap allocator.

Memory management

Stack-based allocators

- Many game allocate memory in a stack-like fashion.
- Whenever a new game level is loaded, memory is allocated for it. Once the level has been loaded, little or no dynamic memory allocation takes place.
- At the conclusion of the level (level cleared or failed), its data is unloaded and all of its memory can be freed.
- It makes a lot of sense to use a stack-like data structure for these kinds of memory allocations.

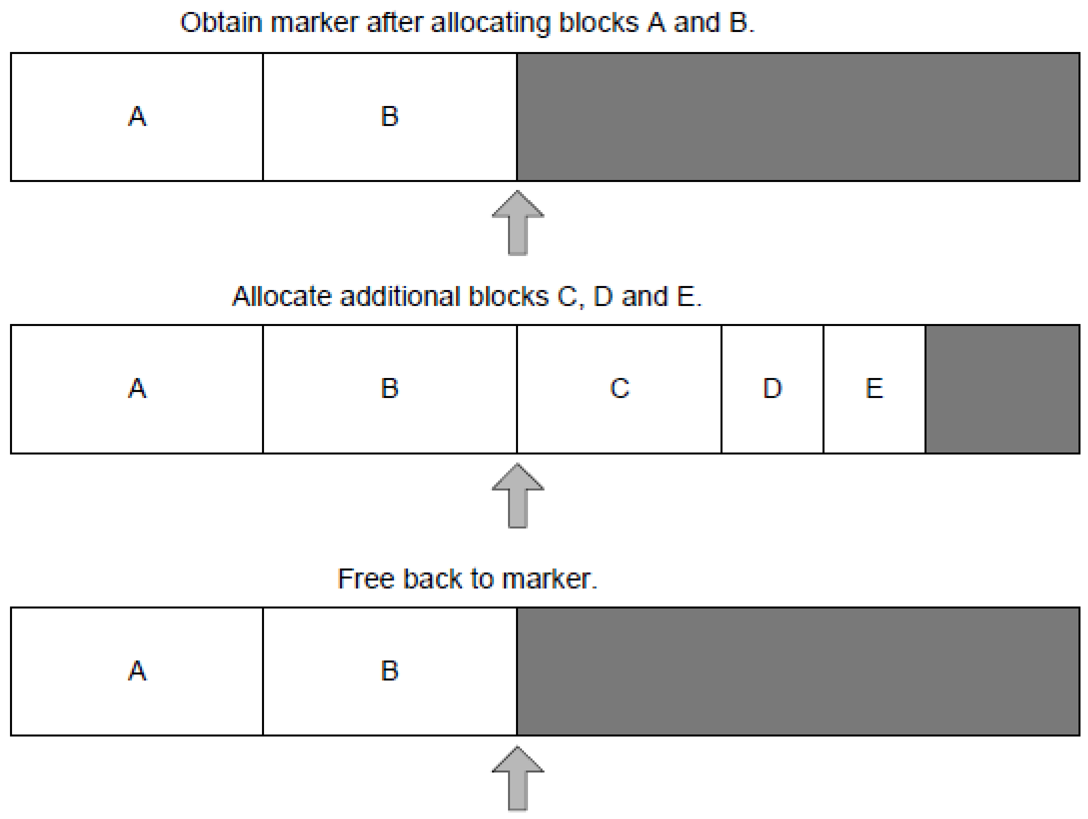
A stack allocator

- A stack allocator is very easy to implement.
- We simply allocate a large contiguous block of memory using malloc() or new(), or by declaring a global array of bytes.
- A pointer to the top of the stack is set and maintained.
- All memory addresses below this pointer are considered to be in use, and all addresses above it are considered to be free.
- The top pointer is initialized to the lowest memory address in the stack.
- Each allocation request simply moves the pointer up by the requested number of bytes.
- The most recently allocated block can be freed by simply moving the top pointer back down by the size of the block.

Memory management

A stack allocator

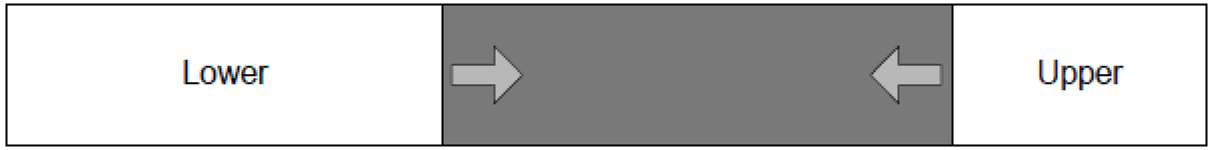
- Memory cannot be freed in an arbitrary order.
- All frees must be performed in an order opposite to that in which they were allocated.
- To enforce this, it provides a function that rolls the stack top back to a previously marked location, thereby freeing all blocks between the current top and the roll-back point.


Obtain marker after allocating blocks A and B.
A
B
Allocate additional blocks C, D and E.
A
B
C
D
E
Free back to marker.
A
B

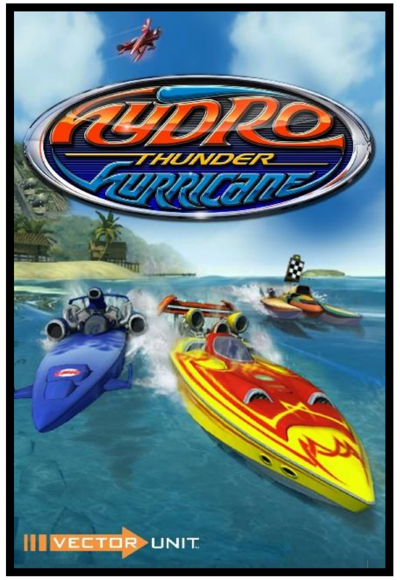
Memory management

Double-ended stack allocators

- Double-ended stack allocator uses two stack allocator - one that allocates up from the bottom of the block and one that allocates down from the top of the block.
- A double-ended stack allocator is useful because it uses memory more efficiently by allowing a trade-off to occur between the memory usage of the bottom stack and the memory usage of the top stack.
  - In some situations, both stacks may use roughly the same amount of memory and meet in the middle of the block.
  - In other situations, one of the two stacks may eat up a lot more memory than the other stack, but all allocation requests can still be satisfied as long as the total amount of memory requested is not larger than the block shared by the two stacks.



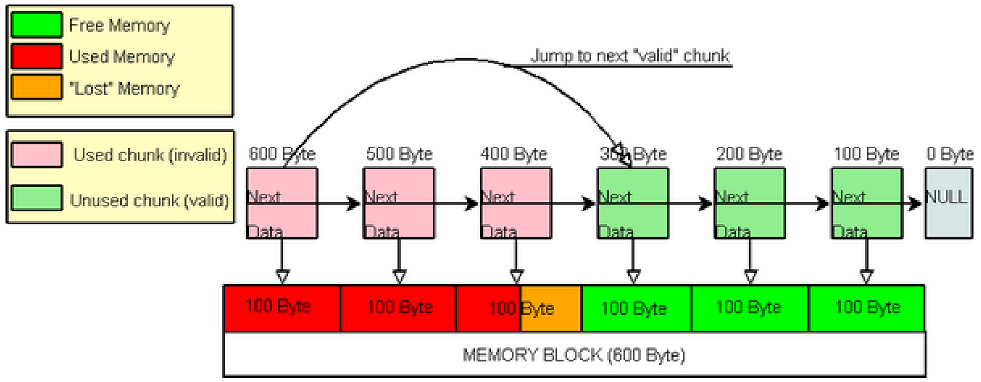
- In Midway’s *Hydro Thunder* arcade game, all memory allocations are made from a single large block of memory managed by a double-ended stack allocator.
  - The bottom stack is used for loading and unloading levels (race tracks).
  - The top stack is used for temporary memory blocks that are allocated and freed every frame.



Memory management

Pool allocators

- It’s quite common in game engine programming to allocate lots of small blocks of memory, each of which are the same size.
  - For example, we might want to allocate and free matrices, or iterators, or links in a linked list, or renderable mesh instances.
- For this type of memory allocation pattern, a pool allocator is often the perfect choice.
- A pool allocator works by pre-allocating a large block of memory whose size is an exact multiple of the size of the elements that will be allocated.
  - For example, a pool of 4×4 matrices would be an exact multiple of 64 bytes - that’s 16 elements per matrix times four bytes (32-bit floats) per element.
- Each element within the pool is added to a linked list of free elements; when the pool is first initialized, the free list contains all of the elements.
- When an allocation request is made, we simply grab the next free element off the free list and return it.
- When an element is freed, we simply track it back onto the free list.



memory pool example[pool]

Memory management

Single-frame and double-buffered memory allocators

- Virtually all game engines allocate at least some temporary data during the game loop. This data is either discarded at the end of each iteration of the loop or used on the next frame and then discarded.
- This allocation pattern is so common that many engines support *single-frame* and *double-buffered* allocators.

Single-frame allocators

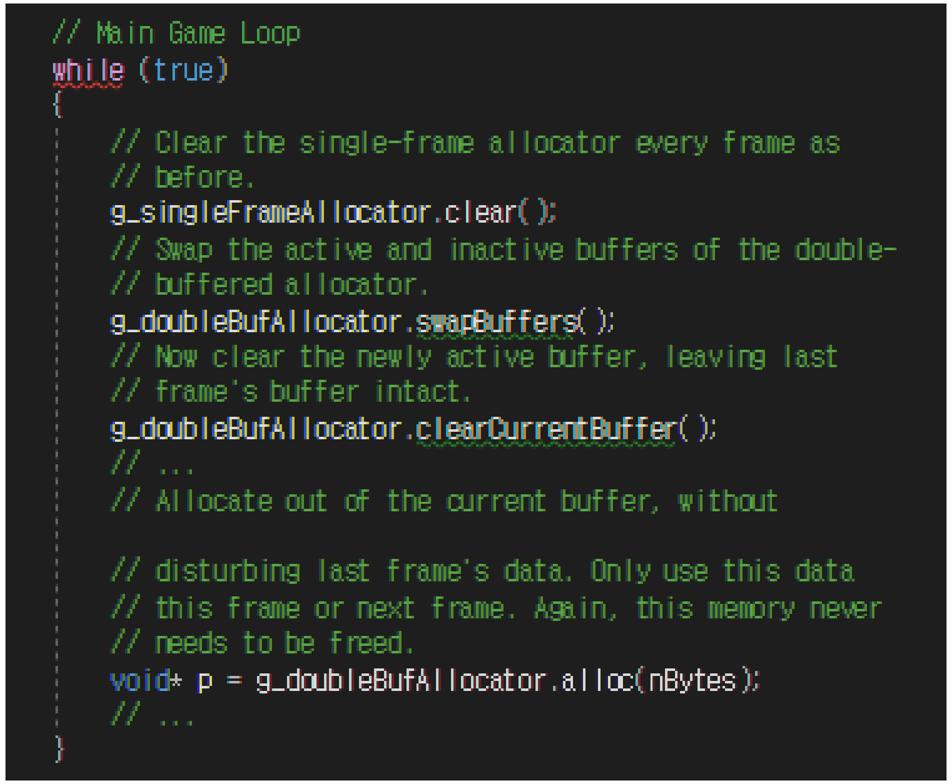
- A single-frame allocator is implemented by reserving a block of memory and managing it with a simple stack allocator as described above.
- At the beginning of each frame, the stack's top pointer is *cleared* to the bottom of the memory block.
- Allocations made during the frame grow toward the top of the block.
- Benefits:
  - Allocated memory needn't ever be freed - we can rely on the fact that the allocator will be cleared at the start of every frame.
  - Blindingly fast.
- Disadvantages:
  - Using a single-frame allocator requires a reasonable level of discipline on the part of the programmer.
  - Programmers must never cache a pointer to a single-frame memory block across the frame boundary.

memory pool example[pool]

Memory management

Double-buffered allocators

- A double-buffered allocator allows a block of memory allocated on frame *i* to be used on frame (*i*+1).
- To accomplish this, we create two single-frame stack allocators of equal size and then ping-pong between them every frame.
- This kind of allocator is extremely useful for caching the results of asynchronous processing on a multicore game console like Xbox360, PlayStation 3 or PlayStation 4.
- On frame *i*, the results can be stored into the buffer we provided.
- On frame (*i*+1), the buffers are swapped. Then the results of the previous frame are now in the inactive buffer, so they will not be cleared in this frame.



```
// Main Game Loop
while (true)
{
    // Clear the single-frame allocator every frame as
    // before.
    g_singleFrameAllocator.clear();
    // Swap the active and inactive buffers of the double-
    // buffered allocator.
    g_doubleBufAllocator.swapBuffers();
    // Now clear the newly active buffer, leaving last
    // frame's buffer intact.
    g_doubleBufAllocator.clearCurrentBuffer();
    // ...
    // Allocate out of the current buffer, without

    // disturbing last frame's data. Only use this data
    // this frame or next frame. Again, this memory never
    // needs to be freed.
    void* p = g_doubleBufAllocator.alloc(nBytes);
    // ...
}
```

Reference

- [frag] https://johnysswlab.com/the-price-of-dynamic-memory-allocation/
- [pool] https://www.codeproject.com/Articles/15527/C-Memory-Pool