**Basics of Memory Management**

• **What does main memory (RAM) contain**? Immediately after booting up, it contains the memory image of the kernel executable, typically in “low memory” or physical memory addresses starting from byte 0. Over time, the kernel allocates the rest of the memory to various running processes. For any process to execute on the CPU, the corresponding instructions and data must be available in main memory. The execution of a process on the CPU generates several requests for memory reads and writes. The memory subsystem of an OS must efficiently handle these requests. Note that contents of main memory may also be cached in the various levels of caches between the CPU and main memory, but the OS largely concerns itself with CPU caches.

• The OS tries to ensure that running processes reside in memory as far as possible. When the system is running low on memory, however, running processes may be moved to a region identified as swap space on a secondary storage device. Swapping processes out leads to a performance penalty and is avoided as far as possible.

• The memory addresses generated by the CPU when accessing the code and data of a running process are called logical addresses. The logical/virtual address space of a process ranges from 0 to a maximum value that depends on the CPU architecture (4GB for 32-bit addresses). Code and data in the program executable are assigned logical addresses in this space by the compiler. On the other hand, the addresses used by the actual memory hardware to locate information are called physical addresses. The physical address space of a system spans from 0 to a maximum value (determined by how much RAM the machine has). The OS plays a big role in mapping logical addresses of a process to physical addresses.

• When a new process is created, there are two ways of allocating memory to it: contiguous allocation or non-contiguous allocation. With contiguous allocation, the kernel tries to find a contiguous portion of physical memory to accommodate the new process. The kernel may use a best-fit or worst-fit or first-fit heuristic to identify a portion of unused memory. The problem with this type of allocation is (external) fragmentation of memory: sometimes, there is enough free memory in the system, but it is fragmented over several locations, so that a contiguous block cannot be found to satisfy a request. To avoid external fragmentation with contiguous allocation, the OS must periodically relocate running processes to other regions of memory.

• With contiguous allocation, the mapping between logical and physical addresses is straightforward. The OS maintains the starting physical address (or base) and the size of the memory image (or limit) for every process it places in memory. This base address value is added to the logical addresses to translate it to a physical address. While contiguous allocation has the benefit of simplicity, it is rarely used in modern operating systems due to the issues around memory fragmentation.

•The most common way of allocating memory to processes in modern operating systems is a type of non-contiguous allocation called paging. The logical address space of every process is divided into fixed-size (e.g., 4KB) chunks called pages. The physical memory is divided into fixed size chunks called frames, which are typically the same size as pages. Every process is allocated some free physical memory frames, and its logical pages are mapped to these physical frames. A page table of a process maintains this mapping from logical page numbers to physical frame numbers. The operating system maintains a separate page table for every process it creates. With paging, the issue of external fragmentation is eliminated, because any free frame can be allocated to any process. However, a smaller problem of internal fragmentation (the last page being partially filled) arises.

• Paging adds a level of indirection between logical and physical addressing, and this provides several benefits. There is now a clean separation between logical addresses and physical addresses: the compiler can assign addresses to code and data without worrying about where the program would reside in memory. Paging also makes sharing of memory between processes easier: two logical pages can point to the same physical frame in the page tables. However, paging adds the additional overhead of address translation.

• How are virtual addresses mapped to physical addresses with paging? The virtual address is first split into a page number and an offset within the page. The page number is mapped to the physical frame number by looking up the page table of the process. The physical address is then obtained from the physical frame number and the offset within the frame. Who does this translation from CPU-generated logical addresses to physical addresses? The OS takes the responsibility of constructing and maintaining page tables during the lifecycle of a process; the PCB contains a pointer to the page table. The OS also maintains a pointer to the page table of the current process in a special CPU register (CR3 in x86) and updates this pointer during context switches. A specialized piece of hardware called the memory management unit / MMU then uses the page table to translate logical addresses requested by the CPU to physical addresses using the logic described above.

• Segmentation is another way of allocating physical memory to a process. With segmentation, the process memory image is divided into segments corresponding to code, data, stack, and so on. Each segment is then given a contiguous chunk of physical memory. A segment table stores the mapping between the segments of a process and the base/limit addresses of that segment. Most modern operating systems, however, use paging, or a combination of segmentation and paging. Unix-like operating systems make minimal use of segmentation.

• Segmentation can be used to create a hierarchical address space, where the segment number forms the most significant bits of the logical address. However, Unix-like operating systems mostly use a flat address model on modern CPU architectures in user mode. Most C compilers for Linux on x86 today, for example, generate logical addresses from 0 to 4GB, and all segment registers are set to zero. The values of segment registers only change when moving from user mode to kernel mode and vice versa.

•At boot time, the first pieces of code that executes must work only with physical addresses. Once the boot loader executes code to construct page tables and turn the MMU on, the kernel code will start working with virtual addresses. Therefore, the booting process must deal with some complexities arising from this transition.

•On all memory accesses, the memory hardware checks that the memory access is indeed allowed and raises a trap if it detects an illegal access (e.g., user mode process accessing kernel memory or the memory of another process). With contiguous memory allocation, it is easy to check if the memory address generated by the CPU indeed belongs to the process or not: before accessing the memory, one can check if the requested address lies in the range [base, base+limit). With paging, every page page has a set of bits indicating permissions. During address translation, the hardware checks that the requested logical address has a corresponding page table entry with sufficient permissions, and raises a trap if an illegal access is detected.

•With a separation of virtual address space and physical address space in modern operating systems, each process can have a large virtual address space. In fact, the combined virtual address space of all processes can be much larger than the physical memory available on the machine, and logical pages can be mapped to physical frames only on a need basis. This concept is called demand paging and is quite common in modern operating systems. With demand paging, the memory allocated to a process is also called virtual memory, because not all of it corresponds to physical memory in hardware.

**Address Space of a Process**

• The virtual address space of a process has two main parts: the user part containing the code/data of the process itself, and the kernel code/data. For example, on a 32-bit x86 system, addresses 0-3GB of the virtual address space of a process could contain user data, and addresses 3-4GB could point to the kernel. The page table of every process contains mappings for the user pages and the kernel pages. The kernel page table entries are common for all processes (as there is only one physical copy of the kernel in memory), while the user page table entries are obviously different.

• Note that every physical memory address that is in use will be mapped into the virtual address space of at least one process. That is, the physical memory address will correspond to a virtual address in the page table of some process. Physical memory that is not mapped into the address space of any process is by definition not accessible, since (almost) all accesses to memory go via the MMU. Some physical memory can be mapped multiple times, e.g., kernel code and data is mapped into the virtual address space of every process.

• Why is the kernel mapped into the address space of every process? Having the kernel in every address space makes it easy to execute kernel code while in kernel mode: one does not have to switch page tables or anything, and executing kernel code is as simple as jumping to a memory location in the kernel part of the address space. Page tables for kernel pages have a special protection bit set, and the CPU must be in kernel mode to access these pages, to protect against rogue processes.

• The user part of the address space contains the executable code of the process and statically allocated data. It also contains a heap for dynamic memory allocation, and a stack, with the heap and stack growing in opposite directions towards each other. Dynamically linked libraries, memory-mapped files, and other such things also form a part of the virtual address space. By assigning a part of the virtual address space to memory mapped files, the data in these files can be accessed just like any other variable in main memory, and not via disk reads and writes. The virtual address space in Linux is divided into memory areas or maps for each of the entities mentioned above.

• The kernel part of the address space contains the kernel code and data. For example, it has various kernel data structures like the list of processes, free pages to allocate to new processes, and so on. The virtual addresses assigned to kernel code and data are the same across all processes.

• One important concept to understand here is that most physical memory will be mapped (at least) twice, once to the kernel part of the address space of processes, and once to the user part of some process. To see why, note that the kernel maintains a list of free frames/pages, which are subsequently allocated to store user process images. Suppose a free frame of size N bytes is assigned a virtual address, say V, by the kernel. Suppose the kernel maintains a 4-byte pointer to this free page, whose value is simply the starting virtual address V of the free page. Even though the kernel only needs this pointer variable to track the page, note that it cannot assign the virtual addresses [V , V + N) to any other variable, because these addresses refer to the memory in that page, and will be used by the kernel to read/write data into that free page. That is, a free page blocks out a page-sized chunk of the kernel address space. Now, when this page is allocated to a new process, the process will assign a different virtual address range to it (say, [U, U + N)), from the user part of its virtual address space, which will be used by the process to read/write data in user mode. So the same physical frame will also have blocked out another chunk of virtual addresses in the process, this time from the user part. That is, the same physical memory is mapped twice, once into the kernel part of the address space (so that the kernel can refer to it), and once into the user part of the address space of a process (so that the process can refer to it in user mode).

• Is this double consumption of virtual addresses a problem? In architectures where virtual address spaces are much larger than the physical memory, this duplication is not a problem, and it is alright to have one byte of physical memory block out two or more bytes of virtual address space. However, in systems with smaller virtual address spaces (due to smaller number of bits available to store memory addresses in registers), one of the following will happen: either the entire physical memory will not be used (as in the case of xv6), or more commonly, some part of user memory will not be mapped in the kernel address space all the time (as in the case of Linux). That is, once the kernel allocates a free page to a process, it will remove its page table mappings that point to that physical memory and use those freed up virtual addresses to point to something else. Subsequently, this physical memory will only be accessible from the user mode of a process, because only the user virtual addresses point to it in the page table. Such memory is called “high memory” in Linux, and high memory is mapped into the kernel address space (i.e., virtual addresses are allocated from the kernel portion of the virtual memory) only on a need basis.

**Memory API**

**Types of Memory**

In running a C program, there are two types of memory that are allocated. The first is called **stack memory**, and allocations and deallocations of it are managed implicitly by the compiler for you, the programmer; for this reason, it is sometimes called **automatic memory**. Declaring memory on the stack in C is easy. For example, let’s say you need some space in a function func() for an integer, called x. To declare such a piece of memory, you just do something like this:

void func()

{

int x; // declares an integer on the stack

...

}

The compiler does the rest, making sure to make space on the stack when you call into func(). When you return from the function, the compiler deallocates the memory for you; thus, if you want some information to live beyond the call invocation, you had better not leave that information on the stack.

It is this need for long-lived memory that gets us to the second type of memory, called heap memory, where all allocations and deallocations are explicitly handled by you, the programmer. A heavy responsibility, no doubt! And certainly, the cause of many bugs. But if you are careful and pay attention, you will use such interfaces correctly and without too much trouble. Here is an example of how one might allocate an integer on the heap:

void func()

{

int \*x = (int \*) malloc(sizeof(int));

...

}

A couple of notes about this small code snippet. First, you might notice that both stack and heap allocation occur on this line: first the compiler knows to make room for a pointer to an integer when it sees your declaration of said pointer (int \*x); subsequently, when the program calls malloc(), it requests space for an integer on the heap; the routine returns the address of such an integer (upon success, or NULL on failure), which is then stored on the stack for use by the program. Because of its explicit nature, and because of its more varied usage, heap memory presents more challenges to both users and systems.

The malloc() call is quite simple: you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns NULL2 . The manual page shows what you need to do to use malloc; type man malloc at the command line and you will see:

#include< stdlib.h >

...

void \*malloc(size\_t size);

From this information, you can see that all you need to do is include the header file stdlib.h to use malloc. In fact, you don’t really need to even do this, as the C library, which all C programs link with by default, has the code for malloc() inside of it; adding the header just lets the compiler check whether you are calling malloc() correctly (e.g., passing the right number of arguments to it, of the right type). The single parameter malloc() takes is of type size t which simply describes how many bytes you need. However, most programmers do not type in a number here directly (such as 10); indeed, it would be considered poor form to do so. Instead, various routines and macros are utilized. For example, to allocate space for a double-precision floating point value, you simply do this:

double \*d = (double \*) malloc(sizeof(double));

Wow, that’s lot of double-ing! This invocation of malloc() uses the sizeof() operator to request the right amount of space; in C, this is generally thought of as a compile-time operator, meaning that the actual size is known at compile time and thus a number (in this case, 8, for a double) is substituted as the argument to malloc(). For this reason, sizeof() is correctly thought of as an operator and not a function call (a function call would take place at run time). You can also pass in the name of a variable (and not just a type) to sizeof(), but in some cases you may not get the desired results, so be careful. For example, let’s look at the following code snippet:

int \*x = malloc(10 \* sizeof(int)); printf("%d\n", sizeof(x));

In the first line, we’ve declared space for an array of 10 integers, which is fine and dandy. However, when we use sizeof() in the next line, it returns a small value, such as 4 (on 32-bit machines) or 8 (on 64-bit machines). The reason is that in this case, sizeof() thinks we are simply asking how big a pointer to an integer is, not how much memory we have dynamically allocated. However, sometimes sizeof() does work as you might expect: int x[10]; printf("%d\n", sizeof(x)); In this case, there is enough static information for the compiler to know that 40 bytes have been allocated. Another place to be careful is with strings. When declaring space for a string, use the following idiom: malloc(strlen(s) + 1), which gets the length of the string using the function strlen(), and adds 1 to it in order to make room for the end-of-string character. Using sizeof() may lead to trouble here.

You might also notice that malloc() returns a pointer to type void. Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a cast; in our example above, the programmer casts the return type of malloc() to a pointer to a double. Casting doesn’t really accomplish anything, other than tell the compiler and other programmers who might be reading your code: “yeah, I know what I’m doing.” By casting the result of malloc(), the programmer is just giving some reassurance; the cast is not needed for the correctness.

The free() Call As it turns out, allocating memory is the easy part of the equation; knowing when, how, and even if to free memory is the hard part. To free heap memory that is no longer in use, programmers simply call

free(): int \*x = malloc(10 \* sizeof(int)); ... free(x);

The routine takes one argument, a pointer returned by malloc(). Thus, you might notice, the size of the allocated region is not passed in by the user, and must be tracked by the memory-allocation library itself.

Other Calls:

There are a few other calls that the memory-allocation library supports. For example, calloc() allocates memory and also zeroes it before returning; this prevents some errors where you assume that memory is zeroed and forget to initialize it yourself (see the paragraph on “uninitialized reads” above). The routine realloc() can also be useful, when you’ve allocated space for something (say, an array), and then need to add something to it: realloc() makes a new larger region of memory, copies the old region into it, and returns the pointer to the new region.

Common Errors

There are a number of common errors that arise in the use of malloc() and free().

1. Forgetting To Allocate Memory
2. Not Allocating Enough Memory
3. Forgetting to Initialize Allocated Memory
4. Forgetting To Free Memory
5. Freeing Memory Before You Are Done With It
6. Freeing Memory Repeatedly
7. Calling free() Incorrectly

**Text & Reference books:**

1. Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci- Dusseau, Arpaci-Dusseau Books, May, (2014).