COMS3010: Operating Systems
Tutorial 3: Memory & Address Translation

October 2019

Contents

1 Terminology

- **Physical Memory** - The hardware memory available to the system. Physical addresses allow the operating system to access physical memory.

- **Virtual Memory** - Virtual memory is a memory management technique in which every process operates in its own address space, under the assumption that it has the entire address space to itself. A virtual address has to be translated into a physical address to access the system's memory.

- **Memory Management Unit** - The memory management unit (MMU) is a computer hardware unit responsible for translating a process's virtual addresses into the corresponding physical addresses for accessing physical memory. It performs the calculations associated with mapping virtual to physical addresses and populates the address translation structures.

- **Page** - In paged virtual memory systems, a page is a contiguous fixed-sized chunk of memory; it is a unit in which memory is allocated by the operating system to processes.

- **Page Table** - The data structure used by a paged virtual memory system to store the mapping between processes' virtual addresses and the system's physical addresses.

- **sbrk** - A low level C function that can be used to increase change the amount of memory allocated to a program. `sbrk(int increment)` will change the amount of memory allocated to the program by 'increment' bytes, and if 'increment' is non-negative, it will return a pointer to the previous position of the heap break, which is the beginning of the newly allocated data.

2 Questions

2.1 Simple Malloc

Write a basic version of malloc using `sbrk`, assuming memory never needs to be freed.

```
void* malloc( size_t size ) {
    ----------------;
}
```

Answer:

```
void* malloc( size_t size ) {
    return sbrk(size);
}
```

2.2 Address Translation Schemes

1. What is the main disadvantage of segmentation schemes when compared to paging?

   Answer:
   Segmentation causes external fragmentation since segments are variable sized and each needs a contiguous portion of physical memory. Since pages are of a fixed size and need not be contiguous, they can fit into any previously freed spaces and program memory can be extended without needing to reshuffle existing pages (provided enough physical memory is available).

2. What happens in a paging scheme when the system starts running out of main memory?

   Answer:
   When the system is running out of space for physical pages in main memory, the operating system will increase the rate of swapping out unused pages to make space. If too many pages are still in use with new ones being requested, this can lead to thrashing.

3. Using paging, what happens when a program tries to access a valid memory address not in main memory?

   Answer:
   The memory management unit will throw a page fault. The page fault handler in the operating system will initiate fetching the data from disk and attempt to find a free physical page to use. If a free page is not found, another page is swapped out and replaced. The page table is then updated to reflect that the new page is in main memory, its valid bit is set to 1, and the memory management unit will access the page again and succeed.

4. Suppose we are running a virtual memory system on a machine with 32 bit addresses. Assume that the page size is 1KB = 1024 bytes and memory is byte-addressed.

   - How many bits do we have available to refer to the virtual page number?

     Answer:
     We need 10 bits to refer to 1024 bytes within a page ($2^{10} = 1024$). Thus, we are left with 22 bits to refer to page numbers.

   - What is the maximal number of entries we can store in a page table?

     Answer:
     The page table needs to contain an entry for every virtual page. As we have 22 bits to refer to pages, we can reference 2^{22} virtual pages. This is, thus, the maximal number of entries we should be able to store in a page table.

   - What is the maximal size of the virtual address space on our system?

     Answer:
     We have the capacity for $2^{22} = 4194304$ pages, and each page is 1024 bytes each. Thus, the maximal size of the address space is $4194304 \times 1024 = 4294967296$ bytes, which is 4 gigabytes.

5. Suppose we now changed the page size to be 4KB = 4096 bytes.

   - How many bits do we have available to refer to the virtual page number?

     Answer:
     We need 12 bits to refer to 4096 bytes within a page ($2^{12} = 4096$). Thus, we are left with 20 bits to refer to page numbers.

   - What is the maximal number of entries we can store in a page table?

     Answer:
     The page table needs to contain an entry for every virtual page. As we have 20 bits to refer to pages, we can reference 2^{20} virtual pages. This is, thus, the maximal number of entries we should be able to store in a page table.

   - What is the maximal size of the virtual address space on our system?

     Answer:
     We have the capacity for 2^{20} pages, and each page is 4096 bytes each. Thus, the maximal size of the address space is $2^{20} \times 4096 = 4294967296$ bytes, which is 4 gigabytes. Notice that the size of the virtual address space is, thus, independent of the page size.

6. We have a (toy) system with the following parameters. The word size is 8 bits; the page size is 4 bits; the virtual address space size is 12 bits; the physical memory size is 20 bits. We consecutively store 1-bit sized chunks named a, b, c, d, e, f, g, h, i, j, k, l in the virtual address space. We count both virtual pages and frames in physical memory starting from 0. The page table maps virtual page 0 to frame 4, virtual page 1 to frame 3, and virtual page 2 to frame 1. Compute the physical addresses of the following chunks (write your answer in hexadecimal notation):

   - b;

     Answer:
     The virtual address is 00000001. Pages are 4-bit big, so the offset is 2 bits. Thus the page number is 0 (000000) and the offset is 01. Page 0 gets mapped to frame 4 (000100); appending the offset, we get the physical address 00010001, that is 0x11.

   - g;

     Answer:
     The virtual address is 00000110. Thus the page number is 1 (000001) and the offset is 10. Page 1 gets mapped to frame 3 (000011); appending the offset, we get the physical address 00001110, that is 0xe.

   - j.

     Answer:
     The virtual address is 00001001. Thus the page number is 2 (000010) and the offset is 01. Page 2 gets mapped to frame 1 (000001); appending the offset, we get the physical address 00000101, that is 0x5.

7. Suppose we are running a virtual memory system on a machine with 32 bit addresses. Assume that the page size is 4KB = 4096 bytes and that a page-table entry takes up 4 bytes. We are using two-level page tables, using the first 10 bits of the virtual address to refer to the top-level page table, the second 10 bits to refer to the second-level page table, and the remaining 12 bits to refer to the offset. All our pages are 4KB big, regardless of whether they belong to the page table or to the process's virtual address space. Why do we need only 10 bits of the virtual address to handle the pages from the page table, but 12 bits to handle the pages belonging to the process's virtual address space?

   Answer:
   In pages belonging to the virtual address space, we need to be able to reference every single byte, thus we need to reference $4096 = 10^{12}$ "things", for which we need 12 bits. In pages belonging to the page table, we need to be able to reference every page-table entry, of which we have $\frac{4096}{4} = 1024$; thus, we need to reference $1024 = 10^{10}$ "things", for which we need 10 bits.

8. We are using two-level page tables to run the virtual memory system. Upon decoding an instruction, we take a page fault. Does this mean that the page we are looking for is not in the main memory of the machine?

   Answer:
   No. We could have taken the page fault because the second-level page table is not in the memory. The page itself may well be in the memory.

9. We are using two-level page tables to run the virtual memory system and our next instruction to be executed is a load instruction. Suppose none of the translations we need to know to execute the instruction is cached. How many times do we need to translate a virtual address to a physical address to be able to execute the instruction?

   Answer:
   First, we need to find out the physical address of the instruction. We need the physical addresses of
   - the 1st-level page table (1st translation), which we cache;
   - the 2nd-level page table referring to the page containing the instruction (2nd translation);
   - the page containing the instruction (3rd translation);
   - the 1st-level page table, which we have just cached, so there's no need to do a translation here;
   - the 2nd-level page table referring to the page containing the data (4th translation);
   - the page containing the data (5th translation).

   So, in all, we need 5 translations. We cache all the translations as we go along, but it's highly unlikely that we'll reuse any of the translations except the very first one, for the 1st-level page table.

2.3 Page Allocation

Consider a system with 8-bit virtual memory addresses, 8 pages of virtual memory, and 4 pages of physical memory.

1. How large is each page in bytes? Assume memory is byte addressed.

   Answer:
   32 bytes.
   The virtual addresses have 8 bits. There are 8 pages so we need 3 bits to reference the page number, leaving 5 bits for the byte offset. Hence the pages are $2^5 = 32$ bytes large.

2. If the number of virtual memory pages doubles, how does the page size change?

   Answer:
   The page size is halved to 16 bytes.
   The number of virtual pages doubles from 8 to 16, so we now need 4 bits for the page number and have 4 bits for the offset. So the new page size would be $2^4 = 16$ bytes.

2.3.1 Page Table

```
int main(void) {
    char *args[5];
    int i;
    for (i = 0; i < 5; i++) {
        // Assume we allocate an entire page every iteration
        args[i] = (char*) malloc(PAGE_SIZE);
    }
    printf( "%s", args[0] );
    return 0;
}
```

Suppose the program running the above code has the following memory allocation and page table.

Memory Segment	Virtual Page Number	Physical Page Number
N/A	000	NULL
Code Segment	001	10
Heap	010	11
N/A	011	NULL
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Sketch what the page table looks like after running the program, just before the program returns. Page out the least recently used page of memory if a page needs to be allocated when physical memory is full, write `PAGEOUT` as the physical page number when this happens. Assume that the stack will never exceed one page of memory.

Answer:

Note that the code segment and stack are in use during the loop, so the least recently used page is always one of the heap pages.

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	00
Code Segment	001	10
Heap	010	11
N/A	011	NULL
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	00
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	11
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	PAGEOUT
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	11
Heap	100	00
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	PAGEOUT
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	PAGEOUT
Heap	100	00
Heap	101	11
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	PAGEOUT
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	PAGEOUT
Heap	100	PAGEOUT
Heap	101	11
Heap	110	00
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	11
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	PAGEOUT
Heap	100	PAGEOUT
Heap	101	PAGEOUT
Heap	110	00
Stack	111	01