COMS2001: Operating Systems
Tutorial 3: Memory & Address Translation

October 12, 2017

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 ) {
    ----------------;
}
```

2.2 Address Translation Schemes

1. What is the main disadvantage of segmentation schemes when compared to paging?
2. What happens in a paging scheme when the system starts running out of main memory?
3. Using paging, what happens when a program tries to access a valid memory address not in main memory?
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.
   - How many bits do we have available to refer to the virtual page number?
   - What is the maximal number of entries we can store in a page table?
   - What is the maximal size of the virtual address space on our system?

   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?
   - What is the maximal number of entries we can store in a page table?
   - What is the maximal size of the virtual address space on our system?
5. 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 consequtively 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 hexidecimal notation):
   - b;
   - g;
   - j.
6. Suppose we are running a virtual memory system on a machine with 32 bit addresses. Assume that the page size is 4KB = 1024 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 wether 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?

7. 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 main memory of the machine?

8. 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 are cached. How many time we need to translate a virtual address to a physical address to be able to execute the instruction?

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.

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

2.3.1 Page Table

```c
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.