COMS2001: Operating Systems
Tutorial 3: Memory & Address Translation

October 20, 2016

Contents

1 Terminology

- **Calling Conventions** - Describe the interface of called code. They define the order in which parameters are allocated, how parameters are passed, which registers the callee must preserve for the caller, how the stack is prepared and restored, etc.

- **Stack Frame** - The stack frame of a function represents its argument data, local variables, and data used to return to the caller function. Starting from the higher memory addresses, the order is typically: parameter values, return address, saved %ebp, then local variables.

- **ESP Register** - On x86 architecture, the %esp register holds the address of the last thing on the stack. The %esp pointer gets moved as things get added or removed from the stack.

- **EBP Register** - The %ebp register stores the position of the beginning of the current stack frame.

- **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 Calling Conventions

Sketch the stack frame of `helper` before it returns.

```
void helper(char* str, int len) {
    char word[len];
    strcpy(word, str, len);
    printf("%s", word);
    return;
}

int main(int argc, char *argv[]) {
    char* str = "Yes";
    helper(str, 3);
}
```

2.3 Argument Passing

Fill out the following function to copy integer arguments from `argv` onto the stack. You can change the %esp by modifying `if_->esp`. Recall that the %esp pointer points to the last thing on a stack frame, and that the stack 'grows' from higher to lower memory addresses.

```
void extend_stack(size_t argc, int* argv, struct intr_frame* if_) {
    // if_->esp has already been loaded
    int i;

    for (_______; ________; _______) {
        ___________ = argv[i];
        if_->esp = ________________;
    }
}
```

2.4 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?

2.5 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?

2.5.1 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.