COMS2001: Operating Systems
Tutorial 3: Memory & Address Translation

October 20, 2016

Contents

1 Questions

1.1 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);
}
```

1.2 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);
}
```

Answer:

This is ordered from higher to lower memory addresses.

3
str
return address
saved EBP
s
e
Y

1.3 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 = ________________;
    }
}
```

Answer:

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

    for (i = argc-1; i >= 0; i--) {
        // populate stack
        *if->esp = argv[i];

        // update stack pointer
        if_->esp = ( (int*) if_->esp - 1 );
    }
}
```

1.4 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.

1.5 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.

   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.

1.5.1 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