# Section 7: Calling Conventions and Address Translation

October 11, 2015

## Contents


**1 Warmup** **2**
1.1 Hello World/Calling Conventions . . . . . 2

**2 Vocabulary** **3**

**3 Problems** **4**
3.1 Argument Passing . . . . . 4
3.2 Struct Passing . . . . . 4
3.3 Page Allocation . . . . . 6
3.4 Simple Malloc . . . . . 8

# 1 Warmup

## 1.1 Hello World/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 = "Hello World!";
  helper(str, 13);
}
```

```
13
str
return address
saved EBP
!
d
l
...
l
e
H
```

# 2 Vocabulary

- **Calling Conventions** - Describe the interface of called code. Define the order in which atomic parameters are allocated, how parameters are passed, which registers the callee must preserve for the caller, how the stack is prepared and restored, etc.
- **%esp**, **%ebp** - In a stack frame (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. The %ebp register stores the position of the beginning of the current stack frame.
- **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 requires translation into a physical address to actually access the system's memory.
- **Physical Memory** - Physical Memory is the actual memory available to the system in hardware. Physical addresses allow the operating system to directly access memory in hardware.
- **Page** - In paged virtual memory systems, a page is a contiguous fixed size chunk of memory which is the smallest amount of memory that can be allocated to a process by the operating system.
- **Page Table and Memory Management Unit** - The memory management unit (MMU) is responsible for translating a process' virtual addresses into the corresponding physical address for accessing physical memory. A page table stores the actual mapping between a given process' virtual addresses and the system's physical addresses, and resides inside the MMU.
- **sbrk** - `sbrk` is a low level (below `malloc`) C function which can be used to increase (or decrease) the amount of memory allocated to a program. `sbrk(int incr)` will increase (or decrease) the amount of memory allocated to the program by `incr` bytes, and in the case of an increase, return a pointer to the beginning of the newly allocated data.

# 3 Problems

## 3.1 Argument Passing

Fill out the following code to copy integer arguments onto the stack. You can change the ESP by modifying `if_.esp`.

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

    for (_______; ________; _______) {

        ___________ = argv[i];

        if_->esp    = ________________;
    }
}
```

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

    for (i = argc-1; i >= 0; i--) {

        *if_->esp = argv[i];

        if_->esp    = ((int*) if_->esp - 1);
    }
}
```

## 3.2 Struct Passing

What does the following C code output?

```
typedef struct elem {
  int val;
} elem_t;

typedef struct container {
  elem_t* elem;
} container_t;

container_t* container_set_val4(container_t* container, int val) {
  elem_t* elem = malloc(sizeof(elem_t));
  elem->val = val;
  container->elem = elem;
  return container;
```

```
}

container_t* container_set_val3(container_t* container, int val) {
  elem_t elem = {val};
  container->elem = &elem;
  return container;
}

container_t container_set_val2(container_t container, int val) {
  elem_t* elem = malloc(sizeof(elem_t));
  elem->val = val;
  container.elem = elem;
  return container;
}

container_t container_set_val1(container_t container, int val) {
  elem_t elem = {val};
  container.elem = &elem;
  return container;
}

int main(int argc, char *argv[]) {
  container_t containers[5];
  memset(&containers, 0, sizeof(container_t) * 5);

  container_t container1 = container_set_val1(containers[1], 100);
  container_t container2 = container_set_val2(containers[2], 200);
  container_t* container3 = container_set_val3(&containers[3], 300);
  container_t* container4 = container_set_val4(&containers[4], 400);

  printf("%d\n", container1.elem->val);
  printf("%d\n", container2.elem->val);
  printf("%d\n", container3->elem->val);
  printf("%d\n", container4->elem->val);

  printf("%d\n", containers[4].elem->val);
  printf("%d\n", containers[3].elem->val);
  printf("%d\n", containers[2].elem->val);
  printf("%d\n", containers[1].elem->val);
}
```

```
Garbage #1
200
Garbage #2
400
400
Garbage #2
Segfault
```

Is there a way to guarantee the value of `container1.elem->val` when it is printed by modifying one of the other helper functions and calling it in the appropriate place? Assume there aren't any other threads or processes writing to memory.

> Declare a large array in another function and memset everything in it to some value, and call that function. The pointer points somewhere in the stack, so by writing a lot an array of data onto the stack, the pointer should now point somewhere in that array.

## 3.3 Page Allocation

Suppose that you have a system with 8-bit virtual memory addresses, 8 pages of virtual memory, and 4 pages of physical memory.

How large is each page? Assume memory is byte addressed.

> 32 bytes

Suppose that a program 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 |

What will the page table look like if the program runs the following function? Page out the least recently used page of memory if a page needs to be allocated when physical memory is full. Assume that the stack will never exceed one page of memory.

What happens when the system runs out of physical memory? What if the program tries to access an address that isn't in physical memory? Describe what happens in the user program, the operating system, and the hardware in these situations.

```
#define PAGE_SIZE 1024; // replace with actual page size

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

| 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 |

### 3.4 Simple Malloc

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

```
void* malloc(size_t size) {

    ----------------------;
}
```

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