MEMORY SEGMENTATION | SEC204

Overview

- Exchanging data
- Optimising memory access
- Memory segmentation
- The stack

EXCHANGING DATA

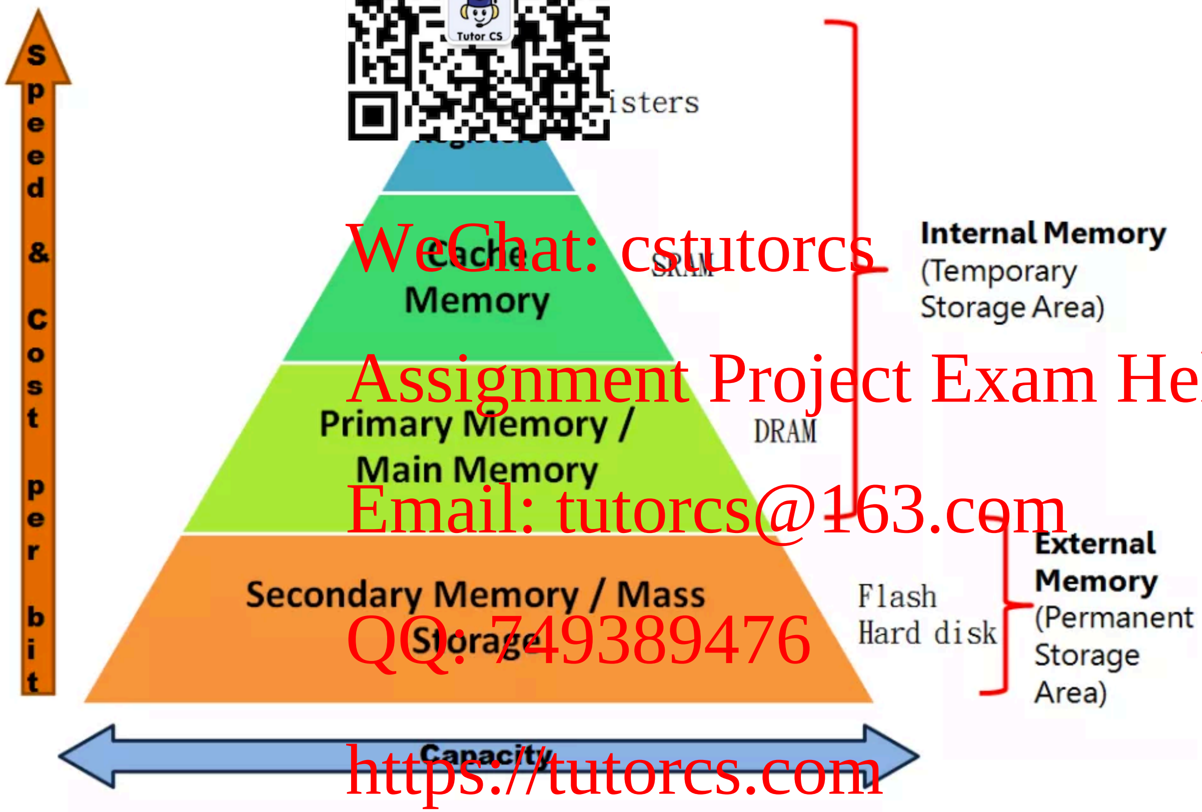
EXCHANGING DATA: NOP

- To swap the values of two registers with the MOV instruction, you need a temporary intermediate register
  - Data exchange functions do that without needing intermediate registers

- XCHG exchanges the values of 2 registers, or a register and a memory location
  - **`xchg %eax, %ebx`**
    exchanges values between %eax and %ebx
  - **`xchg %eax, %eax`**
    This is the NOP operation, which essentially does nothing, other than delay execution or pad bytes

OPTIMISING MEMORY ACCESS

MEMORY HIERARCHY



Source: https://www.vlsifacts.com/classification-of-semiconductor-memories-and-computer-memories/

MEMORY COMPONENTS

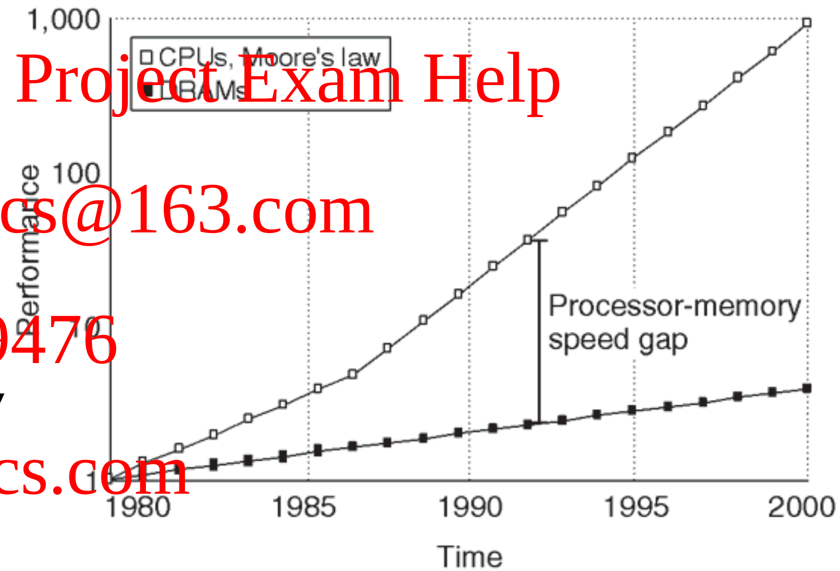
- SRAM (Static Random Access Memory):
  - Value is stored on a pair of inverting gates.

    Very fast, constant access time.

    Needs more space than DRAM (4 to 6 transistors).

    We use it for cache memory

Technology	Speed	$$/Gigabyte
SRAM	0.5-5 ns	$2000-$5000
DRAM	50-70 ns	$20 - $75
Disk	5-20 million ns	$0.20 - $2



Source: https://www.computer.org/csdl/mags/dt/2005/06/d6540.html

- DRAM (Dynamic Random Access Memory):
  - Value is stored as a charge on capacitor.

    Slower than SRAM, variable access time.

    Very dense but slower than SRAM

    We use it for RAM memory

DRAM memory slow, but cheap. SRAM memory faster but expensive

CPU is getting faster more quickly

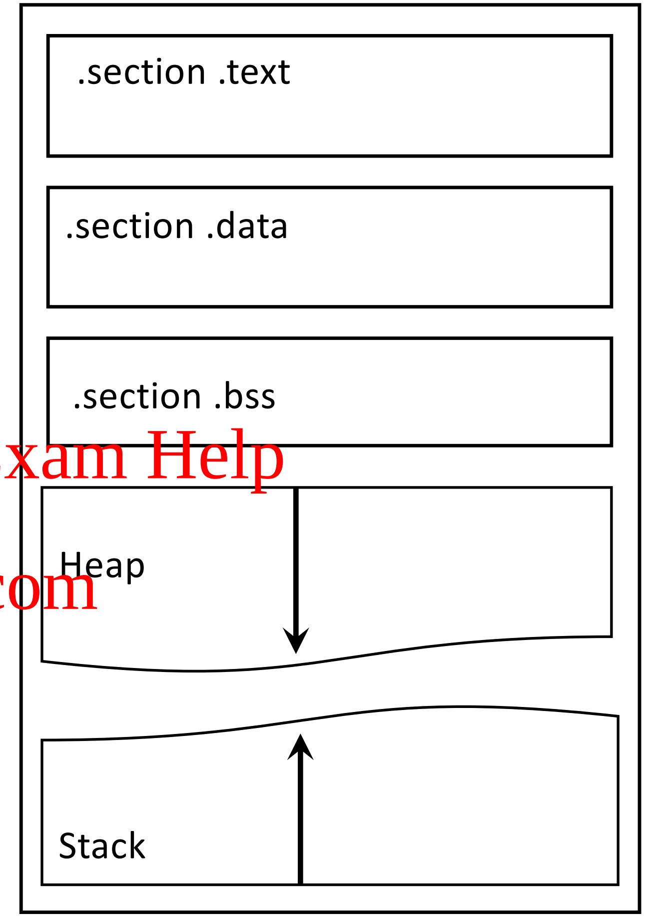
OPTIMISING MEMORY ACCESS

- Memory bottleneck
  - When access to memory [illegible] the computer
  - To avoid this, it is preferable to use registers as much as possible and avoid memory access.
  - Most processors with cache will access sequential blocks of memory and copy into cache at a time.
  - For more efficiency, IA32 suggests data alignment (data memory addresses are multiple of their data size)
    - Align 16-bit data on a 16-byte boundary
    - Align 32-bit data so that its base address is a multiple of four
    - Align 64-bit data so that its base address is a multiple of eight
    - Avoid small data transfers. Instead use a single large data transfer
    - Avoid using larger data sizes (ie 80 and 128-bit floating point values) in the stack
  - Good practice for programmers
    - define and place similarly-sized data elements together at beginning of data section
    - Define strings/buffers and other odd-sized data elements towards the end of the data section

MEMORY SEGMENTATION

PROGRAM'S MEMORY SEGMENTS

- A compiled progra[illegible]y is divided into 5 segments
  - Text (code)
  - Data (initialized static and global variables)
  - Bss (uninitialized variables)
  - Heap
    - Volatile, dynamically allocated memory for program needs (via malloc() and free() in C)
    - Grows towards higher memory addresses
  - Stack
    - Volatile, dynamic, FILO structure
    - Grows towards lower memory addresses



Memory segments example

Download memory_segme[illegible]om the DLE. Compile it and run it:

```
#include <stdio.h>
int global_var;
int global_initialized_var = 5;
void function() {  // This is j[illegible]nction
   int stack_var; // notice this variable has the same name as the one in main()
   printf("the function's stack_var is at address 0x%08x\n", &stack_var);
}
int main() {
   int stack_var; // same name as the variable in function()
   static int static_initialized_var = 5;
   static int static_var;
   int *heap_var_ptr;
   heap_var_ptr = (int *) malloc(4);
   // These variables are in the data segment
   printf("global_initialized_var is at address 0x%08x\n", &global_initialized_var);
   printf("static_initialized_var is at address 0x%08x\n\n", &static_initialized_var);
   // These variables are in the bss segment
   printf("static_var is at address 0x%08x\n", &static_var);
   printf("global_var is at address 0x%08x\n\n", &global_var);
   // This variable is in the heap segment
   printf("heap_var is at address 0x%08x\n\n", heap_var_ptr);
   // These variables are in the stack segment
   printf("stack_var is at address 0x%08x\n", &stack_var);
   function();
}
```

THE STACK

THE STACK

- The stack is a special reserved ory for placing data.
- A stack is a Last-In-First-Out (LIFO/FILO) data structure
  - Data elements are “pushed” on to the top of the stack in sequential manner
  - Data are “popped” off the top of the stack in reverse order
  - You cannot remove data from the middle of the stack
- The stack grows toward lower memory addresses
  - Adding something to the stack means the top of the stack is now at a lower memory address
  - ESP points towards the top of the stack

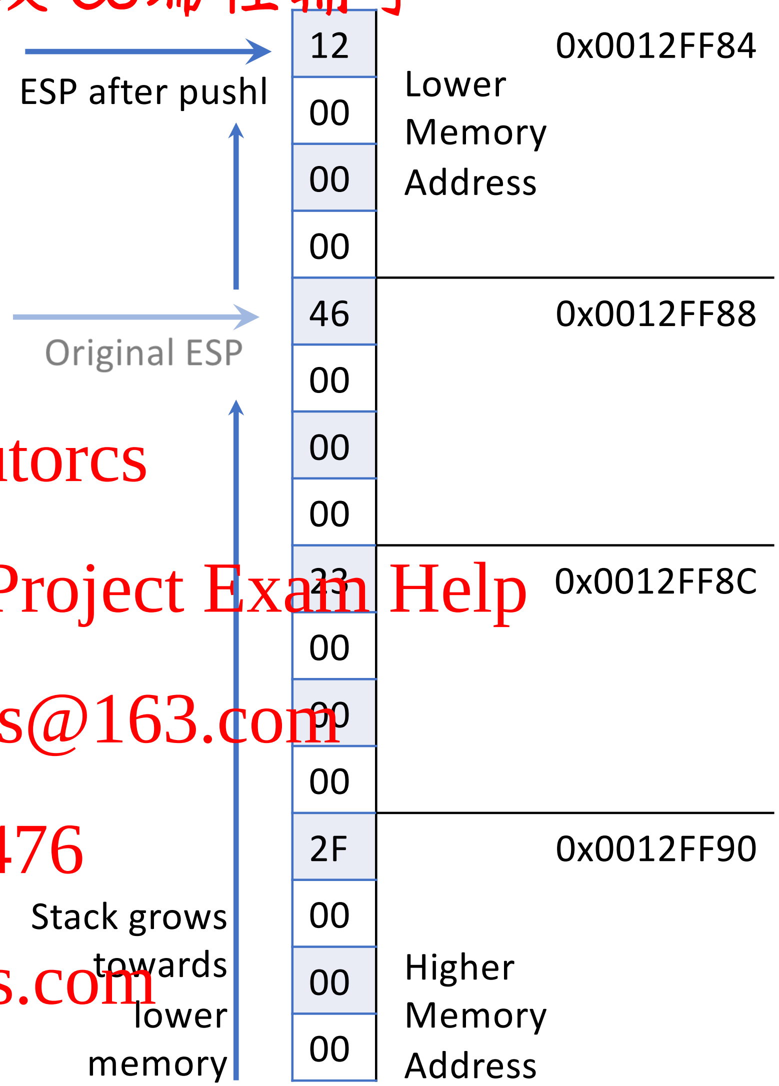
HOW THE STACK WORKS

- The stack is reserved at the end of the memory area
  - ESP points towards the top of the stack
  - EBP points towards the bottom of the working stack
- It grows towards lower memory addresses
  - To add elements to the stack (push), ESP will point to lower memory addresses

  For example:

```
pushl %ecx
```

- It shrinks towards higher memory addresses
  - To remove elements from the stack (pop), ESP will point to higher addresses



WHY WE USE I

- To keep track of [illegible]ctions were called before the current one
- To pass arguments between functions/subroutines
- When running a program:
  - The bottom of the stack contains data elements placed by the O/S when the program is run
  - Any command-line parameters when running the program are also entered onto the stack
  - Then we place our program data

PUSH AND POP

1. Adding data elements to the stack

```
push source
For example (l for long word
32-bits, w for word 16-bits):
pushl %ecx
pushw %cx
pushl $100
```

2. Removing data elements from the stack

```
pop destination
For example (l for long word 32-
bits, w for word 16-bits):
popl %ecx
popw %cx
popl value
```

3. To do these manually:
   - You can manually place data on the stack using ESP as a pointer, then update ESP to point towards the top of the stack.
   - You can manually remove data from the stack by updating the ESP to point towards the previous data element.
   - Will ESP increase value or decrease when removing data?

PUSH, POP EXAMPLE

Remember example mo[illegible] from last week? Where is the pop?

```
.section .data
output:
   .asciz "The value is %d\n"
values:
   .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60
.section .text
.globl _start:
 _start:
   movl $0, %edi
loop:
   movl values(,%edi,4),%eax
   pushl %eax
   pushl $output
   call printf
   addl $8, %esp
   inc %edi
   cmpl $11, %edi
```

…cont…

```
   jne loop
   movl $0, %ebx
   movl $1, %eax
   int $0x80
```

Stack example

1. Create a stack_exam[illegible]with the following contents:

```
void test_function(int a[illegible]nt c, int d) {
   int flag;
   char buffer[10];

   flag = 31337;
   buffer[0] = 'A';
}

int main() {
   test_function(1, 2, 3, 4);
}
```

3. In (gdb):

```
disass main
disass test_function
list main
break 10
break test_function
run
i r esp ebp eip
cont
i r esp ebp eip
cont
```

2. Compile it and run it in gdb to watch how esp, ebp, and eip change

```
   $ gcc -g stack_example.c
   $ gdb -q ./a.out
```

Heap example

1. Download heap_exa… from DLE. Extract below:

```
int main(int argc, char *argv[]
   char *char_ptr;   // a char p
   int *int_ptr;     // an integer pointer
   int mem_size;

   if (argc < 2)      // if there aren't commandline arguments,
      mem_size = 50; // use 50 as the default value..
   else
      mem_size = atoi(argv[1]);

   printf("\t[+] allocating %d bytes of memory on the heap for char_ptr\n", mem_size);
   char_ptr = (char *) malloc(mem_size); // allocating heap memory
```


2. Compile it and run it to watch how heap memory is allocated and freed


```
$ gcc -o heap_example heap_example.c
$ ./heap_example
$ ./heap_example 100
```

FURTHER READING

- Professional Assembl[illegible]e, chapter 5, pg 106-124
- Hacking: The art of exploitation, section 0x270, pg 69-81