# cse30 discussion 7

Ibrahim Awwal

July 20, 2015

## arm assembly

## conditional execution

- `CMP` instruction: compares register(s) and/or immediates
- equivalent to `SUBS` without storing result of subtraction
- Why is this?
- After performing comparison, we can conditionally execute any instruction

## conditional execution - examples

## control flow - if

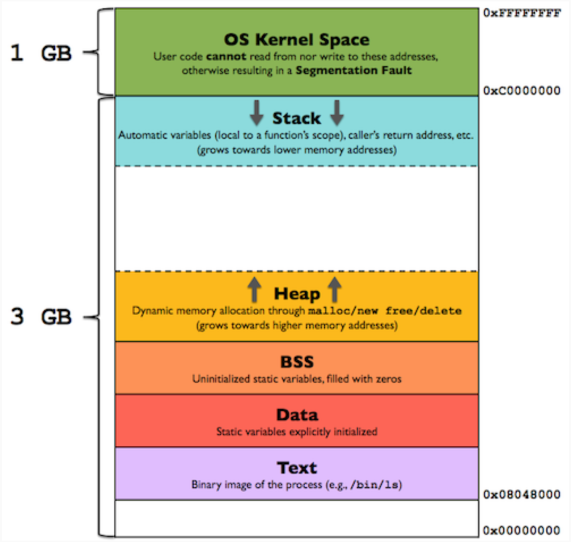
## control flow - for

## function calls

## the stack

- In general, a stack is a Last In, First Out data structure (LIFO)
- Basic operations: *push* data onto stack, *pop* data off of stack
- In terms of computer organization, *the stack* is a region of memory that operates in this manner
- Stores automatic variables, return address, any registers we need to save before calling a function

## memory layout


0xFFFFFFFF
1 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
3 GB
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

## stack nomenclature

- **Ascending** stack grows upwards, i.e. memory addresses go from low to high
- **Descending** stack grows downwards, i.e. memory addresses go from high to low
- **Empty** stack, the stack pointer points to the next free (empty) location on the stack
- **Full** stack, the stack pointer points to the topmost item in the stack

## stack nomenclature

- **Ascending** stack grows upwards, i.e. memory addresses go from low to high
- **Descending** stack grows downwards, i.e. memory addresses go from high to low
- **Empty** stack, the stack pointer points to the next free (empty) location on the stack
- **Full** stack, the stack pointer points to the topmost item in the stack

- The ARM Linux stack convention is to use a **full descending** stack
- That is, addresses grow downwards, and `$sp` points to the last item pushed onto the stack

## push and pop instructions

- Push registers onto, and pop registers off a full descending stack.
- PUSH{cond} reglist
- POP{cond} reglist
- reglist is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.
- PUSH and POP are synonyms for STMDB and LDM (or LDMIA), with the base register sp (r13), and the adjusted address written back to the base register
- source

## system calls: leveraging the os

- You can think of it as calling functions which are part of the OS
- Has a different calling convention from normal functions
- Each system call has a number associated with it
- Store parameters in `r0`-`r6`, system call number in `r7`
- Call syscall using `SVC` instruction
- Examples: `write` writes to a file descriptor, `sbrk` is for allocating more heap space

## system calls: references

- Syscall numbers: `/usr/include/arm-linux-gnueabihf/asm/unistd.h`
- Linux Syscalls (incl. arguments)
- Manpages are accessible under section 2 (eg. `man 2 write`)
- More info

## exercises

## int to hex string

```
char *itohex(int x);
```

- Returns hex representation of integer x as a string
- eg. `itohex(256)` −> `0x00000100`

## linked lists

```
typedef struct node{
    int val;
    struct node *next;
} Node;

Node *newNode(int val);
Node *insertNext(Node *n, int val);
Node *append(Node *n, int val);
void printList(Node *start);
int removeVal(Node *n, int val);
```

## binary tree

```c
typedef struct tree_node {
    int val;
    struct tree_node *left;
    struct tree_node *right;
} TreeNode;

TreeNode *newTreeNode(int val);
TreeNode *insert(TreeNode *n, int val);
void printInOrder(TreeNode *n);
int removeVal(TreeNode *n, int val);
```