cse30 discussion 7

Ibrahim Awwal

July 20, 2015

arm assembly

function calls

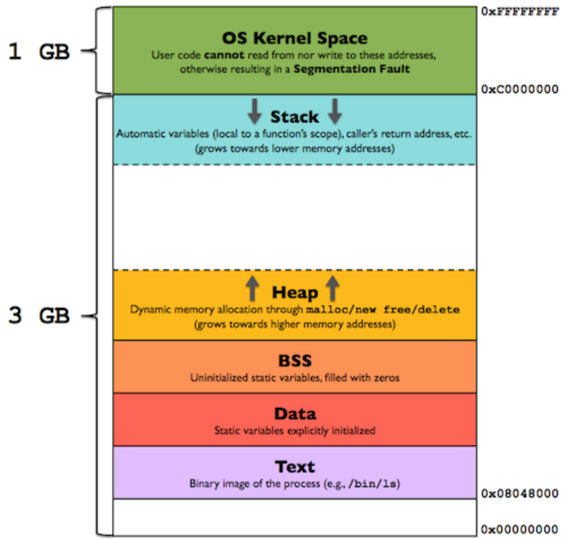
Recap: function calling procedure

1. Save temporary registers onto the stack
2. Put arguments in r0-r3
3. Branch and link to target address (`bl <funcname>`)
4. Callee puts return value in r0-r1
5. Callee branches back to link register (`bx lr`)

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



OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**

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)

BSS
Uninitialized static variables, filled with zeros

Data
Static variables explicitly initialized

Text
Binary image of the process (e.g., `/bin/ls`)

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

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

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

int to hex in c

```c
char *itohex(int x){
    int nibbles = sizeof(int)*2;
    char *out = malloc(nibbles+3);
    out[0] = '0'; out[1] = 'x';
    for(int j=0; j<nibbles; j++){
        int mask = 0xF << 4*j;
        int digit = (mask&x) >> 4*j;
        if (digit < 10)
            out[nibbles-j+1] = '0' + digit;
        else
            out[nibbles-j+1] = 'A'+digit-10;
    }
    out[nibbles+2] = '\0';
    return out;
```

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);
```

linked list - create

```
Node *newNode(int val){
    Node *n = malloc(sizeof(Node));
    n->val = val;
    n->next = NULL;
    return n;
}
```

linked list - insert

```c
Node *insertNext(Node *n, int val){
    Node *next = newNode(val);
    if(n->next != NULL){
        next->next = n->next;
    }
    n->next = next;
    return next;
}
```

linked list - append

```
Node *append(Node *n, int val){
    if(n->next == NULL){
        return insertNext(n, val);
    }else{
        return append(n->next, val);
    }
}
```

linked list - print

```c
void printList(Node *start){
    printf("%d ", start->val);
    if(start->next != NULL){
        printList(start->next);
    }
}
```

linked list - remove

```
int removeVal(Node *n, int val){
    if(n->next == NULL){
        return -1;
    }else{
        if(n->next->val == val){
            n->next = n->next->next;
            return 0;
        }else{
            return removeVal(n->next, val);
        }
    }
}
```

binary tree

```
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);
```

binary tree - new

```c
TreeNode *newTreeNode(int val){
    TreeNode *n = malloc(sizeof(TreeNode));
    n->val = val;
    n->left = NULL;
    n->right = NULL;
    return n;
}
```

binary tree - insert

```
TreeNode *insert(TreeNode *n, int val){
    if(val > n->val){
        if(n->right == NULL){
            n->right = newTreeNode(val);
            return n->right;
        }else{
            return insert(n->right, val);
        }
    }else{
        if(n->left == NULL){
            n->left = newTreeNode(val);
            return n->left;
        }else{
            return insert(n->left, val);
```

binary tree - in order print

```
void printInOrder(TreeNode *n){
    if(n->left){
        printInOrder(n->left);
    }
    printf("%d ", n->val);
    if (n->right){
        printInOrder(n->right);
    }
}
```