Stacks

Textbook Section 6.2

Stacks

- A **Stack** is a LIFO (“Last In, First Out) collection of elements
  - Example: Can of tennis balls, plate dispenser at restaurants
- A stack as a top
- A stack has two fundamental operations:
  - Push
  - Pop

Stacks

How are stacks implemented in a high level language?

Two common choices:

- array → top is indicated by an index
- linked list → top is the head node

For now, let us consider an array implementation: There are two options:

- top starts low and grows up
- top starts high and “grows” down towards zero

Stacks - Considerations

What does top point at?

- Current top element
- First available spot

The most common method is to point top at the current top element

Stacks - Option 1 Stack Grows Up

A stack where **top** starts at -1 and grows up (bottom up stack):

- top points to last pushed element
- empty stack top is -1, so first push will make top 0
- push → updating top and then storing the value

```
void push(intStack stack, int value) {
    stack.top++;
    stack.stack[top] = value;
}
```

- pop → access (optional) and then reduce top

Stacks - Option 2 Stack Grows Down

```
#define STACK_SIZE 100;
int stack[STACK_SIZE]; // a stack of integers
int top = STACK_SIZE;
...
stack[--top] = 1; // push 1, then 2, then 3
stack[--top] = 2; // NOTICE decrement top & then access
stack[--top] = 3; // 3 is on the top
...
cout << stack[top++] << endl; // pop 3, then 2, then 1
cout << stack[top++] << endl; // NOTICE access and increment
top++; // don't have to access to pop!
```

Stacks - Top Down

- the stack is empty when top is 100 (valid indices are 99-0)
- the stack is full when top is 0
- to push:
  - decrement top and then set stack[top]
- to pop:
  - with retrieval: get stack[top] and then increment top
  - without retrieval: increment top
  - notice the order of the operations in this implementation!

Stacks in Assembly Language

- A stack can be implemented in asm as an array.
- For a downward growing stack:
  - Maintain a pointer to the top
  - Pre-decrement the pointer when pushing
  - Post-increment the pointer when popping

Stacks in Assembly Language

```
; a0 = stack pointer
STACK_SIZE equ 100
        lea top,a0
        move.w  #1,-(a0)    ; push 1, then 2, then 3
        move.w  #2,-(a0)
        move.w  #3,-(a0)    ; 3 is on the top
        move.w  (a0)+,d0    ; pop 3 (and save)
        adda.l  #4,a0       ; pop 2 and 1 (don't save)
stack:  ds.wSTACK_SIZE
top:
```

System Stacks

- The 68000 & TOS provide two system stacks:
  - the user stack top pointed to by A7 (SP)
  - the supervisor stack top pointed to by A7’ (SSP)
- In this course we consider only the user stack.
- The system stack is TOOL for the programmer to use.

System Stack - User Stack

- a user program may use this stack (very likely!)
- the user stack memory range is in “high memory”
  - Placed in high memory so:
    - it doesn’t need to be shifted to accommodate user programs
    - can grow as large as required
- the user stack grows downward
- there is no overflow/underflow checking
- the system stacks are always word-aligned
  - pushing a byte actually pushes two bytes

Stacks - Sample Assembly Programs

- Stack_1.s
  - Difference when processing bytes on a user-defined stack and the system stack
- Stack_2.s
  - Viewing how different sizes get stored on the system stack
  - The importance of correctly accessing items on the stack
- Stack_3.s
  - No need to store items being removed from the stack IF not needed

System Stack

```
; a7 = stack pointer
    move.l      #1,-(sp)        ; push 1, then 2, then 3
    move.w      #2,-(sp)
    move.b      #3,-(sp)        ; 3 is on the top
    move.b      (sp)+,d0        ; pop 3 (and save)
    addq.l      #6,sp           ; pop 2 and 1 (don't save)
```

You can push any sized item you want on the stack – but the system maintains the stack word-aligned.

YOU are responsible for accessing items in the correct size!