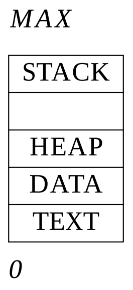
Lecture 11

On the structure of programs in memory, and how to preserve variables throughout function calls by using the stack

Programs in memory

- When a program runs, it is allocated some space in RAM



Text - the code itself

Data - data that persists as the programs runs (e.g. global / static variables)

Heap - also known as ’free storage’. Heap size is dynamic; it constantly grows / shrinks (new / delete), and as such its size isn’t known at the start.

Stack - Contains all local variables, temporary information, parameters, return addresses, etc. The stack is usually small, has a max size that the OS will allow it to be.

- The stack grows downwards -- the larger it is, the closer it gets to the heap
- The heap grows upwards, towards the stack

- Theoretically, if the stack gets large enough, there may be a stack underflow (the stack and heap collide). This may happen if you allocate too much space in a static array, or have recursion that’s too deep.
  - In practice, this won’t happen, because the MAX for a stack is very large.
  - *Note: the MAX is not related to how much RAM you have -- it’s typically huge (terabytes/petabytes)*

All programs sit somewhere in the RAM, each with the previously mentioned structure.

RAM

MAX

PROGRAM 1
PROGRAM 2
PROGRAM 3

0

However, each program believes that it starts at address 0 and ends at its MAX. Then, the operating system breaks it into chunks and scatters it across RAM. As a consequence, we write each program as if it's the only program using the RAM.

More on the stack

What happens if we need to use more registers than we have available in our program? (for example, you may be storing many values that need to be preserved).

Solution: We can use the stack to preserve values while we use registers for other purposes, then restore the registers and values back to their original state.

Stack pointer - points to the lowest edge of the stack

- Stored in $sp
- To allocate memory for the stack, move the stack pointer down (subtract bytes, e.g. 4)

EXAMPLE 1

# preserve previous state of S registers using the stack

```
FUNC:
    ADDI $sp, $sp, -4      #allocate 4 bytes more
    SW   $s0, 0($sp)       #store $s0 in $sp
    ...
    ADDI $s0, $zero, 1     #mess up $s0
    ...
    LW   $s0, 0($sp)       #load original $s0 value back, restore state
    ADDI $sp, $sp, 4       #restore stack pointer to how it was before (deallocate)
    JR   $ra               #return control to caller
```

<u>EXAMPLE 2</u>

```
# preserve previous state of S registers using the stack (2 registers)
FUNC:
    ADDI $sp, $sp, -8      #allocate for 2 registers
    SW   $s0, 4($sp)       #save $s0, 4 bytes up cause the top of new stack portion is 8 bytes up
    SW   $s1, 0($sp)       #goes on bottom of new stack portion
    ...
    ADDI $s0, $zero, 1     #mess up $s0
    ...
    LW  $s0, 4($sp)        #restore $s0 state
    LW  $s1, 0($sp)        #restore $s1 state
    ADDI $sp, $sp, 8       #restore stack pointer to how it was before (deallocate)
    JR  $ra                #return control to caller
```