CS 61 Scribe Notes - 10/21/14
0:00

Cumulative Distribution Function when talking about grade

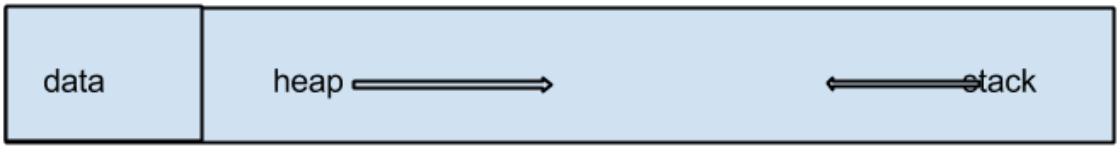
- You may be used to seeing bell curves, but those require fancy math to produce stable results. For example, you change the binning on your histogram, the boundary condition can drastically change the distribution's appearance in important ways
- Graph of CDF:
    - (1) Everybody got 50%: step function that steps from 0 to 100 at 50%.
    - (2) randomly assigned: approximately the line y = x
    - (3) bell curve: an s-shaped curve
- y-axis gives the fraction of scores <= a certain score x
- The current grading breakdown is only tentative; focus on learning, not the grade; be proactive about problem sets and we will notice

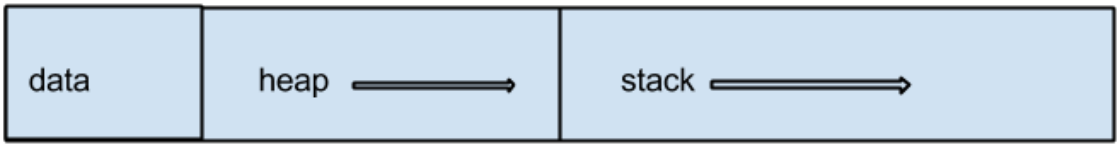
Calling Convention - starting with f51.c

- The way people who write compilers have agreed to use stack, registers, and instructions to enable functional composition
    - Functional composition: when a program can call a function, when a function can be written by a different person, at a different time, etc.
- Every architecture, such as x86, has standards that tell us how we can use registers how we can pass arguments between functions, etc. Environment include values of registers and contents in memory
    - Example registers: %eax, %ebx, %ecx, %edx, %esi, %edi %eip, %esp, %ebp
- One hidden register for comparisons (conditional branch instructions) that we've already seen
    - used when a comparison sets flags and when some instruction acts on those flags
- %esp is special; it has special instructions for push/pop
- %eax means nothing on input
- *Question: Could you use %eax to store stack pointer? Yes, technically.*
- Which registers mean something important at beginning (input)?
    - %esp contains the return address, placed on by call instruction
    - %eax is garbage at input; holds return value at return
- Callee-saved registers
    - registers that have to be restored when function returns (caller assumes callee will leave certain registers intact); e.g. %ebx, %ebp, %esi, %edi, %eip, %esp
- Caller-saved registers
    - registers that you can modify without restoring -> meaning nothing at input; e.g. %eax, %ecx, %edx

Stack

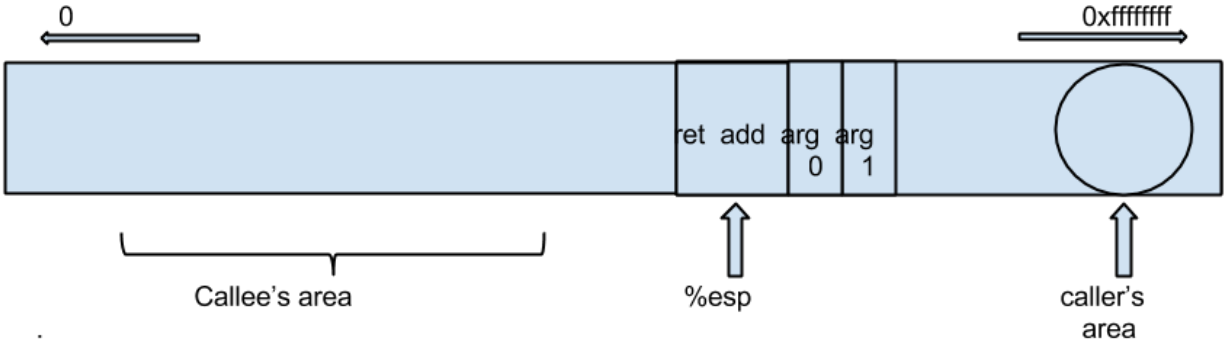
- The name of “stack_top” is metaphorical; the stack is a container where you can access one element more easily than the others (the top). The reason its at bottom is b/c there is a heap.
- The heap grows up but the stack grows down. Why? To allow flexibility for small stack/large heap, etc.
- The first arrangement below is how the stack and heap work; the second arrangement below would cause internal fragmentation.





Has internal fragmentation

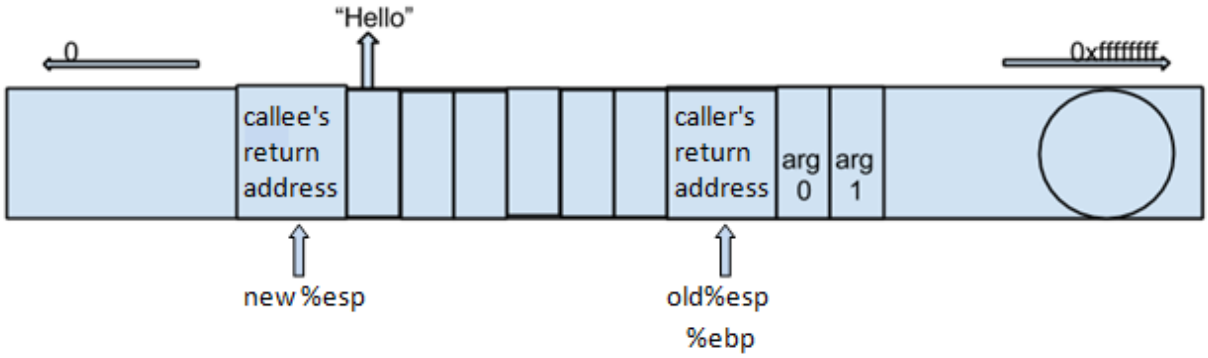
- Picture of the stack:



- The currently running function (callee) owns the top of the stack (lowest memory addresses)
- Arguments are a minimum of 4 bytes each

Return address is stored in %esp

- Suppose we call a function such as printf("Hello")
- Then %ebp saves the caller's stack pointer and %esp moves to a 16-byte aligned place and stores the callee's return address:



Examine Assembly Code

- probably doesn't have arguments, not initializing the space
- *Answer to question: It's easier, not necessarily faster to access top of stack*
    - *by this, we are not saying that access is faster/slower b/c of presence or lack of random access, but because the computations needed are easier*
- Compiler is ensuring stack pointer is aligned to a multiple of 16 (calling convention!)
- shifted over by an additional 12, after we allocated 4 bytes for return address
- by induction, we assumed caller stack pointer was a multiple of 16, so we shift back by 16 for the callee stack pointer - the point is that we want to make sure the next is a multiple of 16
- f53.s: shift 28 back (16 more than 12)
- call x means: pushl[next eip], jump x
- upon return, we popl %eip which really means:
    - movl (%esp), %eip
    - addl $4, %esp

5-min BREAK

Continuing with Assembly

- Offsets
    - Negative offsets off of %ebp are my (i.e. caller's) locals
    - Positives offsets off of %ebp are my (i.e. caller's) arguments
    - Positive offsets off of %esp are next called function's (i.e. callee's) arguments
- pushl %ebp (for complicated code) to store old value of %ebp
- leave pops %ebp
- *Question: Why do we need both %esp and %ebp? Why can't compiler keep track of offset for us?* Most of the time calculation is fine, but dynamically allocated arrays on the stack cause the stack frame to have a variable size
- Two midpoint functions, second one prevents overflow
- There is a Java bug that used first midpoint function for Quicksort

- Unsigned division treated as signed would be wrong, integers rather than unsigned requires you to shift right by 31 to only preserve sign bit
- f64.s: add dereferenced pointers to ints, “add” is something you do to ints
- array of ints
- array of structs of ints
- Test is different from comp, test “logical-and”s the two inputs, while comp subtracts the first input from the second input, and then each set the flags based on the result
- f73.s: loop, initially test whether (n==0) to see if we need to enter loop at all
- f74.s: find index of array element that equals input value
- f88.c: for unsigned, overflow defined in terms of two’s complement
- f87.s: for int, overflow is undefined, so loop forever

Peter Ku
Kyle Kwong
Ankit Gupta
Dawit Gebregziabher