CSO-Recitation 07

CSCI-UA 0201-007

R07: Assessment 05 & Assembly & lab2

Today’s Topics

- Assessment 05
- Assembly
- More about lab2
  - More debugging
  - Some valuable questions asked
- Some exercises

Assembly

C is for people

Why Assembly

- In the real world, computers don’t “understand” code
- They only “understand” a set of instructions
- To run code
  - 1. The CPU fetches an instruction from the memory at the PC(program counter)
  - 2. The CPU decodes that instruction
  - 3. If needed, the CPU fetches data from memory
  - 4. The CPU performs computations
  - 5. If needed, the CPU writes data to memory
  - 6. The CPU increments the PC to the next instruction

Why Assembly

- Computers don’t “understand” assembly either, but assembly maps much more closely to machine instructions than C code
- Assembly code involves instruction “mnemonics”
  - For x86_64, These are things like addq, movq, imul

X86 general purpose registers

- Accessing memory is very, very slow compared to the rest of what a CPU can do
- Registers are fast temporary storage
- X86-64 ISA: 16 8-byte general purpose registers
- Originally there were 8, all 16-bits large
  - %ax, %bx, %cx, %dx, %si, %di, %bp, %sp
  - These have 32-bit counterparts – add an e, eg %eax, %esp
  - These also have 64-bit counterparts – add an r, eg %rax, %rsp
- With 64 bits came 8 more registers, %r8 to %r15
  - These have 32-bit counterparts - add a d, eg %r8d
  - These have 16-bit counterparts – add a w, eg %r8w
- All registers also allow you to access their lowest 8 bits
- %ax, %bx, %cx, and %dx, allow you to access their upper 8 bits

Important Instructions

Instruction	What it does
mov src, dest	dest = src
add src, dest	dest = dest + src
sub src, dest	dest = dest - src
imul src, dest	dest = dest * src
inc dest	dest = dest + 1

More about lab2

Debugging & Some valuable questions

More on debugging

1. Program received signal SIGSEGV, Segmentation fault.
   - GDB will tell you where your code segfaulted
   - GDB can tell you what values are what
     - why your code segfaulted

Debugging a crash

- *run* your program
- Use *bt* to see the call stack
  - You can also use *where* to see where you were last running
- Use *frame* to go to where your code was last running
- Use *list* to see the code that ran
- Check the locals (*info locals*) and args (*info args*) to see if they are bad

More on debugging

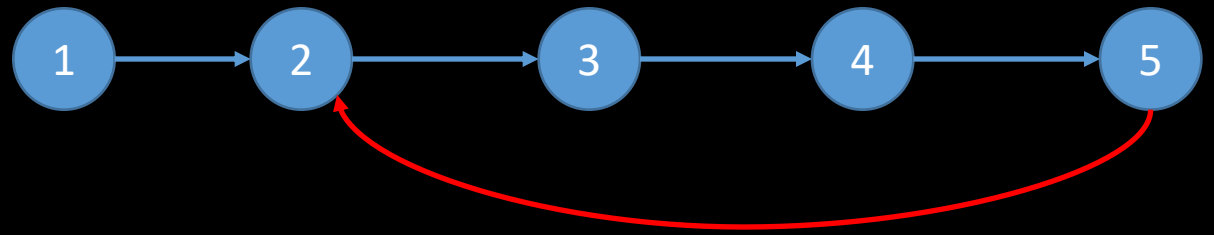
1. Program received signal SIGSEGV, Segmentation fault.
   - GDB will tell you where your code segfaulted
   - GDB can tell you what values are what
     - why your code segfaulted
2. Program get stuck
   - infinite loop

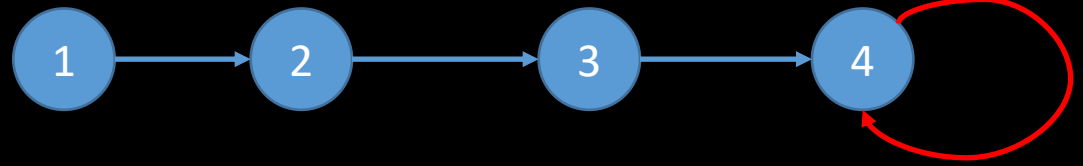
Debugging an infinite loop

- Just *run* it inside gdb and hit control-c (signal)
- *list* the code
  - This is so you can see the loop condition
- *step* over the code
- Check (*print*) the values involved in the loop condition
  - Are they changing the right way? Are the variables changing at all?

Loop in a linked list

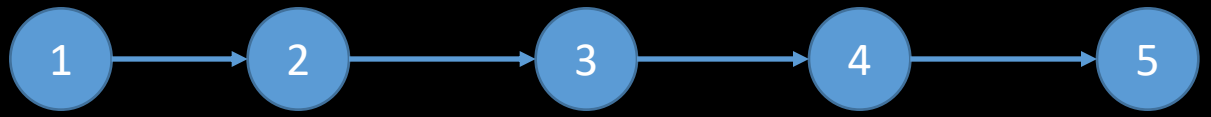
- When I insert a node into the linked list, what will cause a loop?



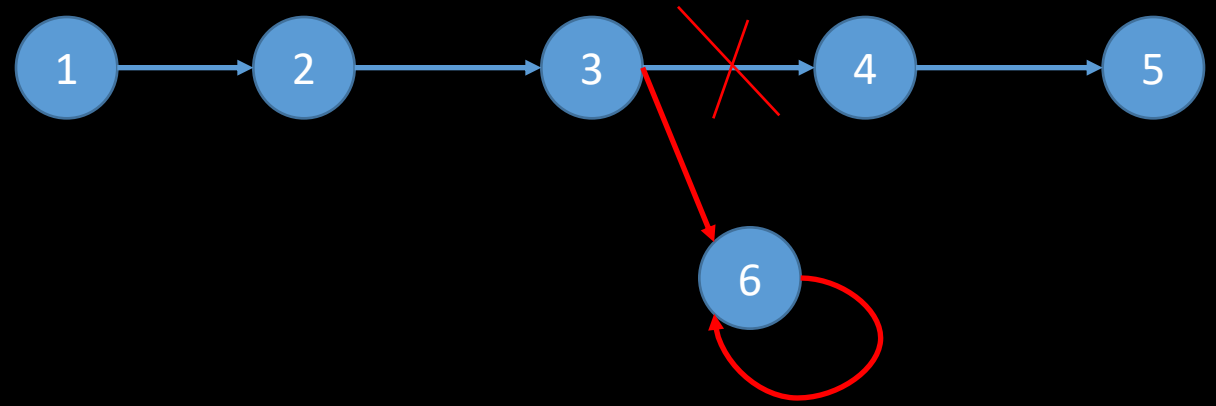


Loop in a linked list

- When I insert a node into the linked list, what will cause a loop?

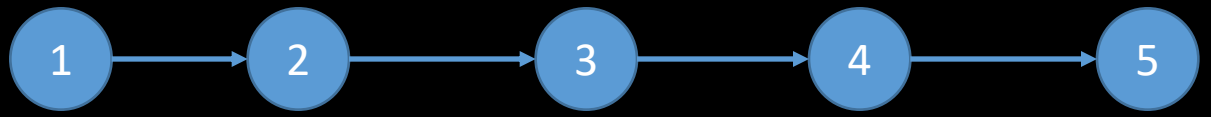


- I need to insert n6 between n3 and n4:
  - suppose our *head* now points to n3
  - head -> next = n6
  - n6 -> tuple.key = keys
  - n6 -> tuple.value= value
  - n6 -> next = head->next

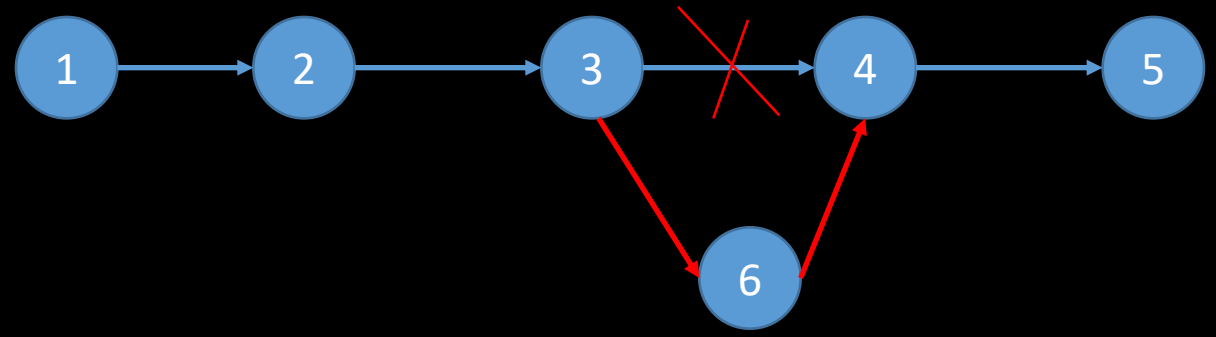


Loop in a linked list

- When I insert a node into the linked list, what will cause a loop?



- I need to insert n6 between n3 and n4:
  - suppose our *head* now points to n3
  - n6 -> next = head->next
  - n6 -> tuple.key = keys
  - n6 -> tuple.value= value
  - head -> next = n6



More on debugging

1. Program received signal SIGSEGV, Segmentation fault.
   - GDB will tell you where your code segfaulted
   - GDB can tell you what values are what
     - why your code segfaulted
2. Program get stuck
   - infinite loop
3. GDB can print structs!
   - p *head will print the fields of the struct pointed to by *head*
4. GDB can interpret numbers however you tell it to!
   - Use the x command to view the data at a memory address
     - x buf means print the value at *buf*
   - x/10b means print 10 (10) bytes (b) – can be used to “x/8b $rdi”
   - use p command with /x to print number in hex notation: p /x val

Function pointer

- In lab2, we invoke the function by function pointer *accum*
- Like normal data pointers (int *, char *, ..), we can have pointers to functions
  - A function pointer points to code, not data. Typically a function pointer stores the start of executable code
  - A function's name can also be used to get functions' address
  - In general, function pointer refer to functions of any signature. return type does not necessarily have to be void
  - Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function
    - why this useful?

Function pointer

- Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function
- We can invoke different functions into one function by using a function pointer
  - as long as the different functions using the same parameters and have the same return types
  - In C, we can use function pointers to avoid code redundancy

Testing

- When you fail in one test case, it does not mean you can only have bugs in this function implementation
  - Even if you have passed the previous test cases..
- No one test can help you test all possible bugs in your code
- Led to an interesting research topic:
  - Proof of Program Correctness