

Christian Suleiman

Take Home Test 2: Recursive Functions

CS342 Section EF

Fall 2019

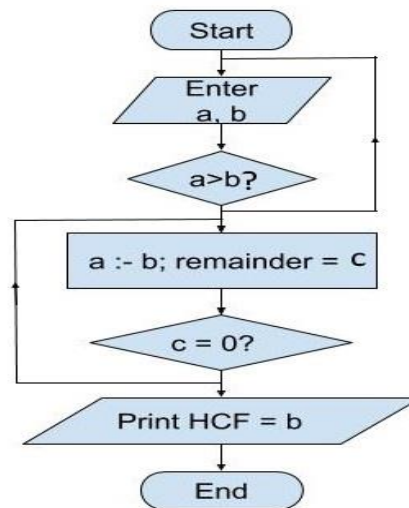
Objective

The objective of this take-home test is to study the process of memory allocation when a processor handles recursive calls. We will look at how the stack is manipulated and accessed during these calls and when it must return to a previous address once the call is complete.

We will be looking at three different environments in which this process takes place. These environments are

- Using the MIPS Instruction Set on MARS simulator
- x86 Intel Processor on Microsoft Visual Studio
- 64-bit Intel Processor on Linux-based GDB

To study recursive functions, we require a recursive function to test on. For this assignment, we will be implementing a recursive algorithm to find the Greatest Common Divisor (GCD) of two positive integers. This algorithm will replicate Euclid's algorithm, the pseudo-code of which can be seen here:



MIPS on MARS

Here is my MIPS Assembly code for implementing Euclid's algorithm.

```
1 | # Christian Suleiman
2 | # GCD Recursive
3 |
4 | li $a0, 44 #load values into reg:
5 | li $a1, 8
6 | li $a2, 0 # hold the result/temp
7 |
8 | sub $sp,$sp,16 # creating space
9 |
10 | sw $ra,0($sp) #storing values in
11 | sw $a0,4($sp)
12 | sw $a1,8($sp)
13 | sw $a2,12($sp)
14 |
```

This first set of instructions is to be used for allocating space to hold our local variables(A, B, result) as well as the return address for when we wish to come back to the main() function (four 4-btyes of data → offset of 16 from stack pointer).

```

19 gcd:
20     # a0 and a1 are the two integer parameters
21     sub $sp,$sp,12
22     sw $ra,0($sp)
23     sw $a0,4($sp)
24     sw $a1,8($sp)
25     move $t0, $a0
26     move $t1, $a1
27
28     #loop:
29     beq $t1, $0, done #if second arg(a0) is 0 then done
30     div $t0, $t1 #divide our operands . . .
31     move $t0, $t1 # a = b in euclid's algo
32     mfhi $t1 # b = remainder
33     move $a1,$t1
34     move $a0,$t0
35     jal gcd #return to start of loop
36 done: #essentially a loop of returning
37     lw $ra,0($sp) # loading top of stack
38     addi $sp,$sp,12 # deallocating space
39     jr $ra
40 EXIT:

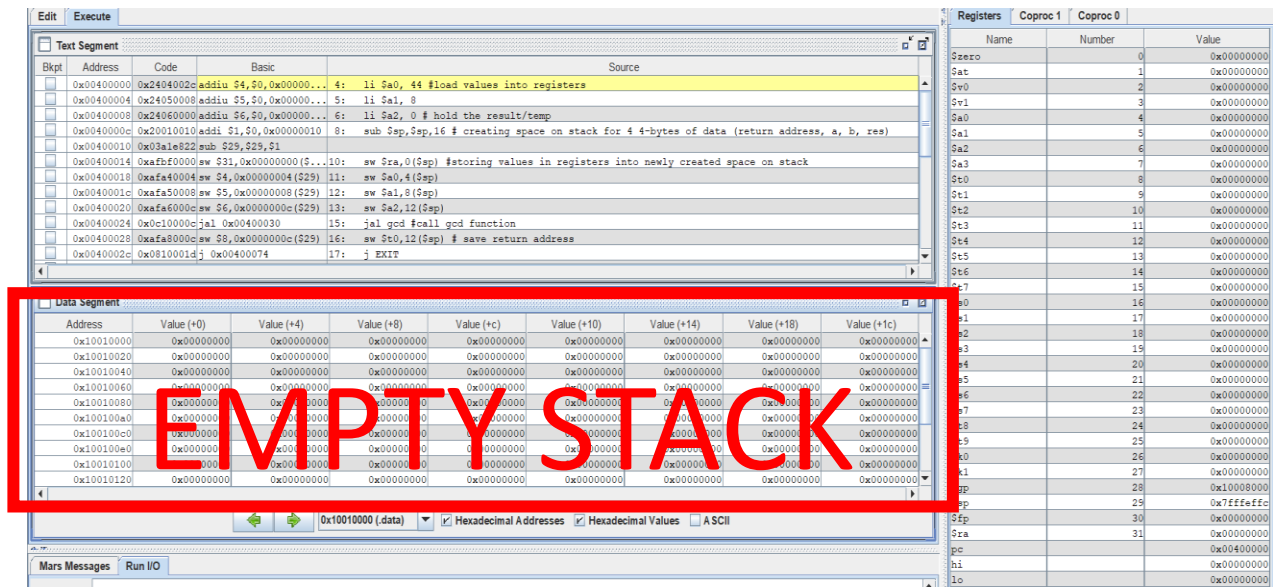
```

In this next set of instructions, we define our GCD function as well as handle memory allocation/deallocation for each recursive call.

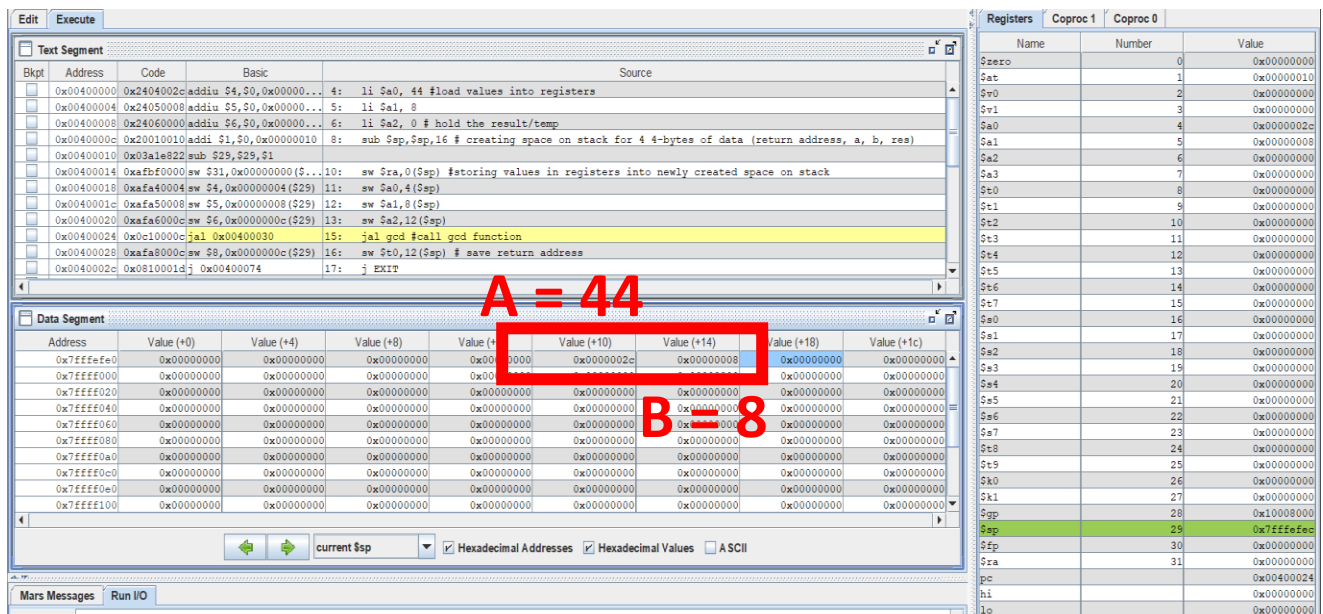
At the beginning of the GCD function, we allocate additional space for the current iteration's set of local variables. So each time we call GCD, we call will create a new stack.

We then begin the loop that exits when $B = 0$, the answer being A . If not then we perform another GCD call with $B, A \% B$ as the arguments. Once we find the solution, we loop through done and deallocate an offset of 12 for each stack frame we generate.

The following figures will show how the stack is modified when performing recursive calls.



This is the stack before we begin executing any instructions, viewed via the MARS debugging Window.



This figure shows the stack allocating space to our local variables A and B on the first call of GCD(44, 8). We also allocate space to hold our result but it will remain empty until we finish the recursive procedures. Also important to note that integers are stored in BIG Endian Format, that is MSB is on the left.

Text Segment

Bkpt	Address	Code	Basic	Source
0x0040003c	0xafa40004	sw \$4,0x00000004(\$29)	23:	sw \$a0,4(\$fp)
0x00400040	0xafa50008	sw \$5,0x00000008(\$29)	24:	sw \$a1,8(\$fp)
0x00400044	0x00044021	addu \$9,\$0,\$4	25:	move \$t0,\$a0
0x00400048	0x00054821	addu \$9,\$0,\$5	26:	move \$t1,\$a1
0x0040004c	0x11200006	beq \$9,\$0,0x00000006	29:	beq \$t1,\$0, done #if second arg(a0 & a1) is 0 we are finished
0x00400050	0x0109001a	div \$9,\$9	30:	div \$t0,\$t1 #divide our operands . . . Quotient gets sent to LO and remainder gets sent to HI
0x00400054	0x00094021	addu \$8,\$0,\$9	31:	move \$t0,\$t1 # a = b in euclid's algorithm
0x00400058	0x00004810	mflhi \$9	32:	mflhi \$t1 # b = remainder
0x0040005c	0x00092821	addu \$5,\$0,\$9	33:	move \$a1,\$t1
0x00400060	0x00082021	addu \$4,\$0,\$8	34:	move \$a0,\$t0
0x00400064	0xc10000c4	jal 0x00400030	35:	jal \$ra # start of loop
0x00400068	0x8fbf0000	lw \$31,0x00000000(\$...	37:	lw \$ra,\$0 # load up the stack pointer before

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7ffffef0	0x00400028	0x00000002c	0x000000008	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7ffffef4	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7ffffef8	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7ffffefc	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7fffff00	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7fffff04	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7fffff08	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7fffff0c	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7fffff10	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000

Registers

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000004
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000004
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$s8	24	0x00000000
\$s9	25	0x00000000
\$t0	26	0x00000000
\$t1	27	0x00000000
\$gp	28	0x10008000
\$fp	29	0x7ffffef0
\$sp	30	0x00000000
\$ra	31	0x00400074
\$pc		0x00400046
\$hi		0x00000000
\$lo		0x00000002

Memory after we have made all of our recursive calls. Each stack contains the return address to the previous stack with the first stack returning to main(). The final stack GCD(4,0) begins the deallocation and return process.

Text Segment

Bkpt	Address	Code	Basic	Source
0x00400028	0xafa8000c	sw \$8,0x0000000c(\$29)	16:	sw \$t0,12(\$fp) # save return address
0x0040002c	0x0010001d	jal 0x00400074	17:	j EXIT
0x00400030	0x0001000c	addi \$1,\$0,0x0000000c	21:	sub \$sp,\$sp,12
0x00400034	0x031e822	sub \$29,\$29,\$1		
0x00400038	0x8fbf0000	lw \$31,0x00000000(\$...	22:	sw \$ra,0(\$fp)
0x0040003c	0xafa40004	sw \$4,0x00000004(\$29)	23:	sw \$a0,4(\$fp)
0x00400040	0xafa50008	sw \$5,0x00000008(\$29)	24:	sw \$a1,8(\$fp)
0x00400044	0x00044021	addu \$9,\$0,\$4	25:	move \$t0,\$a0
0x00400048	0x00054821	addu \$9,\$0,\$5	26:	move \$t1,\$a1
0x0040004c	0x11200006	beq \$9,\$0,0x00000006	29:	beq \$t1,\$0, done #if second arg(a0 & a1) is 0 we are finished
0x00400050	0x0109001a	div \$9,\$9	30:	div \$t0,\$t1 #divide our operands . . . Quotient gets sent to LO and remainder gets sent to HI
0x00400054	0x00094021	addu \$8,\$0,\$9	31:	move \$t0,\$t1 # a = b in euclid's algorithm

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7ffffef0	0x00400028	0x00000002c	0x000000008	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7ffffef4	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7ffffef8	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7ffffefc	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7fffff00	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7fffff04	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7fffff08	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7fffff0c	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000
0x7fffff10	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000	0x000000000

Registers

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000004
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000004
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$s8	24	0x00000000
\$s9	25	0x00000000
\$t0	26	0x00000000
\$t1	27	0x00000000
\$gp	28	0x10008000
\$fp	29	0x7ffffef0
\$sp	30	0x00000000
\$ra	31	0x00400074
\$pc		0x00400074
\$hi		0x00000000
\$lo		0x00000002

Here we see that our after deallocation we have our result stored in the initial space in memory we allocated when we first ran the program.

X86 Intel

Next we will look at Intel's x86 Processor's handling of recursive calls. Here is my C++ code written for this program. It very simply follows Euclid's algorithm pseudo code given to us by Professor Gertner.

```
1 // Christian Suleiman
2 // Recursive GCD
3
4 #include <iostream>
5 using namespace std;
6
7 int gcd(int a, int b) {
8     if (b == 0) { ≤ 1ms elapsed
9         return a;
10    }
11
12    else {
13        gcd(b, a % b);
14    }
15 }
16
17 int main() {
18     cout << gcd(25, 10);
19
20 }
```


Disassembly: Address: main(void)

```

0053183E int 3
0053183F int 3
... C:\Users\Chris\Desktop\College Stuff\CS343 Labs\SuleimanTakeh
int main() {
00531840 push ebp
00531841 mov ebp,esp
00531843 sub esp,0C0h
00531849 push ebx
0053184A push esi
0053184B push edi
0053184C lea edi,[ebp-0C0h]
00531852 mov ecx,30h
00531857 mov eax,0CCCCCCCCh
0053185C rep stos dword ptr esi[edi]
0053185E mov ecx,offset _AA87FFF8.Source@cpp (053C026h)
00531863 call @_CheckForDebuggerJustMyCode@4 (053121Ch)
cout << gcd(44, 8);
00531868 push 8
0053186A push 2Ch
0053186C call gcd (053110Eh) ; sime elapsed
00531871 add esp,8
00531874 mov esi,esp
00531876 mov eax
00531877 mov ecx,dword ptr [__imp_?cout@std@3V?basic_os
0053187D call dword ptr [__imp_std::basic_ostream<char,st
00531883 mov esi,esp
00531885 call __RTC_CheckEsp (0531226h)
}
0053188A xor eax,eax
0053188C pop edi
0053188D pop esi
0053188E pop ebx
0053188F add esp,0C0h
00531895 cmp nbp,esp

```

Registers: EAX = 0053C026, EBX = 00708000, ECX = 0053C026, EDX = 00000001, ESI = 00531339, EDI = 0053FC58, EIP = 0053186C, ESP = 0053FB84, EBP = 0053FC58, EFL = 00000246

Memory: Address: 0x0093F7D8

Main() → calling GCD(44,8)

Little Endian – read as 00 00 00 08

Here we should note that Intel follow LITTLE endian format for storing integers so 8 is stored as 08 00 00 00. This is the main body calling GCD.

Disassembly: Address: gcd(int, int)

```

#include <iostream>
using namespace std;

int gcd(int a, int b) {
00531780 push ebp
00531781 mov ebp,esp
00531783 sub esp,0C0h
00531789 push ebx
0053178A push esi
0053178B push edi
0053178C lea edi,[ebp-0C0h]
005317C2 mov ecx,30h
005317C7 mov eax,0CCCCCCCCh
005317CC rep stos dword ptr esi[edi]
005317CE mov ecx,offset _AA87FFF8.Source@cpp (053C026h)
005317D3 call @_CheckForDebuggerJustMyCode@4 (053121Ch)
if (b == 0) {
005317D8 cmp dword ptr [b],0
005317DC jne gcd+35h (05317E5h)
return a;
005317DE mov eax,dword ptr [a]
005317E1 jmp gcd+49h (05317F9h)
}
else {
005317E3 jmp gcd+49h (05317F9h)
gcd(b, a % b);
005317E5 mov eax,dword ptr [a]
005317E8 cdq
005317E9 idiv eax,dword ptr [b]
005317EC push edx
005317ED mov eax,dword ptr [b]
005317F0 push eax
005317F1 call gcd (053110Eh)
005317F6 add esp,8
}
}

```

Base Pointer pointing to previous call stack

Final Stack – returns A if B = 0

GCD(4, 0)

GCD(8, 4)

Main() → calling GCD(44,8)

Here we see the stacks being constructed for each call, similarly to the MIPS implementation. When we are in the final stack, EBP is pointing to the address of the previous stack so that we can return to it.

64-Bit Intel Processor on Linux Kali

For this assignment I simulated the Kali Linux OS through Oracle's VirtualBox software since I am on a Windows computer. Here is my C code written in the Linux terminal using Vim, almost identical to the previous implementation.

```
int gcd(int a, int b) {  
    if (b == 0) {  
        return a;  
    }  
    else {  
        gcd(b, a % b);  
    }  
}  
  
int main() {  
    int res = gcd(44, 8);  
    return 0;  
}
```

```

(gdb) nexti
0x000055555555165      11      int res = gcd(44, 8);
0x00005555555515b <main+8>: be 08 00 00 00 mov    $0x8,%esi
0x000055555555160 <main+13>: bf 2c 00 00 00 mov    $0x2c,%edi
=> 0x000055555555165 <main+18>: e8 bb ff ff ff callq  0x55555555125 <gcd>
0x00005555555516a <main+23>: 89 45 fc      mov    %eax,-0x4(%rbp)
(gdb) print res
$1 = 0
(gdb) print &res
$2 = (int *) 0x7fffffffe1cc
(gdb) x/32xw $rsp
0x7fffffffe1c0: 0xfffffe2b0      0x00007fff      0x00000000      0x00000000
0x7fffffffe1d0: 0x55555180      0x00005555      0xf7e1dbbb      0x00007fff
0x7fffffffe1e0: 0x00000000      0x00000000      0xffffe2b8      0x00007fff
0x7fffffffe1f0: 0x00040000      0x00000001      0x55555153      0x00005555
0x7fffffffe200: 0x00000000      0x00000000      0xa836139b      0x31384163
0x7fffffffe210: 0x55555040      0x00005555      0xffffe2b0      0x00007fff
0x7fffffffe220: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe230: 0xc8f6139b      0x646d1436      0xbdd0139b      0x646d040a
(gdb)

```

In our code, the return value of GCD is stored in int res. Here we can check the address of res and see that it does not contain a value at the beginning of the program.

```

Breakpoint 2, gcd (a=44, b=8) at gcd.c:2
2      if (b == 0) {
=> 0x000055555555133 <gcd+14>: 83 7d f8 00      cml    $0x0, -0x8(%rbp)
    0x000055555555137 <gcd+18>: 75 05      jne    0x5555555513e <gcd+25>
(gdb) next 5

Breakpoint 2, gcd (a=8, b=4) at gcd.c:2
2      if (b == 0) {
=> 0x000055555555133 <gcd+14>: 83 7d f8 00      cml    $0x0, -0x8(%rbp)
    0x000055555555137 <gcd+18>: 75 05      jne    0x5555555513e <gcd+25>
(gdb) next 5

Breakpoint 2, gcd (a=4, b=0) at gcd.c:2
2      if (b == 0) {
=> 0x000055555555133 <gcd+14>: 83 7d f8 00      cml    $0x0, -0x8(%rbp)
    0x000055555555137 <gcd+18>: 75 05      jne    0x5555555513e <gcd+25>
(gdb) nexti
0x000055555555137      2      if (b == 0) {
    0x000055555555133 <gcd+14>: 83 7d f8 00      cml    $0x0, -0x8(%rbp)
=> 0x000055555555137 <gcd+18>: 75 05      jne    0x5555555513e <gcd+25>
(gdb) x/32xw $rsp
0x7fffffff160: 0x00000000      0x00000000      0x00000000      0x00000004
0x7fffffff170: 0xffffe190      0x00007fff      0x555555151      0x00005555
0x7fffffff180: 0xffffe1a6      0x00007fff      0x00000004      0x00000008
0x7fffffff190: 0xffffe1b0      0x00007fff      0x555555151      0x00005555
0x7fffffff1a0: 0x00000000      0x00000000      0x00000008      0x0000002c
0x7fffffff1b0: 0xffffe1d0      0x00007fff      0x55555516a      0x00005555
0x7fffffff1c0: 0xffffe2b0      0x00007fff      0x00000000      0x00000000
0x7fffffff1d0: 0x555555180      0x00005555      0xf7e1dbbb      0x00007fff
(gdb) print {a,b}
$3 = {4, 0}
(gdb) print {&a,&b}
$4 = {0x7fffffff16c, 0x7fffffff168}

```

Here we can see the construction of each stack that occurs after each function call. We see the arguments for each call are stored in their respective stacks.

```

0x7fffffff160: 0x0000000000000000      0x0000000400000000
0x7fffffff170: 0x00007fffffff190      0x000055555555151
0x7fffffff180: 0x00007fffffff1a6      0x0000000800000004
0x7fffffff190: 0x00007fffffff1b0      0x000055555555151
0x7fffffff1a0: 0x0000000000000000      0x0000002c00000008
0x7fffffff1b0: 0x00007fffffff1d0      0x00005555555516a
0x7fffffff1c0: 0x00007fffffff1e2b0      0x0000000000000000
0x7fffffff1d0: 0x000055555555180      0x00007ffff7e1dbbb
(gdb)

```

We should also note that the arguments are being stored in 64-bits here. Although integers seem to be Big Endian, they are actually stored as Little Endian.

```

=> 0x000055555555151 <gcd+44>: c9 leaveq
    0x000055555555152 <gcd+45>: c3 retq
(gdb) nexti
0x000055555555152      8      }
    0x000055555555151 <gcd+44>: c9 leaveq
=> 0x000055555555152 <gcd+45>: c3 retq
(gdb) nexti
gcd (a=8, b=4) at gcd.c:8
8      }
=> 0x000055555555151 <gcd+44>: c9 leaveq
    0x000055555555152 <gcd+45>: c3 retq
(gdb) nexti
gcd (a=44, b=8) at gcd.c:8
8      }
=> 0x000055555555151 <gcd+44>: c9 leaveq
    0x000055555555152 <gcd+45>: c3 retq
(gdb) nexti
0x000055555555152      8      }
    0x000055555555151 <gcd+44>: c9 leaveq
=> 0x000055555555152 <gcd+45>: c3 retq
(gdb) nexti
0x00005555555516a in main () at gcd.c:11
11      int res = gcd(44, 8);
    0x00005555555515b <main+8>: be 08 00 00 00 mov    $0x8,%esi
    0x000055555555160 <main+13>: bf 2c 00 00 00 mov    $0x2c,%edi
    0x000055555555165 <main+18>: e8 bb ff ff ff callq  0x55555555125 <gcd>
=> 0x00005555555516a <main+23>: 89 45 fc      mov    %eax,-0x4(%rbp)

```

Return of final GCD

Return of
penultimate GCD

Return of initial GCD
call back to main()

Here we see the returning to previous stacks until we return to main()


```
(gdb) print res
$5 = 4
(gdb) print &res
$6 = (int *) 0x7fffffffelcc
(gdb) x/32xw $rsp
0x7fffffffelc0: 0xffffe2b0      0x00007fff      0x00000000      0x00000004
0x7fffffffeld0: 0x55555180      0x00005555      0xf7e1dbbb      0x00007fff
0x7fffffffefe0: 0x00000000      0x00000000      0xffffe2b8      0x00007fff
0x7fffffffef0: 0x00040000      0x00000001      0x55555153      0x00005555
0x7fffffffef20: 0x00000000      0x00000000      0xa836139b      0x31384163
0x7fffffffef40: 0x55555040      0x00005555      0xffffe2b0      0x00007fff
0x7fffffffef60: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffef80: 0xc8f6139b      0x646d1436      0xbdd0139b      0x646d040a
(gdb) printf "esp:%x\nebp:%x\nrip:%x\n", $esp, $ebp, $rip, $eax
esp:ffffelc0
ebp:ffffeld0
rip:5555516d
eax:4
```

When we finally have our result, it is stored in register EAX. Once we return to main(), we store the value of EAX into our 'res' variable.

Conclusion

We have studied how recursive functions are handled by three different processors. Although they are relatively quick, the main disadvantage of recursive calls is the large amount of memory used up in creating stacks. This can lead to stack overflow. We have also observed the key differences and similarities between these three environments.