Recursive Procedure Calls

Brandon Chin

CSC342 - Instructor: Prof. Izidor Gertner

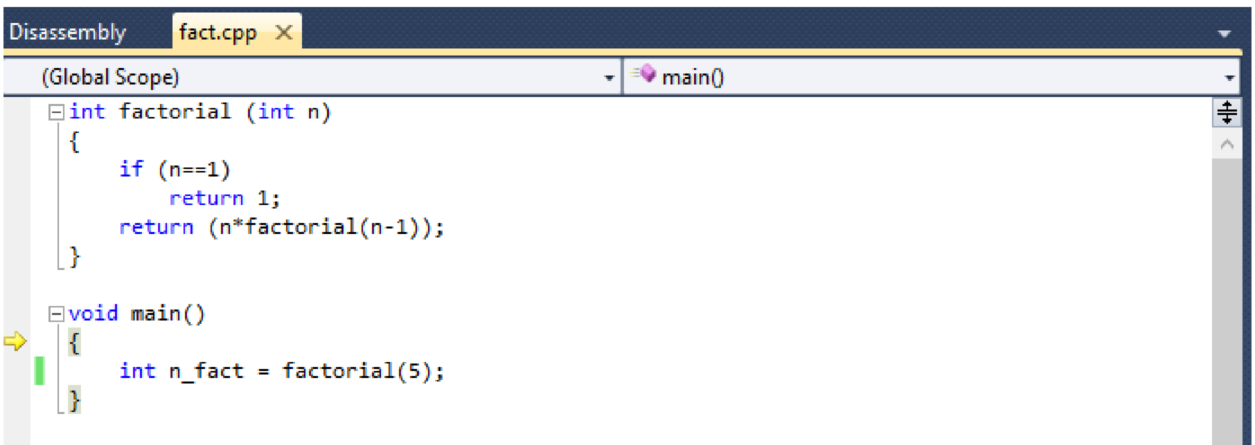
3/30/2015

Objective:

Recursion refers to the occurrence of allowing a function to call itself. Meaning the solution to a larger procedure depends on the smaller cases of the same procedure. Inside memory, a new stack frame must be created for each instance of the recursive procedure during execution. This is what we will primarily analyze though debugging and disassembling over three major operating systems; Windows, Linux, and MIPS/MARS.

Windows OS

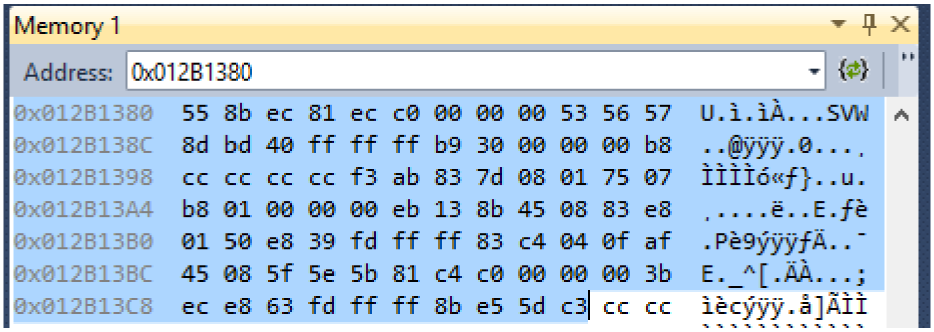
For Windows, we will be using the MS Debugger from Microsoft Visual C++ 2010 Express on a 64-bit Intel Core i5 CPU. Let's debug and analyze the following code segment written in C++:



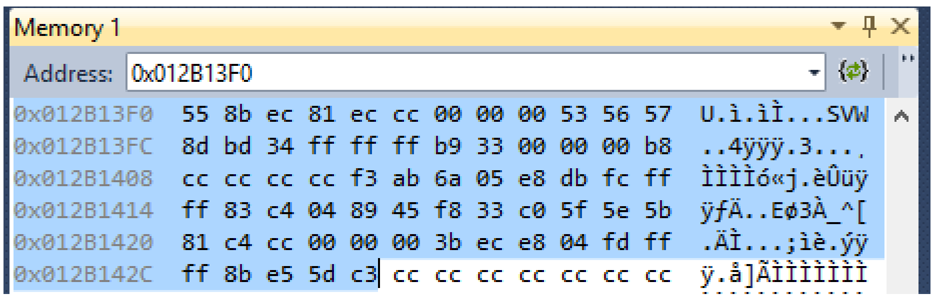
(C++ Code)

We have two functions -- the factorial function, which utilizes recursion to calculate the factorial of a value n, and the main function, which calls the factorial function on the value 5. The yellow arrow indicates the location where we will be entering the program (*i.e.* the main function).

The instruction code for our program is compiled and stored into memory.



(Instruction Code - Factorial Function)



(Instruction Code - Main Function)

Instance #1: Before First Instruction

Now let's begin debugging. The disassembly of the main function at the instance before any instruction is executed is shown below:

```
Disassembly  fact.cpp
Address: main(void)
Viewing Options
     8: void main()
     9: {
012B13F0 55                 push        ebp
012B13F1 8B EC              mov         ebp,esp
012B13F3 81 EC CC 00 00 00  sub         esp,0CCh
012B13F9 53                 push        ebx
012B13FA 56                 push        esi
012B13FB 57                 push        edi
012B13FC 8D BD 34 FF FF FF  lea         edi,[ebp-0CCh]
012B1402 B9 33 00 00 00     mov         ecx,33h
012B1407 B8 CC CC CC CC     mov         eax,0CCCCCCCCh
012B140C F3 AB              rep stos    dword ptr es:[edi]
```

(Disassembly #1 - main())

```
Registers
 EAX = 00902110 EBX = 7F5AF000 ECX = 00903D20 EDX = 00000001 ESI = 012B1109
 EDI = 012B1109 EIP = 012B13F0 ESP = 0044FA14 EBP = 0044FA60 EFL = 00000206
```

(Registers #1)

Let's bring our attention to three specific registers which are highlighted above:

- The EIP stores the address of the instruction which will be executed next. This is also indicated by the yellow arrow displayed next to the same address in the disassembly window.
- The ESP typically stores the address at the top of the stack. In this case, it contains the address which stores the return address that the main procedure must return to after execution is completed.
- The EBP typically stores the address at the base of the stack. In this case, it contains the address which stores the base pointer of the previous calling procedure.

```
Memory 1

0x0044FA14  af 19 2b 01
0x0044FA18  01 00 00 00
0x0044FA1C  20 3d 90 00
0x0044FA20  10 21 90 00
0x0044FA24  ee 31 21 52
```

(Memory #1 - ESP)

```
Memory 1

0x0044FA60  68 fa 44 00
0x0044FA64  df 17 2b 01
0x0044FA68  7c fa 44 00
0x0044FA6C  04 7c 65 75
0x0044FA70  00 f0 5a 7f
```

(Memory #1 - EBP)

*** NOTE -- Intel Processors store data in Little Endian Notation*

Instance #2: Creating the Main() Stack Frame

When the main function begins, a stack frame must be created in memory. This can be seen at the first instruction where a base pointer (ebp) is pushed into memory. This marks the beginning of the main stack frame, and does so by first setting up the stack pointer (esp).

```
Disassembly  fact.cpp
Address: main(void)
Viewing Options
      8: void main()
      9: {
012B13F0 55                   push        ebp
012B13F1 8B EC                mov         ebp,esp
012B13F3 81 EC CC 00 00 00    sub         esp,0CCh
012B13F9 53                   push        ebx
012B13FA 56                   push        esi
012B13FB 57                   push        edi
012B13FC 8D BD 34 FF FF FF    lea         edi,[ebp-0CCh]
012B1402 B9 33 00 00 00       mov         ecx,33h
012B1407 B8 CC CC CC CC       mov         eax,0CCCCCCCCh
012B140C F3 AB                rep stos    dword ptr es:[edi]
```

(Disassembly #2 - main())

```
Registers
EAX = 00902110 EBX = 7F5AF000 ECX = 00903D20 EDX = 00000001 ESI = 012B1109
EDI = 012B1109 EIP = 012B13F1 ESP = 0044FA10 EBP = 0044FA60 EFL = 00000206
```

(Registers #2)

```
Memory 1
0x0044FA10  60 fa 44 00
0x0044FA14  af 19 2b 01
0x0044FA18  01 00 00 00
0x0044FA1C  20 3d 90 00
0x0044FA20  10 21 90 00
```

(Memory #2 - ESP)

This instruction has set ESP equal to the address 0x0044FA10. It contains the address of the base pointer of the previous calling procedure, 0x0044FA60.

Instance #3: Set Base Pointer and Stack Pointer of Main()

Next, the stack pointer (esp) is moved to the location of the base pointer (ebp), which is achieved by setting the base pointer equal to the stack pointer. The base pointer will always point to the base of the stack, however, the stack pointer will mark the top of the stack, and will grow accordingly as data is pushed and popped into and from the stack.

```
Disassembly  fact.cpp
Address: main(void)
Viewing Options
     8: void main()
     9: {
012B13F0 55                   push        ebp
012B13F1 8B EC                mov         ebp,esp
012B13F3 81 EC CC 00 00 00    sub         esp,0CCh
012B13F9 53                   push        ebx
012B13FA 56                   push        esi
012B13FB 57                   push        edi
012B13FC 8D BD 34 FF FF FF    lea         edi,[ebp-0CCh]
012B1402 B9 33 00 00 00       mov         ecx,33h
012B1407 B8 CC CC CC CC       mov         eax,0CCCCCCCCh
012B140C F3 AB                rep stos    dword ptr es:[edi]
```

(Disassembly #3 - main())

```
Registers
EAX = 00902110 EBX = 7F5AF000 ECX = 00903D20 EDX = 00000001 ESI = 012B1109
EDI = 012B1109 EIP = 012B13F3 ESP = 0044FA10 EBP = 0044FA10 EFL = 00000206
```

(Register #3)

```
Memory 1
0x0044FA10  60 fa 44 00
0x0044FA14  af 19 2b 01
0x0044FA18  01 00 00 00
0x0044FA1C  20 3d 90 00
0x0044FA20  10 21 90 00
```

(Memory #3 - EBP)

Instance #4: Stack Space Allocation

```
Disassembly  fact.cpp
Address: main(void)
Viewing Options
     8: void main()
     9: {
012B13F0 55                   push        ebp
012B13F1 8B EC                mov         ebp,esp
012B13F3 81 EC CC 00 00 00    sub         esp,0CCh
012B13F9 53                   push        ebx
012B13FA 56                   push        esi
012B13FB 57                   push        edi
012B13FC 8D BD 34 FF FF FF    lea         edi,[ebp-0CCh]
012B1402 B9 33 00 00 00       mov         ecx,33h
012B1407 B8 CC CC CC CC       mov         eax,0CCCCCCCCh
012B140C F3 AB                rep stos    dword ptr es:[edi]
```

(Disassembly #4 - main())

```
Registers
 EAX = 00902110 EBX = 7F5AF000 ECX = 00903D20 EDX = 00000001 ESI = 012B1109
 EDI = 012B1109 EIP = 012B13F9 ESP = 0044F944 EBP = 0044FA10 EFL = 00000216
```

(Registers #4)

```
Memory 1

0x0044F944  60 f9 44 00
0x0044F948  7c f9 44 00
0x0044F94C  70 f3 2a 50
0x0044F950  44 48 73 e5
0x0044F954  00 00 00 00
```

(Memory #4 - ESP)

The instruction "*81 EC CC 00 00 00 sub esp, 0CCh*" subtracts 204 bytes from the stack pointer, and stores the result by overwriting the stack pointer register. This can be seen from the instruction segment *CC 00 00 00*, or the segment *0CCh*, which represents the decimal number 204 in hexadecimal. What this is doing is moving the stack pointer and allocating 204 bytes between the stack pointer and the base pointer on stack. This brings the stack pointer to the new location 0x0044F944 and clears the allocated space over the next few instructions.

```
Memory 1
 Address: 0x0044F944
0x0044F944  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F962  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F980  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F99E  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F9BC  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F9DA  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F9F8  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc 60 fa 44 00 af 19
```

(Memory #4 - ESP to EBP is cleared)

Instance #5: Call and Jump to Factorial(5)

```
Disassembly  fact.cpp
 Address: main(void)
 Viewing Options
    10:     int n_fact = factorial(5);
012B140E 6A 05              push        5
012B1410 E8 DB FC FF FF     call        factorial (12B10F0h)
012B1415 83 C4 04           add         esp,4
012B1418 89 45 F8           mov         dword ptr [n_fact],eax
    11: }
```

(Disassembly #5 - main())

```
Registers
EAX = CCCCCCCC EBX = 7F5AF000 ECX = 00000000 EDX = 00000001 ESI = 012B1109
EDI = 0044FA10 EIP = 012B1410 ESP = 0044F934 EBP = 0044FA10 EFL = 00000216
```

(Registers #5)

```
Memory 1

0x0044F934  05 00 00 00
0x0044F938  09 11 2b 01
0x0044F93C  09 11 2b 01
0x0044F940  00 f0 5a 7f
0x0044F944  cc cc cc cc
```

(Memory #5 - ESP #1)

This instruction pushes the parameter n = 5 to the stack, just before calling the factorial(int n) function. The value can be seen at the location of the current stack pointer.

Immediately after, the call factorial procedure instruction is made and the program is then taken to the address 0x012B10F0. This address contains the instruction code to jump to the start of the factorial function in memory, which is at address 0x012B1380. In addition, the return address is stored in the address 0x0044F930 by the stack pointer, which contains 0x012B1415 -- the instruction immediately after the call factorial procedure.

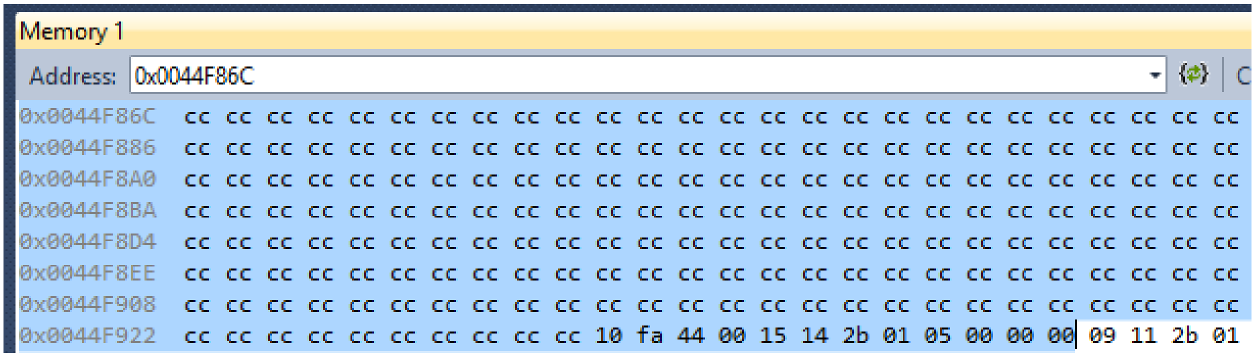
```
Memory 1

0x0044F930  15 14 2b 01
0x0044F934  05 00 00 00
0x0044F938  09 11 2b 01
0x0044F93C  09 11 2b 01
0x0044F940  00 f0 5a 7f
```

(Memory #5 - ESP #2)

Instance #6: Factorial (5)

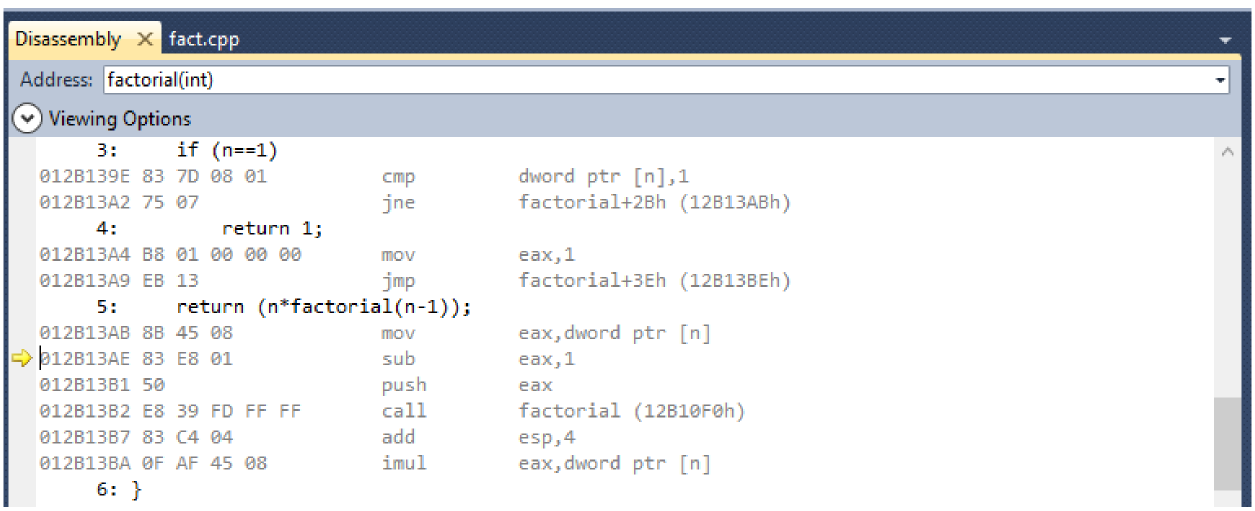
```
Disassembly  fact.cpp
Address: factorial(int)
Viewing Options
     1: int factorial (int n)
     2: {
012B1380 55                   push        ebp
012B1381 8B EC                mov         ebp,esp
012B1383 81 EC C0 00 00 00    sub         esp,0C0h
012B1389 53                   push        ebx
012B138A 56                   push        esi
012B138B 57                   push        edi
012B138C 8D BD 40 FF FF FF    lea         edi,[ebp-0C0h]
012B1392 B9 30 00 00 00       mov         ecx,30h
012B1397 B8 CC CC CC CC       mov         eax,0CCCCCCCCh
012B139C F3 AB                rep stos    dword ptr es:[edi]
```

(Disassembly #6 - factorial(5) #1)

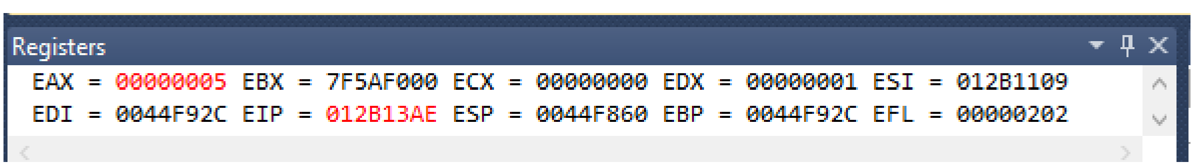


(Memory #6 - factorial(5) stack frame)

Now a stack frame must be set up for the factorial(5) procedure call. Similar to how the main() stack frame was setup, we can see the base pointer is initialized to address 0x0044F92C, 0C0h (192 bytes) are allocated and cleared in memory, and the stack pointer is currently set to address 0x0044F86C.



(Disassembly #6 - factorial(5) #2)



(Registers #6 - EAX stores value 5)

Here we can see the first conditional is ignored, and the program has jumped to the instruction at address 0x012B13AB. The value 5 is then stored into register EAX.

```
Registers
EAX = 00000004 EBX = 7F5AF000 ECX = 00000000 EDX = 00000001 ESI = 012B1109
EDI = 0044F92C EIP = 012B13B2 ESP = 0044F85C EBP = 0044F92C EFL = 00000202
```

(Registers #6 - EAX and ESP store value 4)

```
Memory 1
0x0044F85C  04 00 00 00
0x0044F860  10 fa 44 00
0x0044F864  09 11 2b 01
0x0044F868  00 f0 5a 7f
0x0044F86C  cc cc cc cc
```

(Memory #6 - ESP store value 4)

In the following instructions, 1 is subtracted from the value stored in EAX, then EAX is pushed onto the stack by the stack pointer at address 0x0044F85C. The program then recalls the factorial() procedure, and jumps back to the address of the first factorial() procedure instruction, 0x012B1380. This time, however, n is equal to 4.

The return address is also stored at the address 0x0044F858, containing the address 0x012B13B7, which is the instruction immediately after the instruction that called the procedure.

```
Memory 1
0x0044F858  b7 13 2b 01
0x0044F85C  04 00 00 00
0x0044F860  10 fa 44 00
0x0044F864  09 11 2b 01
0x0044F868  00 f0 5a 7f
```

(Memory #6 - ESP stores return address)

Instance #7: Factorial(4)

```
Memory 1
Address: 0x0044F794
0x0044F794  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F7AE  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F7C8  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F7E2  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F7FC  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F816  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F830  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F84A  cc cc cc cc cc cc cc cc cc cc 2c f9 44 00 b7 13 2b 01 04 00 00 00 10 fa 44 00
```

(Memory #7 - factorial(4) stack frame)

```
Registers
EAX = 00000003 EBX = 7F5AF000 ECX = 00000000 EDX = 00000001 ESI = 012B1109
EDI = 0044F854 EIP = 012B1380 ESP = 0044F780 EBP = 0044F854 EFL = 00000206
```

(Registers #7)

```
Memory 1
0x0044F780  b7 13 2b 01
0x0044F784  03 00 00 00
0x0044F788  2c f9 44 00
0x0044F78C  09 11 2b 01
0x0044F790  00 f0 5a 7f
```

(Memory #7 - return address and value 3)

Similarly, a stack frame is created for the factorial(4) procedure call -- a base pointer is initialized to address 0x0044F854, n is decremented by 1, this value is then pushed onto the stack, the return address is stored, and the program recalls the factorial function once again.

One thing to notice is the return address for this frame stores the same instruction as the return address of the previous factorial(5) frame. This is because each instance of the recursive procedure is called by the same instruction, and will return to the instruction immediately afterwards.

Instance #8: Factorial(3)

```
Memory 1
Address: 0x0044F6BC                                                         Co
0x0044F6BC  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F6D4  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F6EC  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F704  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F71C  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F734  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F74C  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F764  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F77C  54 f8 44 00 b7 13 2b 01 03 00 00 00 2c f9 44 00 09 11 2b 01 00 f0 5a 7f
```

(Memory #8 - factorial(3) stack frame)

```
Registers
EAX = 00000002 EBX = 7F5AF000 ECX = 00000000 EDX = 00000001 ESI = 012B1109
EDI = 0044F77C EIP = 012B1380 ESP = 0044F6A8 EBP = 0044F77C EFL = 00000202
```

(Registers #8)

```
Memory 1

0x0044F6A8  b7 13 2b 01
0x0044F6AC  02 00 00 00
0x0044F6B0  54 f8 44 00
0x0044F6B4  09 11 2b 01
0x0044F6B8  00 f0 5a 7f
```

(Memory #8 - return address and value 2)

Instance #9: Factorial(2)

```
Memory 1
Address: 0x0044F5E4                                                        Co
0x0044F5E4  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F5FE  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F618  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F632  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F64C  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F666  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F680  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F69A  cc cc cc cc cc cc cc cc cc cc 7c f7 44 00 b7 13 2b 01 02 00 00 00 54 f8 44 00
```

(Memory #9 - factorial(2) stack frame)

```
Registers
  EAX = 00000001 EBX = 7F5AF000 ECX = 00000000 EDX = 00000001 ESI = 012B1109
  EDI = 0044F6A4 EIP = 012B1380 ESP = 0044F5D0 EBP = 0044F6A4 EFL = 00000202
```

(Registers #9)

```
Memory 1

0x0044F5D0  b7 13 2b 01
0x0044F5D4  01 00 00 00
0x0044F5D8  7c f7 44 00
0x0044F5DC  09 11 2b 01
0x0044F5E0  00 f0 5a 7f
```

(Memory #9 - return address and value 1)

Instance #10: Factorial(1)

```
Memory 1
Address: 0x0044F50C
0x0044F50C  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F526  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F540  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F55A  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F574  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F58E  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F5A8  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0044F5C2  cc cc cc cc cc cc cc cc cc cc a4 f6 44 00 b7 13 2b 01 01 00 00 00 7c f7 44 00
```

(Memory #10 - factorial(1) stack frame)

Now we have reached the last instance of the recursive procedure. This time, condition where n = 1 (the base case) will not be skipped, and is going to run. In this case, the condition simply returns the value 1. This value is then stored into register EAX.

```
Disassembly   fact.cpp
Address: factorial(int)
Viewing Options
     3:     if (n==1)
012B139E 83 7D 08 01          cmp         dword ptr [n],1
012B13A2 75 07                jne         factorial+2Bh (12B13ABh)
     4:         return 1;
012B13A4 B8 01 00 00 00       mov         eax,1
012B13A9 EB 13                jmp         factorial+3Eh (12B13BEh)
     5:     return (n*factorial(n-1));
012B13AB 8B 45 08             mov         eax,dword ptr [n]
012B13AE 83 E8 01             sub         eax,1
012B13B1 50                   push        eax
012B13B2 E8 39 FD FF FF       call        factorial (12B10F0h)
012B13B7 83 C4 04             add         esp,4
012B13BA 0F AF 45 08          imul        eax,dword ptr [n]
     6: }
```

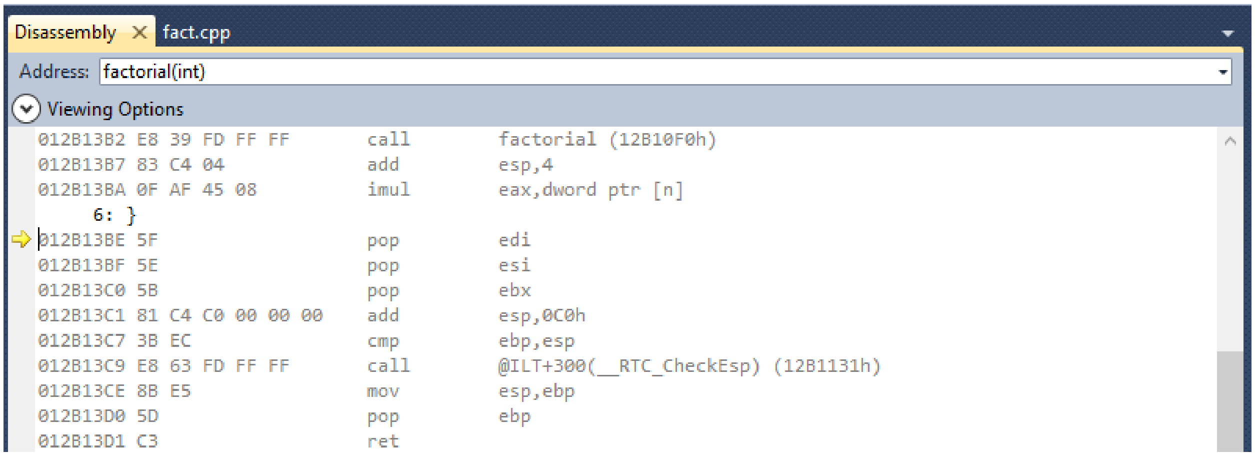
(Disassembly #10 - factorial(1))

```
Registers
EAX = 00000001 EBX = 7F5AF000 ECX = 00000000 EDX = 00000001 ESI = 012B1109
EDI = 0044F5CC EIP = 012B13A9 ESP = 0044F500 EBP = 0044F5CC EFL = 00000246
```

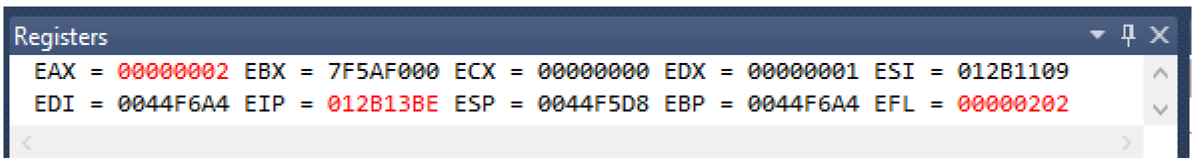
(Registers #10 - EAX = 1)

Instance #11: Return and Evaluate n_fact

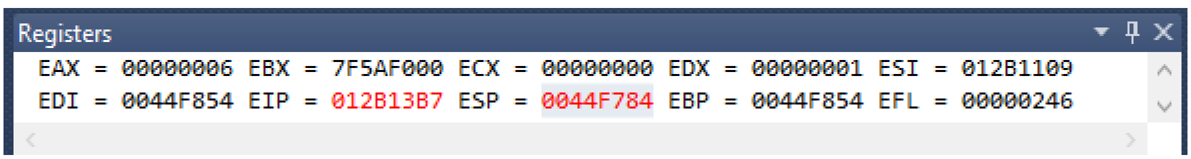
Next, the program jumps to the part of the code that will begin popping all of the factorial frames that are on stack. Each time, it multiplies our local variables cumulatively into register EAX. It will also continue to be called to the same return address (instruction 0x012B13B7) until all of the factorial stack frames are cleared.



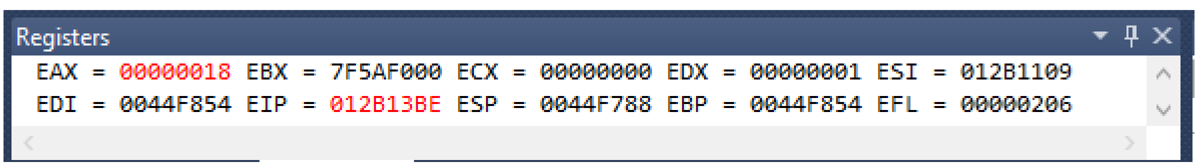
(Disassembly #11 - factorial())



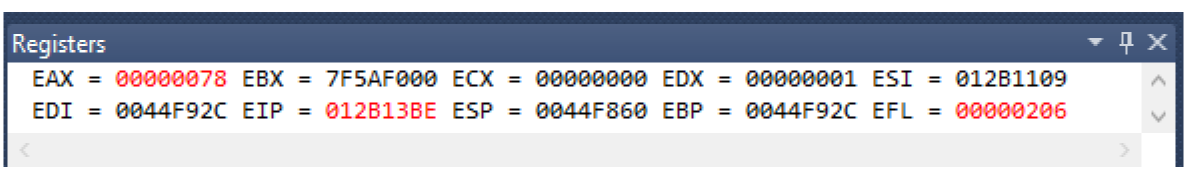
(Registers #11 - EAX = 2)



(Registers #11 - EAX = 6)



(Registers #11 - EAX = 18)



(Registers #11 - EAX = 78)

As we can see, EAX now equates to the hexadecimal value 0x00000078, or the decimal value 120, which is exactly what we expected. (5! = (5*4*3*2*1) = 120)

Lastly, we go back to main(), store the value of EAX into the variable n_fact (address 0x0044FA08), and proceed to popping the main stack frame, and returning to the address after the main call procedure.

```
Disassembly  fact.cpp
Address: main(void)
Viewing Options
    10:     int n_fact = factorial(5);
012B140E 6A 05                push        5
012B1410 E8 DB FC FF FF       call        factorial (12B10F0h)
012B1415 83 C4 04             add         esp,4
012B1418 89 45 F8             mov         dword ptr [n_fact],eax
    11: }
012B141B 33 C0                xor         eax,eax
012B141D 5F                   pop         edi
012B141E 5E                   pop         esi
012B141F 5B                   pop         ebx
012B1420 81 C4 CC 00 00 00    add         esp,0CCh
012B1426 3B EC                cmp         ebp,esp
012B1428 E8 04 FD FF FF       call        @ILT+300(__RTC_CheckEsp) (12B1131h)
012B142D 8B E5                mov         esp,ebp
012B142F 5D                   pop         ebp
012B1430 C3                   ret
```

(Disassembly #11 - main())

```
Memory 1

0x0044FA08  78 00 00 00
0x0044FA0C  cc cc cc cc
0x0044FA10  60 fa 44 00
0x0044FA14  af 19 2b 01
0x0044FA18  01 00 00 00
```

(Memory #11 - n_fact = 0x00000078)

Stack Visual

Address	Content		
0044FA60	0044FA68	*EBP of calling*	
...	...		
0044FA14	012B19AF	*Return Address*	Main()
0044FA10	0044FA60	*EBP of main()*	
...	...		
0044FA08	00000078	*n_fact*	
...	...		
0044F934	00000005	*n = 5*	Factorial(5)
0044F930	012B1415	*Return Address*	
0044F92C	0044FA10	*EBP of fact(5)*	
...	...		
0044F85C	00000004	*n = 4*	Factorial(4)
0044F858	012B13B7	*Return Address*	
0044F854	0044F92C	*EBP of fact(4)*	
...	...		
0044F784	00000003	*n = 3*	Factorial(3)
0044F780	012B13B7	*Return Address*	
0044F77C	0044F854	*EBP of fact(3)*	
...	...		
0044F6AC	00000002	*n = 2*	Factorial(2)
0044F6A8	012B13B7	*Return Address*	
0044F6A4	0044F77C	*EBP of fact(2)*	
...	...		
0044F5D4	00000001	*n = 1*	Factorial(1)
0044F5D0	012B13B7	*Return Address*	
0044F5CC	0044F6A4	*EBP of fact(1)*	
...	...		
...	...	*ESP*	

Stack Growth

Linux OS

For Linux, we will be using Ubuntu to run GCC and GDB on a 64 bit AMD Phenom II X4 CPU. Let's first look at the following code segment, written in C:

```
fact.cpp

int factorial(int n)
{
    if (n == 1)
        return 1;
    return (n * factorial(n - 1));
}

int main()
{
    int n_fact = factorial(5);

    return 0;
}
```

(Source Code in C)

Instance 1: Main Frame

Now, if we begin debugging, we can set a break point at main(), and look at its disassembly:

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000000000400518 <+0>:     push   %rbp
   0x0000000000400519 <+1>:     mov    %rsp,%rbp
   0x000000000040051c <+4>:     sub    $0x10,%rsp
=> 0x0000000000400520 <+8>:     mov    $0x5,%edi
   0x0000000000400525 <+13>:    callq  0x4004ed <factorial>
   0x000000000040052a <+18>:    mov    %eax,-0x4(%rbp)
   0x000000000040052d <+21>:    leaveq
   0x000000000040052e <+22>:    retq
End of assembler dump.
```

(Disassembly - main())

Here, we can see the main stack frame is created by first pushing the base pointer (rbp) to stack, moving the stack pointer (rsp) to the same location as the base pointer, and allocating 0x10 (16 bytes) for the frame.

```
(gdb) info registers
rax            0x400518 4195608
rbx            0x0      0
rcx            0x0      0
rdx            0x7fffffffde88   140737488346760
rsi            0x7fffffffde78   140737488346744
rdi            0x1      1
rbp            0x7fffffffdd90   0x7fffffffdd90
rsp            0x7fffffffdd80   0x7fffffffdd80
r8             0x7ffff7dd4e80   140737351863936
r9             0x7ffff7dea560   140737351951712
r10            0x7fffffffdc20   140737488346144
r11            0x7ffff7a36dd0   140737348070864
r12            0x400400 4195328
r13            0x7fffffffde70   140737488346736
r14            0x0      0
r15            0x0      0
rip            0x400520 0x400520 <main+8>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
---Type <return> to continue, or q <return> to quit---d
gs             0x0      0
```

(Registers)

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdda0:
 rip = 0x400520 in main (fact.c:10); saved rip = 0x7ffff7a36ec5
 source language c.
 Arglist at 0x7fffffffdd90, args:
 Locals at 0x7fffffffdd90, Previous frame's sp is 0x7fffffffdda0
 Saved registers:
  rbp at 0x7fffffffdd90, rip at 0x7fffffffdd98
```

```
(gdb) x $rsp
0x7fffffffdd80: 0xffffde70
```

(Info Frame)

We can see the instruction pointer (rip) is at 0x400520, the base pointer is at 0x7FFFFFFFDD90, and the stack pointer is at 0x7FFFFFFFDD80.

Instance 2: Factorial(5) Frame

Next, the program stores the parameter n = 5 into the register edi, followed by calling the factorial procedure. The program is then taken to the location in memory where the factorial procedure has been compiled.

```
(gdb) disassemble
Dump of assembler code for function factorial:
   0x00000000004004ed <+0>:     push   %rbp
   0x00000000004004ee <+1>:     mov    %rsp,%rbp
   0x00000000004004f1 <+4>:     sub    $0x10,%rsp
   0x00000000004004f5 <+8>:     mov    %edi,-0x4(%rbp)
   0x00000000004004f8 <+11>:    cmpl   $0x1,-0x4(%rbp)
   0x00000000004004fc <+15>:    jne    0x400505 <factorial+24>
   0x00000000004004fe <+17>:    mov    $0x1,%eax
   0x0000000000400503 <+22>:    jmp    0x400516 <factorial+41>
=> 0x0000000000400505 <+24>:    mov    -0x4(%rbp),%eax
   0x0000000000400508 <+27>:    sub    $0x1,%eax
   0x000000000040050b <+30>:    mov    %eax,%edi
   0x000000000040050d <+32>:    callq  0x4004ed <factorial>
   0x0000000000400512 <+37>:    imul   -0x4(%rbp),%eax
   0x0000000000400516 <+41>:    leaveq
   0x0000000000400517 <+42>:    retq
End of assembler dump.
```

(Disassembly - factorial())

First, a stack frame is created for this procedure call, then, the value 5, stored in register edi, is moved into the memory location of the base pointer offset by 4 bytes (rbp - 4). The program then jumps to the else statement of our code, line 5. It will then move this value into register eax, subtract 1 from this value, and then move it back into register edi, just before recalling the factorial procedure on this new parameter.

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdd80:
 rip = 0x400505 in factorial (fact.c:5); saved rip = 0x40052a
 called by frame at 0x7fffffffdda0
 source language c.
 Arglist at 0x7fffffffdd70, args: n=5
 Locals at 0x7fffffffdd70, Previous frame's sp is 0x7fffffffdd80
 Saved registers:
  rbp at 0x7fffffffdd70, rip at 0x7fffffffdd78
```

```
(gdb) x $rbp-4
0x7fffffffdd6c: 0x00000005
```

(Info Frame and content of (rbp - 4))

Instance 3: Factorial(4) Frame

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdd60:
 rip = 0x400505 in factorial (fact.c:5); saved rip = 0x400512
 called by frame at 0x7fffffffdd80
 source language c.
 Arglist at 0x7fffffffdd50, args: n=4
 Locals at 0x7fffffffdd50, Previous frame's sp is 0x7fffffffdd60
 Saved registers:
  rbp at 0x7fffffffdd50, rip at 0x7fffffffdd58
(gdb) x $rbp-4
0x7fffffffdd4c: 0x00000004
```

(Info Frame)

Instance 4: Factorial(3) Frame

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdd40:
 rip = 0x400505 in factorial (fact.c:5); saved rip = 0x400512
 called by frame at 0x7fffffffdd60
 source language c.
 Arglist at 0x7fffffffdd30, args: n=3
 Locals at 0x7fffffffdd30, Previous frame's sp is 0x7fffffffdd40
 Saved registers:
  rbp at 0x7fffffffdd30, rip at 0x7fffffffdd38
(gdb) x $rbp-4
0x7fffffffdd2c: 0x00000003
```

(Info Frame)

Instance 5: Factorial(2) Frame

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdd20:
 rip = 0x400505 in factorial (fact.c:5); saved rip = 0x400512
 called by frame at 0x7fffffffdd40
 source language c.
 Arglist at 0x7fffffffdd10, args: n=2
 Locals at 0x7fffffffdd10, Previous frame's sp is 0x7fffffffdd20
 Saved registers:
  rbp at 0x7fffffffdd10, rip at 0x7fffffffdd18
(gdb) x $rbp-4
0x7fffffffdd0c: 0x00000002
```

(Info Frame)

Instance 6: Factorial(1) Frame

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdd00:
 rip = 0x4004f8 in factorial (fact.c:3); saved rip = 0x400512
 called by frame at 0x7fffffffdd20
 source language c.
 Arglist at 0x7fffffffdcf0, args: n=1
 Locals at 0x7fffffffdcf0, Previous frame's sp is 0x7fffffffdd00
 Saved registers:
  rbp at 0x7fffffffdcf0, rip at 0x7fffffffdcf8
(gdb) x $rbp-4
0x7fffffffdcec: 0x00000001
```

(Info Frame)

Instance 7: Return

The base case is finally met, and the factorial procedure returns 1, pops the frame from stack and jumps back to the previous frame. The program will continue to do this, at the same time begin to evaluate the product of all the stack arguments.

```
(gdb) info registers
rax            0x1      1
```

```
(gdb) info registers
rax            0x2      2
```

```
(gdb) info registers
rax            0x6      6
```

```
(gdb) info registers
rax            0x18     24
```

```
(gdb) info registers
rax            0x78     120
```

We get the value for 5! to be 120, or 0x78, which is what we expected. Finally, the program will return to the instruction in main right after the instruction that called the first factorial function, and proceed to pop the main function off stack.

Stack Visual

Address	Content		
0x7FFFFFFFDD90	...	*RBP of main()*	Main()
...	...		
...	0x78	*n_fact*	
...	...		
0x7FFFFFFFDD70	0x7FFFFFFFDD90	*RBP of fact(5)*	Factorial(5)
0x7FFFFFFFDD6C	0x5	*n = 5*	
...	...		
0x7FFFFFFFDD50	0x7FFFFFFFDD70	*RBP of fact(4)*	Factorial(4)
0x7FFFFFFFDD4C	0x4	*n = 4*	
...	...		
0x7FFFFFFFDD30	0x7FFFFFFFDD50	*RBP of fact(3)*	Factorial(3)
0x7FFFFFFFDD2C	0x3	*n = 3*	
...	...		
0x7FFFFFFFDD10	0x7FFFFFFFDD30	*RBP of fact(2)*	Factorial(2)
0x7FFFFFFFDD0C	0x2	*n = 2*	
...	...		
0x7FFFFFFFDCF0	0x7FFFFFFFDD10	*RBP of fact(1)*	Factorial(1)
0x7FFFFFFFDCEC	0x1	*n = 1*	
...	...	*RSP*	

Stack Growth

MIPS/MARS

Let's disassemble the following piece of code written in MIPS assembly:

fact.asm

```
.text
        main:
                addi $a0, $a0, 5        # n = 5
                jal factorial           # call factorial procedure
                li $v0, 10
                syscall

        factorial:
                slti $t0, $a0, 1        # $t0 only equals 0 when $a0 = 1
                beq $t0, $zero, L1
                addi $v0, $zero, 1
                addi $sp, $sp, 8
                jr $ra

        L1:                             # the loop that handles decrementing n
                addi $sp, $sp, -8
                sw $ra, 4($sp)
                sw $a0, 0($sp)
                addi $a0, $a0, -1
                jal factorial
                lw $a0, 0($sp)
                lw $ra, 4($sp)
                addi $sp, $sp, 8
                mul $v0, $a0, $v0
                jr $ra
```

(MIPS Assembly Source Code)

Instance 1:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x20840005	addi $4,$4,0x00000005	3: addi $a0, $a0, 5 # n = 5
	0x00400004	0x0c100004	jal 0x00400010	4: jal factorial # call factorial procedure
	0x00400008	0x2402000a	addiu $2,$0,0x0000000a	5: li $v0, 10
	0x0040000c	0x0000000c	syscall	6: syscall
	0x00400010	0x28880001	slti $8,$4,0x00000001	9: slti $t0, $a0, 1 # $t0 only equals 0 when $a0 = 1
	0x00400014	0x11000003	beq $8,$0,0x00000003	10: beq $t0, $zero, L1
	0x00400018	0x20020001	addi $2,$0,0x00000001	11: addi $v0, $zero, 1
	0x0040001c	0x23bd0008	addi $29,$29,0x0000...	12: addi $sp, $sp, 8
	0x00400020	0x03e00008	jr $31	13: jr $ra
	0x00400024	0x23bdfff8	addi $29,$29,0xffff...	16: addi $sp, $sp, -8

(Text Segment - main)

$a0	4	0x00000005

(Register $a0 stores parameter n = 5)

Instance 2:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400024	0x23bdfff8	addi $29,$29,0xffff...	16: addi $sp, $sp, -8
	0x00400028	0xafbf0004	sw $31,0x00000004($29)	17: sw $ra, 4($sp)
	0x0040002c	0xafa40000	sw $4,0x00000000($29)	18: sw $a0, 0($sp)
	0x00400030	0x2084ffff	addi $4,$4,0xffffffff	19: addi $a0, $a0, -1
	0x00400034	0x0c100004	jal 0x00400010	20: jal factorial
	0x00400038	0x8fa40000	lw $4,0x00000000($29)	21: lw $a0, 0($sp)
	0x0040003c	0x8fbf0004	lw $31,0x00000004($29)	22: lw $ra, 4($sp)
	0x00400040	0x23bd0008	addi $29,$29,0x0000...	23: addi $sp, $sp, 8
	0x00400044	0x70821002	mul $2,$4,$2	24: mul $v0, $a0, $v0
	0x00400048	0x03e00008	jr $31	25: jr $ra

(Text Segment - L1)

$sp	29	0x7fffeff4
$fp	30	0x00000000
$ra	31	0x00400008
pc		0x00400028

(Stack pointer ($sp), return address ($ra), program pointer ($pc))

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefe0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000005	0x00400008	0x00000000
0x7ffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff0a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff0c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

(Data Segment - return address and n = 5)

Here we can see the value 5 has been stored into memory address 0x7FFFEFF4, and the return address for this procedure call in 0x7FFFEFF8.

Instance 3:

Text Segment

Bkpt	Address	Code	Basic	Source	
	0x00400024	0x23bdfff8	addi $29,$29,0xffff...	16:	addi $sp, $sp, -8
	0x00400028	0xafbf0004	sw $31,0x00000004($29)	17:	sw $ra, 4($sp)
	0x0040002c	0xafa40000	sw $4,0x00000000($29)	18:	sw $a0, 0($sp)
	0x00400030	0x2084ffff	addi $4,$4,0xffffffff	19:	addi $a0, $a0, -1
	0x00400034	0x0c100004	jal 0x00400010	20:	jal factorial
	0x00400038	0x8fa40000	lw $4,0x00000000($29)	21:	lw $a0, 0($sp)
	0x0040003c	0x8fbf0004	lw $31,0x00000004($29)	22:	lw $ra, 4($sp)
	0x00400040	0x23bd0008	addi $29,$29,0x0000...	23:	addi $sp, $sp, 8
	0x00400044	0x70821002	mul $2,$4,$2	24:	mul $v0, $a0, $v0
	0x00400048	0x03e00008	jr $31	25:	jr $ra

(Text Segment - L1)

$a0	4	0x00000004

(Register $a0 stores parameter n = 4)

The value n, stored in register $a0, has now been decremented by one.

Instance 4:

Text Segment

Bkpt	Address	Code	Basic	Source	
	0x00400010	0x28880001	slti $8,$4,0x00000001	9:	slti $t0, $a0, 1 # $t0 only equals 0 when $a0 = 1
	0x00400014	0x11000003	beq $8,$0,0x00000003	10:	beq $t0, $zero, L1
	0x00400018	0x20020001	addi $2,$0,0x00000001	11:	addi $v0, $zero, 1
	0x0040001c	0x23bd0008	addi $29,$29,0x0000...	12:	addi $sp, $sp, 8
	0x00400020	0x03e00008	jr $31	13:	jr $ra
	0x00400024	0x23bdfff8	addi $29,$29,0xffff...	16:	addi $sp, $sp, -8
	0x00400028	0xafbf0004	sw $31,0x00000004($29)	17:	sw $ra, 4($sp)
	0x0040002c	0xafa40000	sw $4,0x00000000($29)	18:	sw $a0, 0($sp)
	0x00400030	0x2084ffff	addi $4,$4,0xffffffff	19:	addi $a0, $a0, -1
	0x00400034	0x0c100004	jal 0x00400010	20:	jal factorial

(Text Segment - L1)

$sp	29	0x7fffefec
$fp	30	0x00000000
$ra	31	0x00400038
pc		0x00400030

Stack pointer ($sp), return address ($ra), program pointer ($pc))

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefe0	0x00000000	0x00000000	0x00000000	0x00000004	0x00400038	0x00000005	0x00400008	0x00000000
0x7ffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff0a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff0c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

(Data Segment - return address and n = 4)

Instance 5:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400010	0x28880001	slti $8,$4,0x00000001	9: slti $t0, $a0, 1 # $t0 only equals 0 when $a0 = 1
	0x00400014	0x11000003	beq $8,$0,0x00000003	10: beq $t0, $zero, L1
	0x00400018	0x20020001	addi $2,$0,0x00000001	11: addi $v0, $zero, 1
	0x0040001c	0x23bd0008	addi $29,$29,0x0000...	12: addi $sp, $sp, 8
	0x00400020	0x03e00008	jr $31	13: jr $ra
	0x00400024	0x23bdfff8	addi $29,$29,0xffff...	16: addi $sp, $sp, -8
	0x00400028	0xafbf0004	sw $31,0x00000004($29)	17: sw $ra, 4($sp)
	0x0040002c	0xafa40000	sw $4,0x00000000($29)	18: sw $a0, 0($sp)
	0x00400030	0x2084ffff	addi $4,$4,0xffffffff	19: addi $a0, $a0, -1
	0x00400034	0x0c100004	jal 0x00400010	20: jal factorial

(Text Segment - L1)

$a0	4	0x00000003

(Register $a0 stores parameter n = 3)

Instance 6:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400010	0x28880001	slti $8,$4,0x00000001	9: slti $t0, $a0, 1 # $t0 only equals 0 when $a0 = 1
	0x00400014	0x11000003	beq $8,$0,0x00000003	10: beq $t0, $zero, L1
	0x00400018	0x20020001	addi $2,$0,0x00000001	11: addi $v0, $zero, 1
	0x0040001c	0x23bd0008	addi $29,$29,0x0000...	12: addi $sp, $sp, 8
	0x00400020	0x03e00008	jr $31	13: jr $ra
	0x00400024	0x23bdfff8	addi $29,$29,0xffff...	16: addi $sp, $sp, -8
	0x00400028	0xafbf0004	sw $31,0x00000004($29)	17: sw $ra, 4($sp)
	0x0040002c	0xafa40000	sw $4,0x00000000($29)	18: sw $a0, 0($sp)
	0x00400030	0x2084ffff	addi $4,$4,0xffffffff	19: addi $a0, $a0, -1
	0x00400034	0x0c100004	jal 0x00400010	20: jal factorial

(Text Segment - L1)

$sp	29	0x7fffefe4
$fp	30	0x00000000
$ra	31	0x00400038
pc		0x00400030

(Stack pointer ($sp), return address ($ra), program pointer ($pc))

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefe0	0x00000000	0x00000003	0x00400038	0x00000004	0x00400038	0x00000005	0x00400008	0x00000000
0x7ffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff0a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff0c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

(Data Segment - return address and n = 3)

Instance 7:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400010	0x28880001	slti $8,$4,0x00000001	9: slti $t0, $a0, 1 # $t0 only equals 0 when $a0 = 1
	0x00400014	0x11000003	beq $8,$0,0x00000003	10: beq $t0, $zero, L1
	0x00400018	0x20020001	addi $2,$0,0x00000001	11: addi $v0, $zero, 1
	0x0040001c	0x23bd0008	addi $29,$29,0x0000...	12: addi $sp, $sp, 8
	0x00400020	0x03e00008	jr $31	13: jr $ra
	0x00400024	0x23bdfff8	addi $29,$29,0xffff...	16: addi $sp, $sp, -8
	0x00400028	0xafbf0004	sw $31,0x00000004($29)	17: sw $ra, 4($sp)
	0x0040002c	0xafa40000	sw $4,0x00000000($29)	18: sw $a0, 0($sp)
	0x00400030	0x2084ffff	addi $4,$4,0xffffffff	19: addi $a0, $a0, -1
	0x00400034	0x0c100004	jal 0x00400010	20: jal factorial

(Text Segment - L1)

$a0	4	0x00000002

(Register $a0 stores parameter n = 2)

Instance 8:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400010	0x28880001	slti $8,$4,0x00000001	9: slti $t0, $a0, 1 # $t0 only equals 0 when $a0 = 1
	0x00400014	0x11000003	beq $8,$0,0x00000003	10: beq $t0, $zero, L1
	0x00400018	0x20020001	addi $2,$0,0x00000001	11: addi $v0, $zero, 1
	0x0040001c	0x23bd0008	addi $29,$29,0x0000...	12: addi $sp, $sp, 8
	0x00400020	0x03e00008	jr $31	13: jr $ra
	0x00400024	0x23bdfff8	addi $29,$29,0xffff...	16: addi $sp, $sp, -8
	0x00400028	0xafbf0004	sw $31,0x00000004($29)	17: sw $ra, 4($sp)
	0x0040002c	0xafa40000	sw $4,0x00000000($29)	18: sw $a0, 0($sp)
	0x00400030	0x2084ffff	addi $4,$4,0xffffffff	19: addi $a0, $a0, -1
	0x00400034	0x0c100004	jal 0x00400010	20: jal factorial

(Text Segment - L1)

$sp	29	0x7fffefdc
$fp	30	0x00000000
$ra	31	0x00400038
pc		0x00400030

(Stack pointer ($sp), return address ($ra), program pointer ($pc))

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefc0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000002
0x7fffefe0	0x00400038	0x00000003	0x00400038	0x00000004	0x00400038	0x00000005	0x00400008	0x00000000
0x7ffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff0a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

(Data Segment - return address and n = 2)

Instance 9:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400010	0x28880001	slti $8,$4,0x00000001	9: slti $t0, $a0, 1 # $t0 only equals 0 when $a0 = 1
	0x00400014	0x11000003	beq $8,$0,0x00000003	10: beq $t0, $zero, L1
	0x00400018	0x20020001	addi $2,$0,0x00000001	11: addi $v0, $zero, 1
	0x0040001c	0x23bd0008	addi $29,$29,0x0000...	12: addi $sp, $sp, 8
	0x00400020	0x03e00008	jr $31	13: jr $ra
	0x00400024	0x23bdfff8	addi $29,$29,0xffff...	16: addi $sp, $sp, -8
	0x00400028	0xafbf0004	sw $31,0x00000004($29)	17: sw $ra, 4($sp)
	0x0040002c	0xafa40000	sw $4,0x00000000($29)	18: sw $a0, 0($sp)
	0x00400030	0x2084ffff	addi $4,$4,0xffffffff	19: addi $a0, $a0, -1
	0x00400034	0x0c100004	jal 0x00400010	20: jal factorial

(Text Segment - L1)

$a0	4	0x00000001

(Register $a0 stores parameter n = 1)

Instance 10:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400010	0x28880001	slti $8,$4,0x00000001	9: slti $t0, $a0, 1 # $t0 only equals 0 when $a0 = 1
	0x00400014	0x11000003	beq $8,$0,0x00000003	10: beq $t0, $zero, L1
	0x00400018	0x20020001	addi $2,$0,0x00000001	11: addi $v0, $zero, 1
	0x0040001c	0x23bd0008	addi $29,$29,0x0000...	12: addi $sp, $sp, 8
	0x00400020	0x03e00008	jr $31	13: jr $ra
	0x00400024	0x23bdfff8	addi $29,$29,0xffff...	16: addi $sp, $sp, -8
	0x00400028	0xafbf0004	sw $31,0x00000004($29)	17: sw $ra, 4($sp)
	0x0040002c	0xafa40000	sw $4,0x00000000($29)	18: sw $a0, 0($sp)
	0x00400030	0x2084ffff	addi $4,$4,0xffffffff	19: addi $a0, $a0, -1
	0x00400034	0x0c100004	jal 0x00400010	20: jal factorial

(Text Segment - L1)

$sp	29	0x7fffefd4
$fp	30	0x00000000
$ra	31	0x00400038
pc		0x00400030

(Stack pointer ($sp), return address ($ra), program pointer ($pc))

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefc0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000001	0x00400038	0x00000002
0x7fffefe0	0x00400038	0x00000003	0x00400038	0x00000004	0x00400038	0x00000005	0x00400008	0x00000000
0x7ffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff0a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

(Data Segment - return address and n = 1)

Instance 11:

Now the stack frames must be popped and the values must multiplied.

lo		0x00000002

lo		0x00000006

lo		0x00000018

lo		0x00000078

Once again, we get the end result of 0x78, or 120.

Stack Visual

Address	Content		
0x7FFFEFF8	0x00400008	*return address*	Factorial(5)
0x7FFFEFF4	0x00000005	*n = 5*	
0x7FFFEFF0	0x00400038	*return address*	Factorial(4)
0x7FFFEFEC	0x00000004	*n = 4*	
0x7FFFEFE8	0x00400038	*return address*	Factorial(3)
0x7FFFEFE4	0x00000003	*n = 3*	
0x7FFFEFE0	0x00400038	*return address*	Factorial(2)
0x7FFFEFDC	0x00000002	*n = 2*	
0x7FFFEFD8	0x00400038	*return address*	Factorial(1)
0x7FFFEFD4	0x00000001	*n = 1*	

Stack Growth

Conclusion:

We have used three different platforms to analyze how a recursive factorial procedure is interpreted by the CPU and stored in memory. We saw how stack frames must be created in memory for each instance the procedure is recalled. At each instance, the argument of n is decremented by one and stored in memory. Stack frames are then destroyed once the recursive base case is met, and the program links back to the previous frame via the return address. During this process, the arguments at each frame level are multiplied to give us the final result of n factorial. This summarizes how the CPU handles recursive procedure calls.