SCC.131: Digital Systems
Memory layout (focus on x86-64)

Ioannis Chatzigeorgiou
(i.chatzigeorgiou@lancaster.ac.uk)

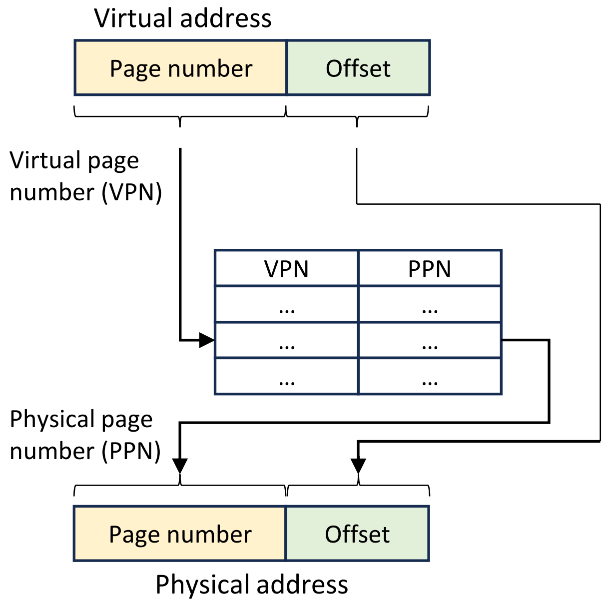
Summary of the last lecture

In the last lecture, we discussed about:

- The capabilities of micro:bit for wireless communication.
- The characteristics of the proprietary radio mode at 2.4 GHz.
- The concept of a datagram and key instructions for transmitting and receiving datagrams using variables of type PacketBuffer.
- Setting up a listener for the radio component and calling an event handler when a datagram is received.
- The creation of groups and the extraction of the received signal strength.
- Steps on how to build a simple one-way wireless communication system.

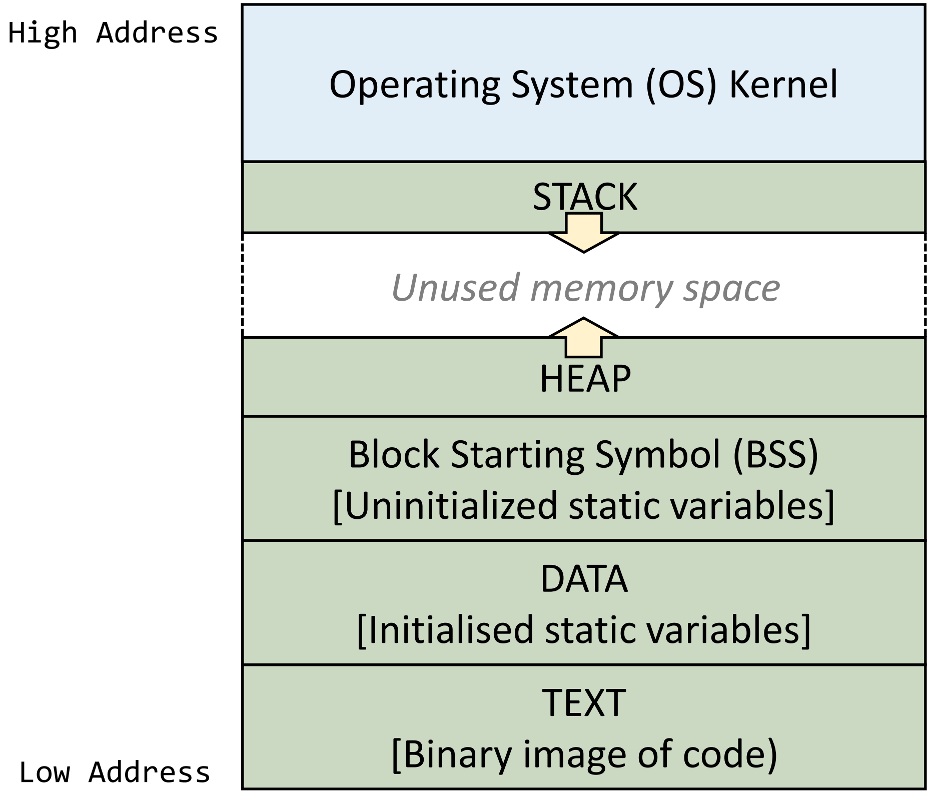
Virtual memory

- In principle, a 64-bit microprocessor can address 2^{64} bytes of byte-addressable memory.
- Read/write operations use **virtual** memory addresses.
- The **memory management unit** (MMU) in the microprocessor uses a lookup table, called the **translation lookaside buffer** (TLB), to translate virtual to physical addresses.



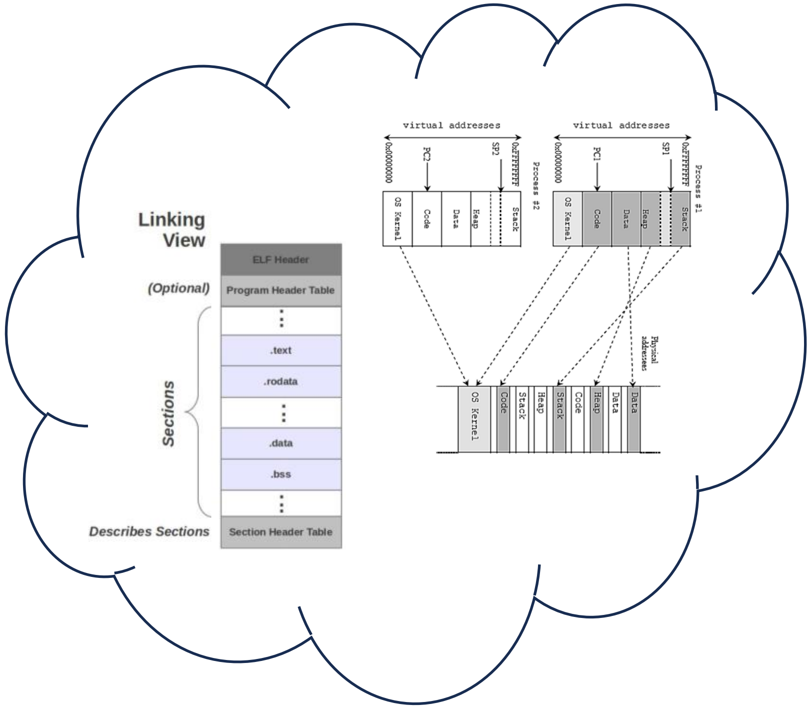
- The operating system dynamically allocates **pages** of memory (typically 4 KB per page for x86 architectures) to processes and creates entries in the TLB. Your code remains agnostic to coordination with other process when accessing memory.

Virtual memory layout



- **OS Kernel:** Memory reserved by the OS to monitor and control mapping between physical and virtual addresses.
- **STACK:** Data needed by function calls, including arguments and local variables. The set of values pushed for one function call is referred to as a *stack frame*.
- **HEAP:** Dynamic memory allocation for variables whose size is known at runtime and cannot be statically determined by the compiler before program execution (e.g., `malloc`, `new`, `free`, `delete`).
- **BSS** and **DATA**: Global variables that are initialized to zero or do not have explicit initialization (BSS), or global variables that have been initialized (DATA).
- **TEXT**: The binary executable instructions of a program.

Did I hear about virtual memory before?



Week 9, Lecture 1, Slides 6 and 18
(Compiler, assembler, linker and loader)

```c
#include <stdio.h>

void test(){
  int k;
  *(&k+288)=3;
}

int main() {
  int i = 5;
  char a[10]="TEST1";
  printf("%s\n",a);
  test();
  i=i+i;
  printf("%d\n",i);
}
```

Week 10, Lecture 1, Slide 14
(Debugging)

Virtual memory allocation – Examples (1/2)

- Compile C program to generate object file:

  `gcc -c SCC131_example.c`

- List the size (in bytes) of each section of the object file:

  `size SCC131_example.o`

```
text   data   bss   dec   hex filename
 130      0     0   130    82 SCC131_example.o
```

$\underbrace{\text{130 \quad 82}}$

Total size (in decimal and hexadecimal representation)

SCC131_example.c

```
#include <stdio.h>
int main(void) {
  printf("hello world\n");
}
```

Virtual memory allocation – Examples (2/2)

```
#include <stdio.h>
int i=5;
int main(void) {
  printf("hello world\n");
}
```

text	data	bss	dec	hex	filename
130	4	0	134	86	SCC131_example.o

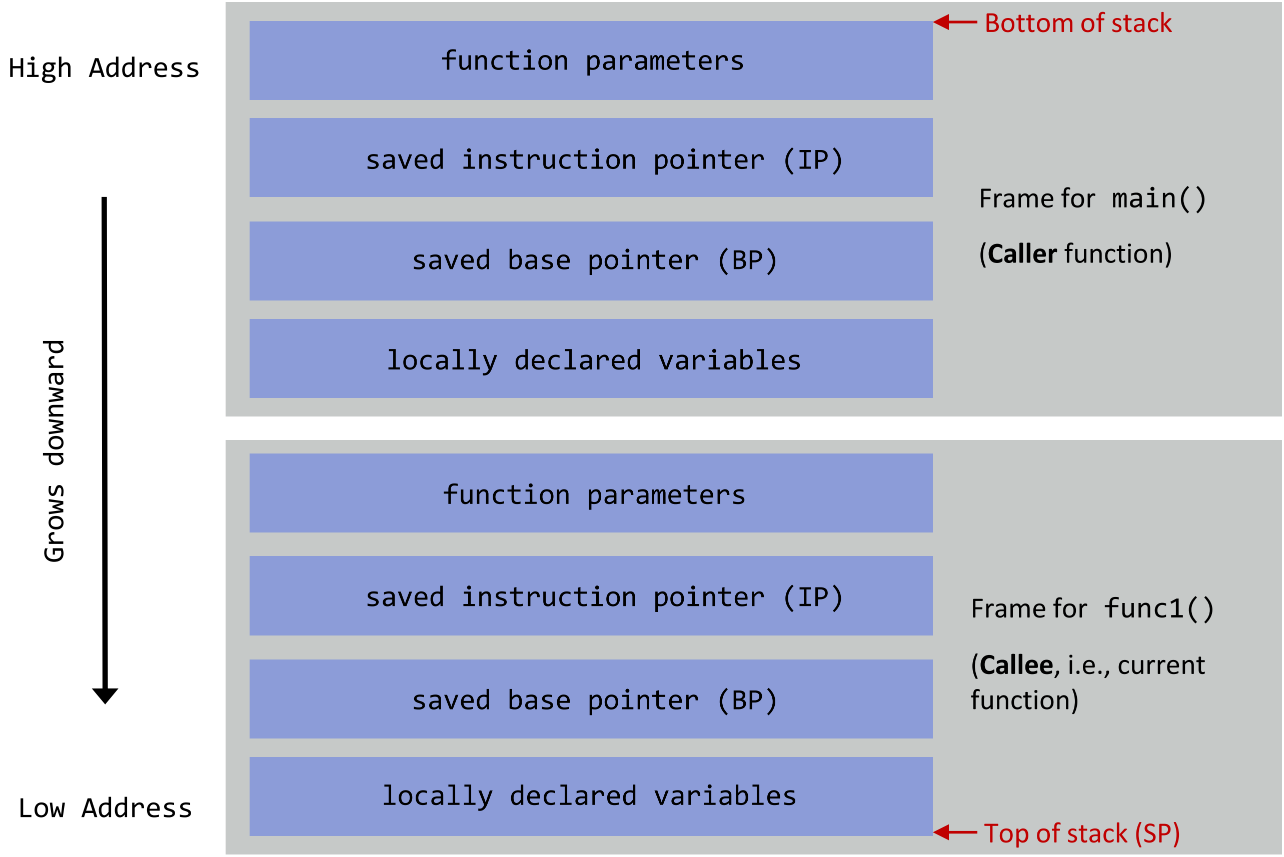
```
#include <stdio.h>
int i;
int main(void) {
  printf("hello world\n");
}
```

text	data	bss	dec	hex	filename
130	0	4	134	86	SCC131_example.o

```
#include <stdio.h>
int main(void) {
  int i=5;
  printf("hello world\n");
}
```

text	data	bss	dec	hex	filename
141	0	0	141	8d	SCC131_example.o

Stack layout



The **stack pointer** (SP) points to the lowest address of the stack for the current frame (top of stack).

The **frame pointer** or **base pointer** (FP or BP) points to a reference address of the previous frame.

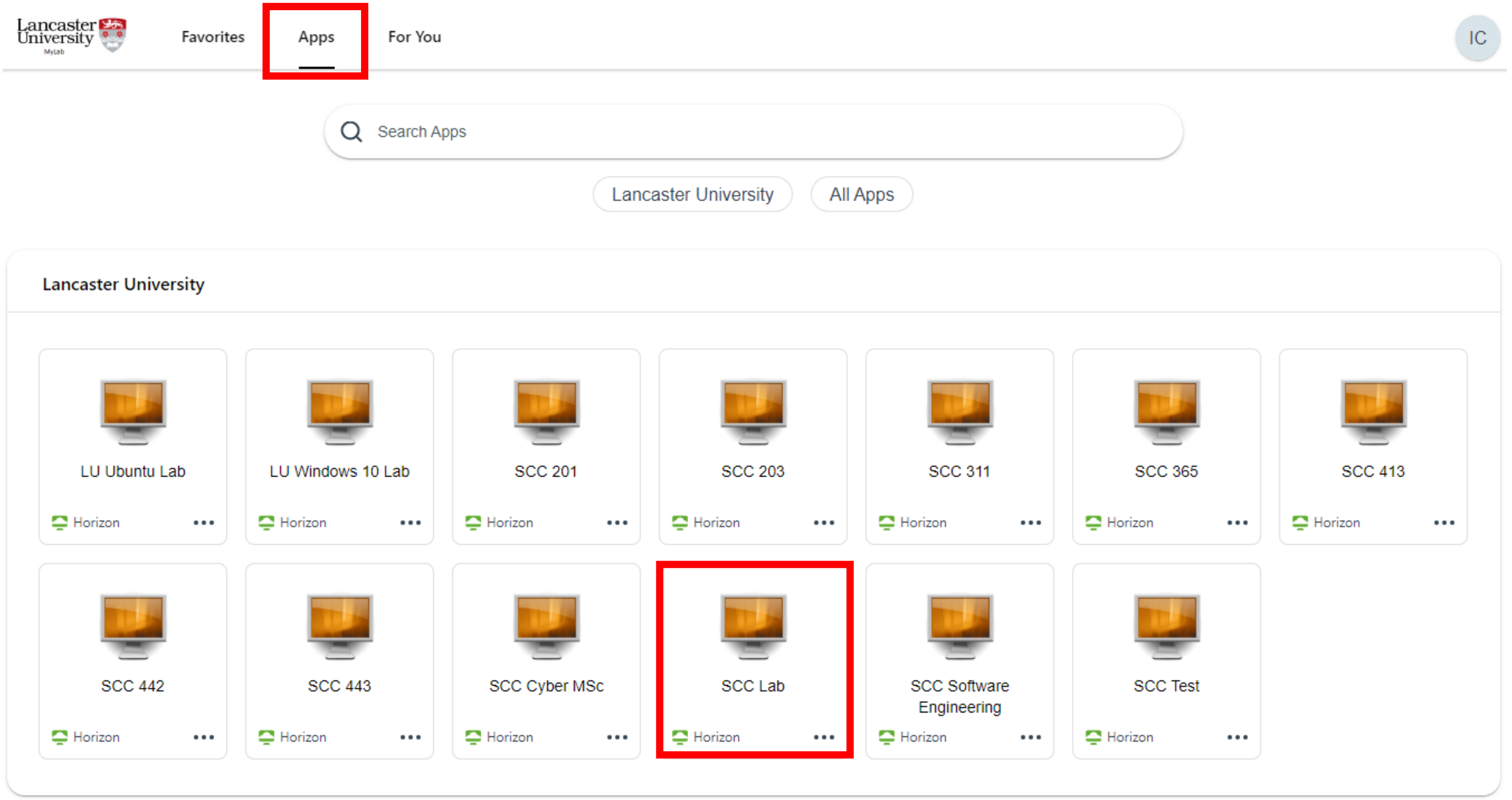
The **saved instruction pointer** (IP) points to the address that the function will return to.

Parameter values are pushed in *reverse order*, so that the first parameter will be the last to be pushed to the stack. Therefore, the first parameter will be at a *fixed offset* from the BP regardless of the number of parameters.

Step-by-step demonstration

Connect to MyLab (for guidelines, check ASK).

Select the ‘Apps’ tab and click ‘SCC Lab’ to access the appropriate virtual machine.



Step 1: Determine the CPU architecture

To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:         x86_64
  CPU op-mode(s):     32-bit, 64-bit
  Address sizes:      45 bits physical, 48 bits virtual
  Byte Order:         Little Endian
```

Step 1: Determine the CPU architecture

To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:        x86_64
  CPU op-mode(s):    32-bit, 64-bit
  Address sizes:     45 bits physical, 48 bits virtual
  Byte Order:        Little Endian
```

64-bit architecture typically used by Intel and AMD processors.

2^{64} bytes of byte-addressable memory (in theory).

Step 1: Determine the CPU architecture

To display information about the CPU architecture:

```
lscpu
```

The output will look like:

64-bit addresses have been limited to 48-bit addresses.

```
Architecture:        x86_64
  CPU op-mode(s):    32-bit, 64-bit
  Address sizes:     45 bits physical, 48 bits virtual
  Byte Order:        Little Endian
```

Step 1: Determine the CPU architecture

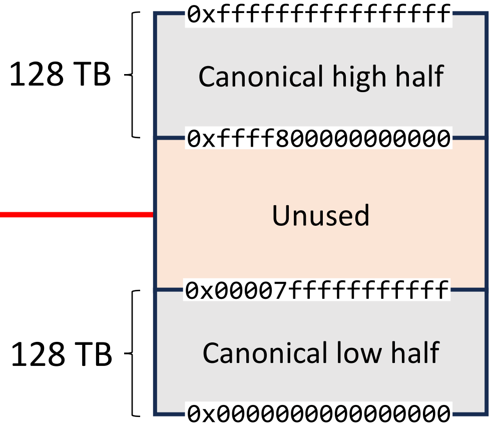
To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:          x86_64
  CPU op-mode(s):      32-bit, 64-bit
  Address sizes:       45 bits physical, 48 bits virtual
  Byte Order:          Little Endian
```

48-bit addressing using 64 bits



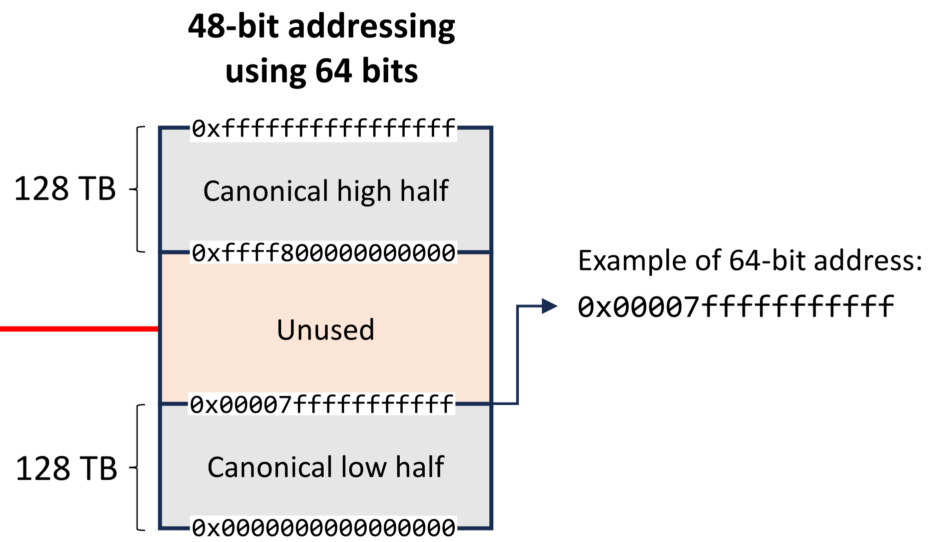
Step 1: Determine the CPU architecture

To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:        x86_64
  CPU op-mode(s):    32-bit, 64-bit
  Address sizes:     45 bits physical, 48 bits virtual
  Byte Order:        Little Endian
```



Step 1: Determine the CPU architecture

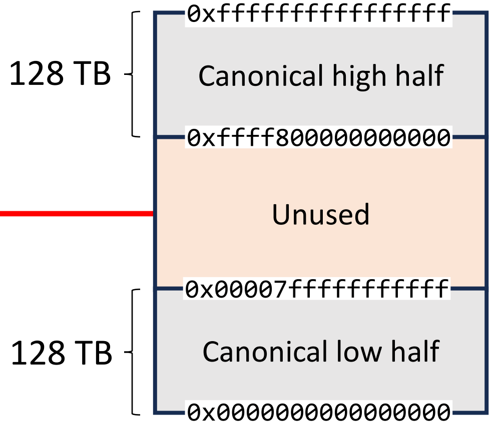
To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:          x86_64
  CPU op-mode(s):      32-bit, 64-bit
  Address sizes:       45 bits physical, 48 bits virtual
  Byte Order:          Little Endian
```

48-bit addressing using 64 bits



Prefix used to denote hexadecimal (base-16) numbers.

0x00007fffffffffff

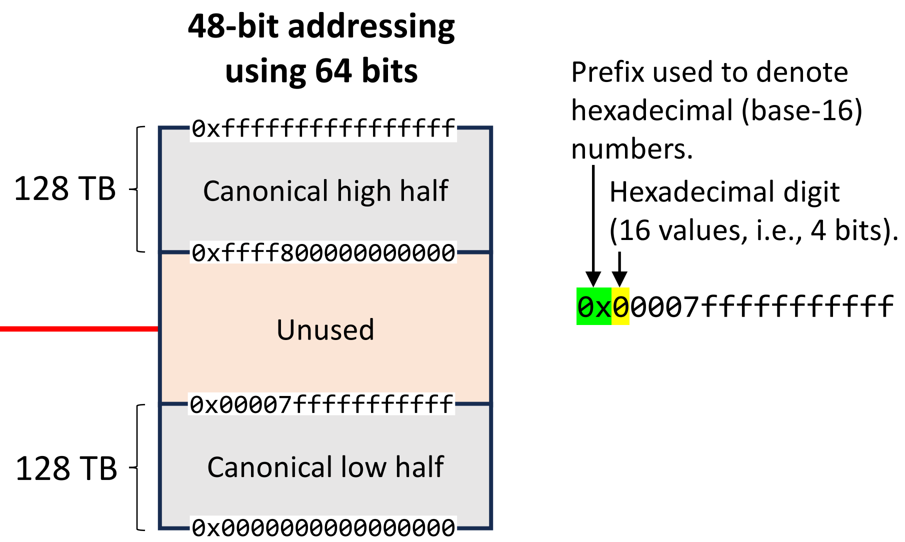
Step 1: Determine the CPU architecture

To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:          x86_64
  CPU op-mode(s):      32-bit, 64-bit
  Address sizes:       45 bits physical, 48 bits virtual
  Byte Order:          Little Endian
```



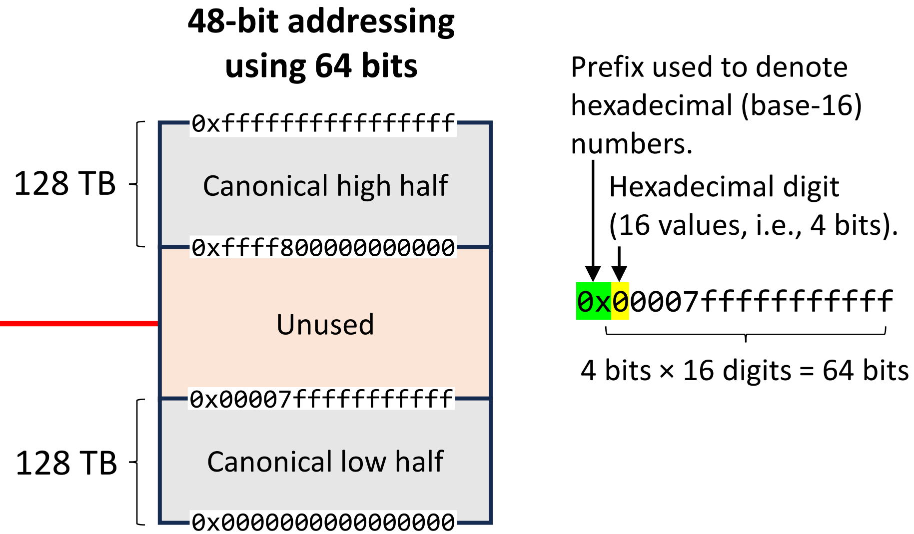
Step 1: Determine the CPU architecture

To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:        x86_64
  CPU op-mode(s):    32-bit, 64-bit
  Address sizes:     45 bits physical, 48 bits virtual
  Byte Order:        Little Endian
```



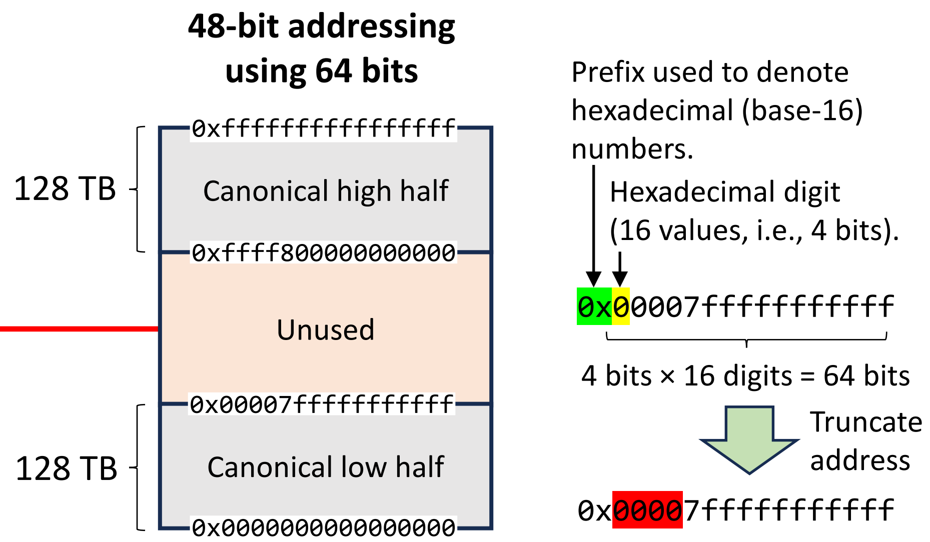
Step 1: Determine the CPU architecture

To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:        x86_64
  CPU op-mode(s):    32-bit, 64-bit
  Address sizes:     45 bits physical, 48 bits virtual
  Byte Order:        Little Endian
```



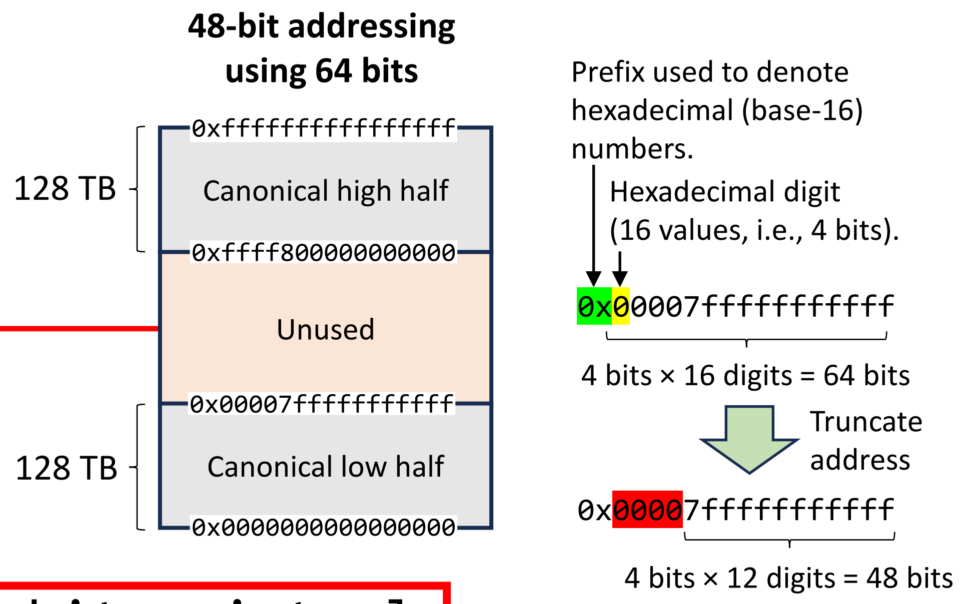
Step 1: Determine the CPU architecture

To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:          x86_64
  CPU op-mode(s):      32-bit, 64-bit
  Address sizes:       45 bits physical, 48 bits virtual
  Byte Order:          Little Endian
```



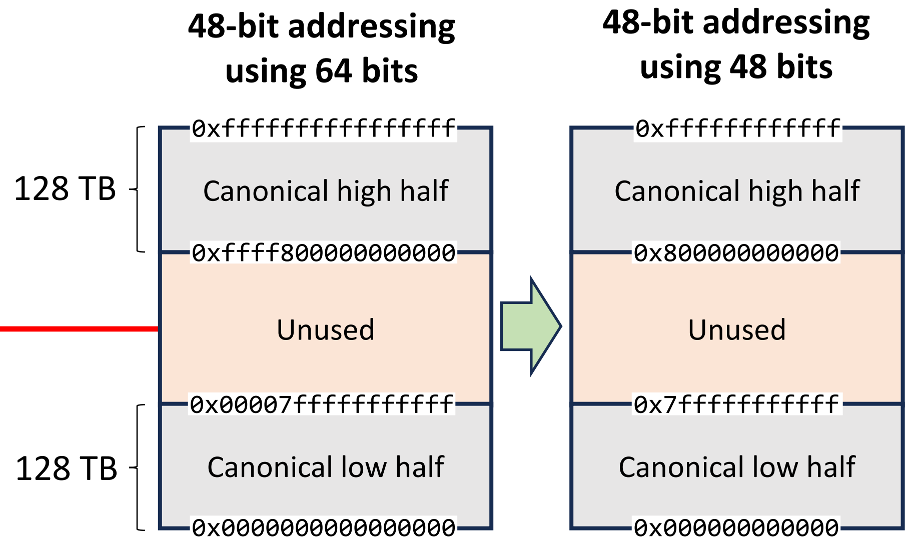
Step 1: Determine the CPU architecture

To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:          x86_64
  CPU op-mode(s):      32-bit, 64-bit
  Address sizes:       45 bits physical, 48 bits virtual
  Byte Order:          Little Endian
```



Step 1: Determine the CPU architecture

Lancaster University

To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:          x86_64
  CPU op-mode(s):      32-bit, 64-bit
  Address sizes:       45 bits physical, 48 bits virtual
  Byte Order:          Little Endian
```

Stack

0x00000ff00008	0x12
0x00000ff00007	0x34
0x00000ff00006	0x56
0x00000ff00005	0x78

Memory address of variable

Least significant **byte** first

0x12345678

32-bit integer variable

Step 1: Determine the CPU architecture

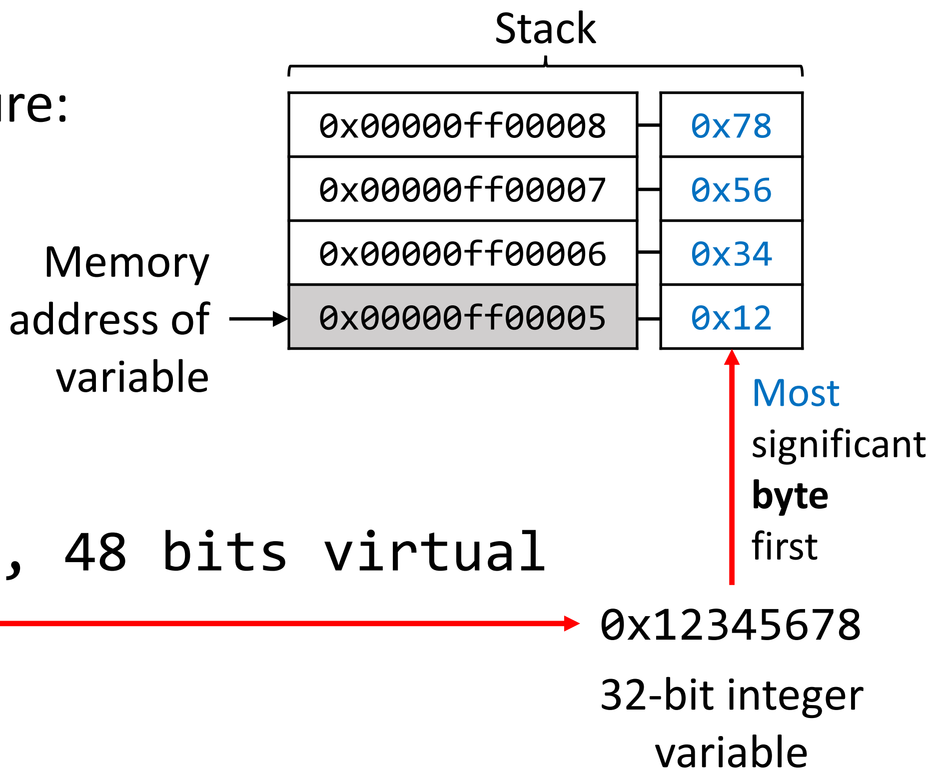
To display information about the CPU architecture:

```
lscpu
```

The output will look like:

```
Architecture:          x86_64
  CPU op-mode(s):      32-bit, 64-bit
  Address sizes:       45 bits physical, 48 bits virtual
  Byte Order:          Big Endian
```

What if?

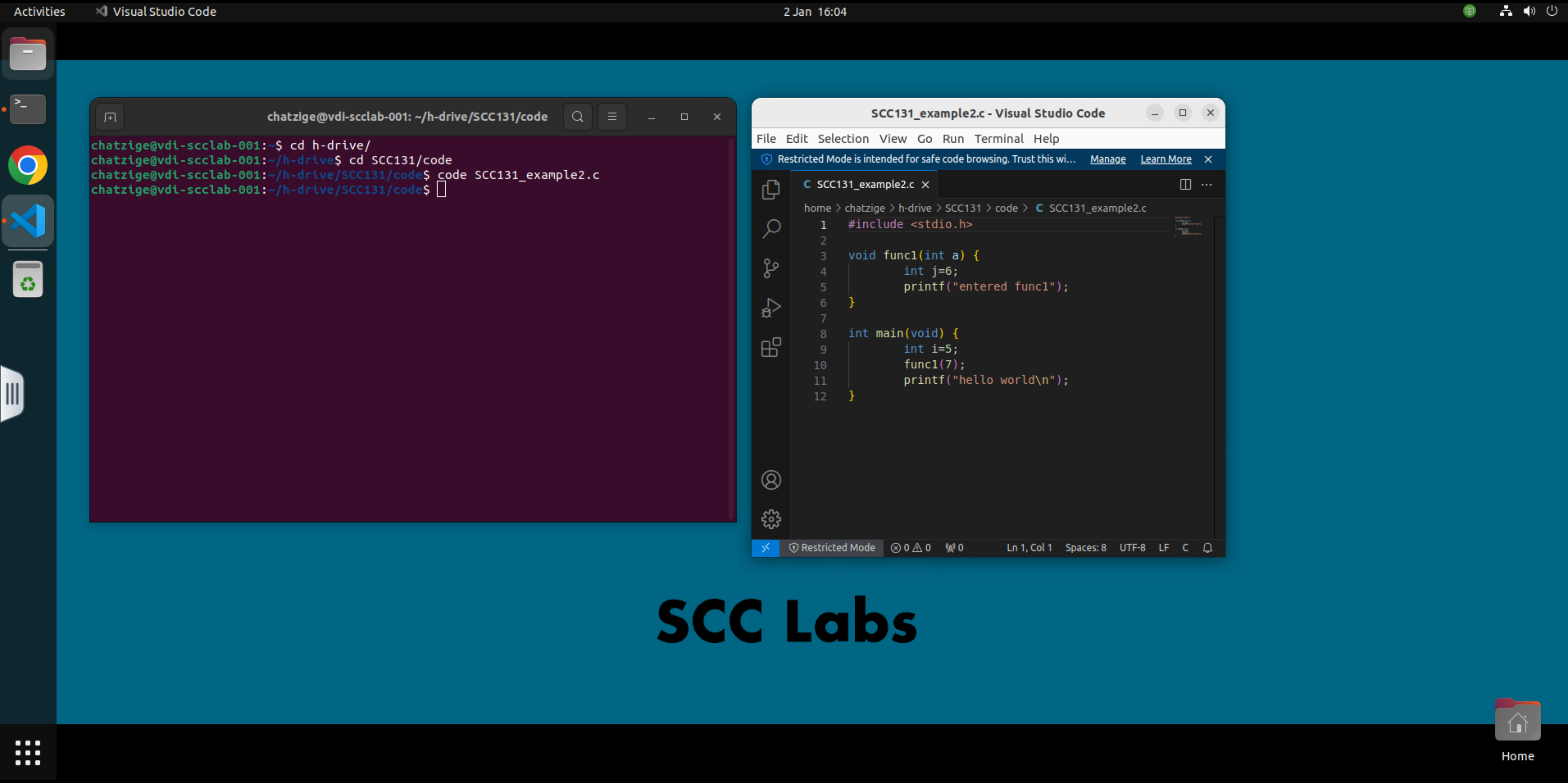


Step 2: Develop code to examine

SCC131_example2.c

```
1    #include <stdio.h>
2
3    void func1(int a) {
4      int j=6;
5      printf("entered func1");
6    }
7
8    int main(void) {
9      int i=5;
10     func1(7);
11     printf("hello world\n");
12   }
```

- This program will be examined using GDB.
- Notice that the program consists of two functions: `main()` and `func1()`.
- Function `main()` is the caller and `func1()` is the callee.
- Both functions contain local variables.
- Breakpoints will be introduced in **lines 5 and 11**, and the stack frames will be displayed.


Activities
Visual Studio Code
2 Jan 16:04
chatzige@vdi-scclab-001: ~/h-drive/SCC131/code
chatzige@vdi-scclab-001:~$ cd h-drive/
chatzige@vdi-scclab-001:~/h-drive$ cd SCC131/code
chatzige@vdi-scclab-001:~/h-drive/SCC131/code$ code SCC131_example2.c
chatzige@vdi-scclab-001:~/h-drive/SCC131/code$
SCC131_example2.c - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Restricted Mode is intended for safe code browsing. Trust this wi...
Manage
Learn More
SCC131_example2.c
home > chatzige > h-drive > SCC131 > code > SCC131_example2.c
#include <stdio.h>
void func1(int a) {
int j=6;
printf("entered func1");
}
int main(void) {
int i=5;
func1(7);
printf("hello world\n");
}
Restricted Mode
Ln 1, Col 1
Spaces: 8
UTF-8
LF
C
SCC Labs
Home

Step 3: Compile and run code using GDB

```
gcc -g SCC131_example2.c -o SCC131_example2
```

Generate debug information (-g) to be used by GDB and create executable file (-o).

Step 3: Compile and run code using GDB

```
gcc -g SCC131_example2.c -o SCC131_example2
gdb SCC131_example2
(gdb) █
```

Debug the executable file from the command line.

Step 3: Compile and run code using GDB

```
gcc -g SCC131_example2.c -o SCC131_example2
gdb SCC131_example2
(gdb) break 5
Breakpoint 1 at 0x117c: file SCC131_example2.c, line 5.
(gdb) break 11
Breakpoint 2 at 0x11b0: file SCC131_example2.c, line 11.
(gdb) 
```

Introduce breakpoints in lines 5 and 11.

Step 3: Compile and run code using GDB

```
gcc -g SCC131_example2.c -o SCC131_example2
gdb SCC131_example2
(gdb) break 5
Breakpoint 1 at 0x117f: file SCC131_example2.c, line 5.
(gdb) break 11
Breakpoint 2 at 0x11b3: file SCC131_example2.c, line 11.
(gdb) run
Starting program: /home/chatzige/h-drive/SCC131/code/SCC131_example2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Breakpoint 1, func1 (a=7) at SCC131_example2.c:5
5               printf("entered func1");
(gdb) 
```

Run the executable file until the first breakpoint (in line 5) is reached.

Step 3: Examine the stack using GDB

```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_example2.c:10
```

Display the call stack (same as `backtrace`)

Step 3: Examine the stack using GDB

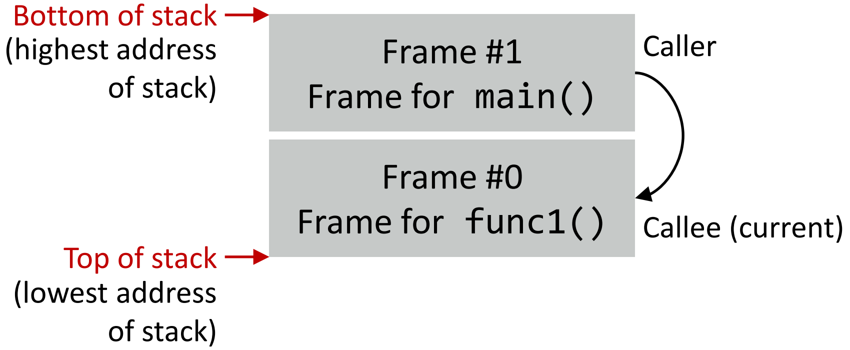
```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_example2.c:10
```

```
1   #include <stdio.h>
2
3   void func1(int a) {
4     int j=6;
5     printf("entered func1");
6   }
7
8   int main(void) {
9     int i=5;
10    func1(7);
11    printf("hello world\n");
12  }
```

Step 3: Examine the stack using GDB

```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_example2.c:10
```

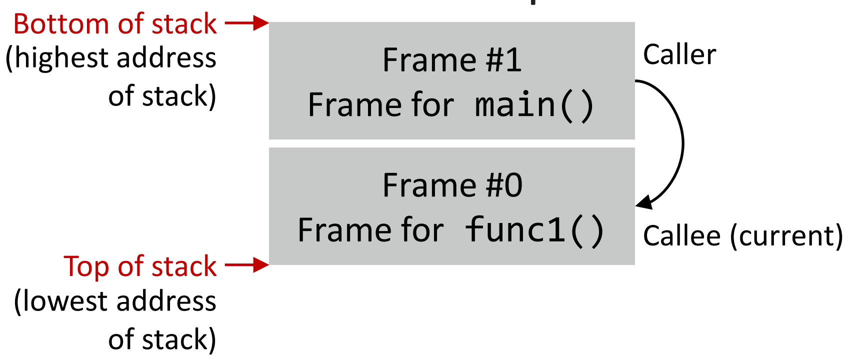
In this example:



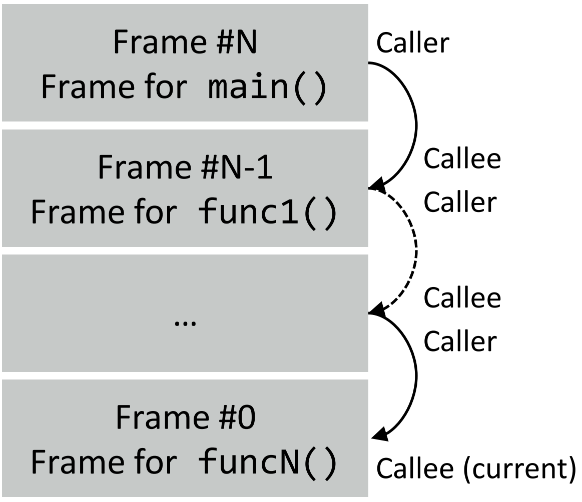
Step 3: Examine the stack using GDB

```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_example2.c:10
```

In this example:



If we had N nested functions:



Frame #0 always denotes the **current** frame.

The indices of frames are updated whenever execution of a function is completed.

Step 3: Examine the stack using GDB

```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_example2.c:10
(gdb) info frame 0
Stack frame at 0x7fffffffdf60:
 rip = 0x55555555517f in func1 (SCC131_example2.c:5); saved rip = 0x5555555551b3
 called by frame at 0x7fffffffdf80
 source language c.
 Arglist at 0x7fffffffdf50, args: a=7
 Locals at 0x7fffffffdf50, Previous frame's sp is 0x7fffffffdf60
 Saved registers:
  rbp at 0x7fffffffdf50, rip at 0x7fffffffdf58
```

Display information about stack frame 0, which corresponds to func1.

Step 3: Examine the stack using GDB

```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_exa
(gdb) info frame 0
Stack frame at 0x7fffffffdf60:
 rip = 0x55555555517f in func1 (SCC131_example2
 called by frame at 0x7fffffffdf80
 source language c.
 Arglist at 0x7fffffffdf50, args: a=7
 Locals at 0x7fffffffdf50, Previous frame's sp is 0x7fffffffdf60
 Saved registers:
  rbp at 0x7fffffffdf50, rip at 0x7fffffffdf58
```

Address that signifies the beginning of this frame (frame 0) and the end of the previous frame (frame 1). When execution of `func1` is completed, this frame will be popped from the stack and this address will become the new stack pointer.

Step 3: Examine the stack using GDB

```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_example2.c:10
(gdb) info frame 0
Stack frame at 0x7fffffffdf60:
 rip = 0x55555555517f in func1 (SCC131_example2.c:5); saved rip = 0x5555555551b3
 called by frame at 0x7fffffffdf80
 source language c.
 Arglist at 0x7fffffffdf50, args: a=7
 Locals at 0x7fffffffdf50, Previous frame's sp is 0x
 Saved registers:
  rbp at 0x7fffffffdf50, rip at 0x7fffffffdf58
```

Address of the *next* instruction pointer, i.e., the address that the frame will return to when operation of func1 resumes (line 5 in func1).

Step 3: Examine the stack using GDB

```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_example2.c:10
(gdb) info frame 0
Stack frame at 0x7fffffffdf60:
 rip = 0x55555555517f in func1 (SCC131_example2.c:5); saved rip = 0x5555555551b3
 called by frame at 0x7fffffffdf80
 source language c.
 Arglist at 0x7fffffffdf50, args: a=7
 Locals at 0x7fffffffdf50, Previous frame's sp is 0
 Saved registers:
  rbp at 0x7fffffffdf50, rip at 0x7fffffffdf58
```

Address (and location in the stack) that this frame will return to at the end of `func1`, when control returns to the previous frame (immediately *after* the call of `func1` in line 10).

Step 3: Examine the stack using GDB

```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_example2.c:10
(gdb) info frame 0
Stack frame at 0x7fffffffdf60:
 rip = 0x55555555517f in func1 (SCC131_example2.c:5); saved rip = 0x5555555551b3
 called by frame at 0x7fffffffdf80
 source language c.
 Arglist at 0x7fffffffdf50, args: a=7
 Locals at 0x7fffffffdf50, Previous frame's sp is 0x7fffffffdf60
 Saved registers:
  rbp at 0x7fffffffdf50, rip at 0x7fffffffdf58
```

Address of the frame that called this frame, i.e., address of frame 1 (the caller).

Step 3: Examine the stack using GDB

```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_example2.c:10
(gdb) info frame 0
Stack frame at 0x7fffffffdf60:
 rip = 0x55555555517f in func1 (SCC131_example2.c:5); saved rip = 0x5555555551b3
 called by frame at 0x7fffffffdf80
 source language c.
 Arglist at 0x7fffffffdf50, args: a=7
 Locals at 0x7fffffffdf50, Previous frame's sp is 0x7ff
 Saved registers:
  rbp at 0x7fffffffdf50, rip at 0x7fffffffdf58
```

Location of the base pointer (reference for this frame that points to the base pointer of the previous frame).

Step 3: Examine the stack using GDB

```
(gdb) info stack
#0  func1 (a=7) at SCC131_example2.c:5
#1  0x00005555555551b3 in main () at SCC131_example2.c:10
(gdb) info frame 1
Stack frame at 0x7fffffffdf80:
 rip = 0x5555555551b3 in main (SCC131_example2.c:10); saved rip = 0x7ffff7c29d90
 caller of frame at 0x7fffffffdf60
 source language c.
 Arglist at 0x7fffffffdf70, args:
 Locals at 0x7fffffffdf70, Previous frame's sp is 0x7fffffffdf80
 Saved registers:
  rbp at 0x7fffffffdf70, rip at 0x7fffffffdf78
```

Display information about stack frame 1, which corresponds to main.

Step 4: Visualise the stack

```
(gdb) x/28xw $sp
0x7fffffffdf30:    0x00000000    0x00000000    0x00000000    0x00000007
0x7fffffffdf40:    0x00000000    0x00000000    0x00000000    0x00000006
0x7fffffffdf50:    0xffffdf70    0x00007fff    0x555551b3    0x00005555
0x7fffffffdf60:    0x00000000    0x00000000    0x00000000    0x00000005
0x7fffffffdf70:    0x00000001    0x00000000    0xf7c29d90    0x00007fff
0x7fffffffdf80:    0x00000000    0x00000000    0x55555196    0x00005555
```

8 digits = 32 bits = 4 bytes at `0x7fffffffdf80`

4 bytes at `0x7fffffffdf84`

4 bytes at `0x7fffffffdf88`

4 bytes at `0x7fffffffdf8c`

Look at the memory and display (`x`) the 28 words (`w`) [note: a *word* contains 32 bits] in hexadecimal format (`x`) located just below the top of the stack, specified by the stack pointer (`$sp`).

Step 4: Visualise the stack (manually)

```
0x7fffffffdf84: 0x00000000
0x7fffffffdf80: 0x00000000 -------------- Beginning of Frame 1 (main) --------------
0x7fffffffdf7c: 0x00007fff } rip at 0x7fffffffdf78 is 0x00007ffff7c29d90 (saved IP)
0x7fffffffdf78: 0xf7c29d90
0x7fffffffdf74: 0x00000000 } rbp at 0x7fffffffdf70 is 0x0000000000000001 (saved BP)
0x7fffffffdf70: 0x00000001
0x7fffffffdf6c: 0x00000005 } i=5 (locally declared variable)
0x7fffffffdf68: 0x00000000
0x7fffffffdf64: 0x00000000
0x7fffffffdf60: 0x00000000 -------------- Beginning of Frame 0 (func1) -------------
0x7fffffffdf5c: 0x00005555 } rip at 0x7fffffffdf58 is 0x00005555555551b3 (saved IP)
0x7fffffffdf58: 0x555551b3
0x7fffffffdf54: 0x00007fff } rbp at 0x7fffffffdf50 is 0x00007fffffffdf70 (saved BP)
0x7fffffffdf50: 0xffffdf70
0x7fffffffdf4c: 0x00000006 } j=6 (locally declared variable)
0x7fffffffdf48: 0x00000000
0x7fffffffdf44: 0x00000000
0x7fffffffdf40: 0x00000000
0x7fffffffdf3c: 0x00000007
0x7fffffffdf38: 0x00000000
0x7fffffffdf34: 0x00000000
0x7fffffffdf30: 0x00000000 } sp at 0x7fffffffdf30 (top of stack)
```

SCC131_example2.c

```c
#include <stdio.h>

void func1(int a) {
  int j=6;
  printf("entered func1");
}

int main(void) {
  int i=5;
  func1(7);
  printf("hello world\n");
}
```

Step 4: Visualise the stack (manually)

```
0x7fffffffdf84: 0x00000000
0x7fffffffdf80: 0x00000000 -------------- Beginning of Frame 1 (main) --------------
0x7fffffffdf7c: 0x00007fff }
0x7fffffffdf78: 0xf7c29d90 } rip at 0x7fffffffdf78 is 0x00007ffff7c29d90 (saved IP)
0x7fffffffdf74: 0x00000000 }
0x7fffffffdf70: 0x00000001 } rbp at 0x7fffffffdf70 is 0x0000000000000001 (saved BP)
0x7fffffffdf6c: 0x00000005 } i=5 (locally declared variable)
0x7fffffffdf68: 0x00000000
0x7fffffffdf64: 0x00000000
0x7fffffffdf60: 0x00000000 -------------- Beginning of Frame 0 (func1) -------------
0x7fffffffdf5c: 0x00005555 }
0x7fffffffdf58: 0x555551b3 } rip at 0x7fffffffdf58 is 0x00005555555551b3 (saved IP)
0x7fffffffdf54: 0x00007fff }
0x7fffffffdf50: 0xffffdf70 } rbp at 0x7fffffffdf50 is 0x00007fffffffdf70 (saved BP)
0x7fffffffdf4c: 0x00000006 } j=6 (locally declared variable)
0x7fffffffdf48: 0x00000000
0x7fffffffdf44: 0x00000000
0x7fffffffdf40: 0x00000000
0x7fffffffdf3c: 0x00000007 } a=7 (function parameter) <---- Why is it here? ------ Shouldn't it be here?
0x7fffffffdf38: 0x00000000
0x7fffffffdf34: 0x00000000
0x7fffffffdf30: 0x00000000 } sp at 0x7fffffffdf30 (top of stack)
```

Step 5: Check the registers

- In the **x86** architecture, all function parameters appear at the beginning of the relevant stack frame.
- In the **x86-64** architecture, often abbreviated as **x64**, the **first six function parameters** are passed in registers and treated as local variables. The **remaining** function parameters appear at the beginning of the stack frame.
- Type `info reg` to print the content of the registers and notice that value 7 has been stored in register `%rdi`.

Register	Purpose
`%rax`	temp register; return value
`%rbx`	callee-saved
`%rcx`	used to pass 4th argument to functions
`%rdx`	used to pass 3rd argument to functions
`%rsp`	stack pointer
`%rbp`	callee-saved; base pointer
`%rsi`	used to pass 2nd argument to functions
`%rdi`	used to pass 1st argument to functions
`%r8`	used to pass 5th argument to functions
`%r9`	used to pass 6th argument to functions
`%r10-r11`	temporary
`%r12-r15`	callee-saved registers

Step 6: Disassemble the code (func1)

```
(gdb) disass func1
Dump of assembler code for function func1:
   0x0000555555555169 <+0>:   endbr64
   0x000055555555516d <+4>:   push   %rbp
   0x000055555555516e <+5>:   mov    %rsp,%rbp
   0x0000555555555171 <+8>:   sub    $0x20,%rsp
   0x0000555555555175 <+12>:  mov    %edi,-0x14(%rbp)
   0x0000555555555178 <+15>:  movl   $0x6,-0x4(%rbp)
=> 0x000055555555517f <+22>:  lea    0xe7e(%rip),%rax    # 0x555555556004
   0x0000555555555186 <+29>:  mov    %rax,%rdi
   0x0000555555555189 <+32>:  mov    $0x0,%eax
   0x000055555555518e <+37>:  call   0x555555555070 <printf@plt>
   0x0000555555555193 <+42>:  nop
   0x0000555555555194 <+43>:  leave
   0x0000555555555195 <+44>:  ret
End of assembler dump.
```

Recall:

```
(gdb) info frame 0
Stack frame at 0x7fffffffdf60:
 rip = 0x55555555517f in func1 (SCC131_example2.c:5);
```

Step 6: Disassemble the code (main)

```
(gdb) disass main
Dump of assembler code for function main:
   0x0000555555555196 <+0>:    endbr64
   0x000055555555519a <+4>:    push   %rbp
   0x000055555555519b <+5>:    mov    %rsp,%rbp
   0x000055555555519e <+8>:    sub    $0x10,%rsp
   0x00005555555551a2 <+12>:   movl   $0x5,-0x4(%rbp)
   0x00005555555551a9 <+19>:   mov    $0x7,%edi
   0x00005555555551ae <+24>:   call   0x555555555169 <func1>
   0x00005555555551b3 <+29>:   lea    0xe58(%rip),%rax    # 0x555555556012
   0x00005555555551ba <+36>:   mov    %rax,%rdi
   0x00005555555551bd <+39>:   call   0x555555555060 <puts@plt>
   0x00005555555551c2 <+44>:   mov    $0x0,%eax
   0x00005555555551c7 <+49>:   leave
   0x00005555555551c8 <+50>:   ret
End of assembler dump.
```

Recall:

```
(gdb) info frame 1
Stack frame at 0x7fffffffdf80:
 rip = 0x5555555551b3 in main (SCC131_example2.c:10);
```

Step 6: Disassemble the code (`main`)

```
(gdb) disass main
Dump of assembler code for function main:
   0x0000555555555196 <+0>:    endbr64
   0x000055555555519a <+4>:    push   %rbp
   0x000055555555519b <+5>:    mov    %rsp,%rbp
   0x000055555555519e <+8>:    sub    $0x10,%rsp
   0x00005555555551a2 <+12>:   movl   $0x5,-0x4(%rbp)
   0x00005555555551a9 <+19>:   mov    $0x7,%edi
   0x00005555555551ae <+24>:   call   0x555555555169 <func1>
   0x00005555555551b3 <+29>:   lea    0xe58(%rip),%rax    # 0x555555556012
   0x00005555555551ba <+36>:   mov    %rax,%rdi
   0x00005555555551bd <+39>:   call   0x555555555060 <puts@plt>
   0x00005555555551c2 <+44>:   mov    $0x0,%eax
   0x00005555555551c7 <+49>:   leave
   0x00005555555551c8 <+50>:   ret
End of assembler dump.
```

Evidence that value 7 has been moved to register `%rdi` – this is the input parameter to `func1`.
Note that `%rdi` is a 64-bit register. We use `%edi` to access the 32 least significant bits of `%rdi`, given that 7 has been stored as a 32-bit integer.

Memory layout – Bringing it all together

SCC131_example3.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int i = 5;
5
6   int func1(int a)
7   {
8       int k = 5;
9   }
10
11  int main()
12  {
13      char * buffer;
14      buffer = (char*)malloc(i+1);
15      if (buffer == NULL) exit(1);
16      func1(i);
17      return 0;
18  }
```

```
(gdb) break 9
Breakpoint 1 at 0x117b: file SCC131_example3.c, line 9.
(gdb) run
Starting program: /home/chatzige/h-drive/SCC131/code/SCC131_example3
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-
gnu/libthread_db.so.1".
Breakpoint 1, func1 (a=5) at SCC131_example3.c:9
9           }
(gdb) print &k
$1 = (int *) 0x7fffffffdf3c
(gdb) print &i
$2 = (int *) 0x555555558010 <i>
(gdb) frame 1
#1  0x00005555555551bf in main () at SCC131_example3.c:16
16                  func1(i);
(gdb) print buffer
$3 = 0x5555555592a0 ""
```

Memory layout – Bringing it all together

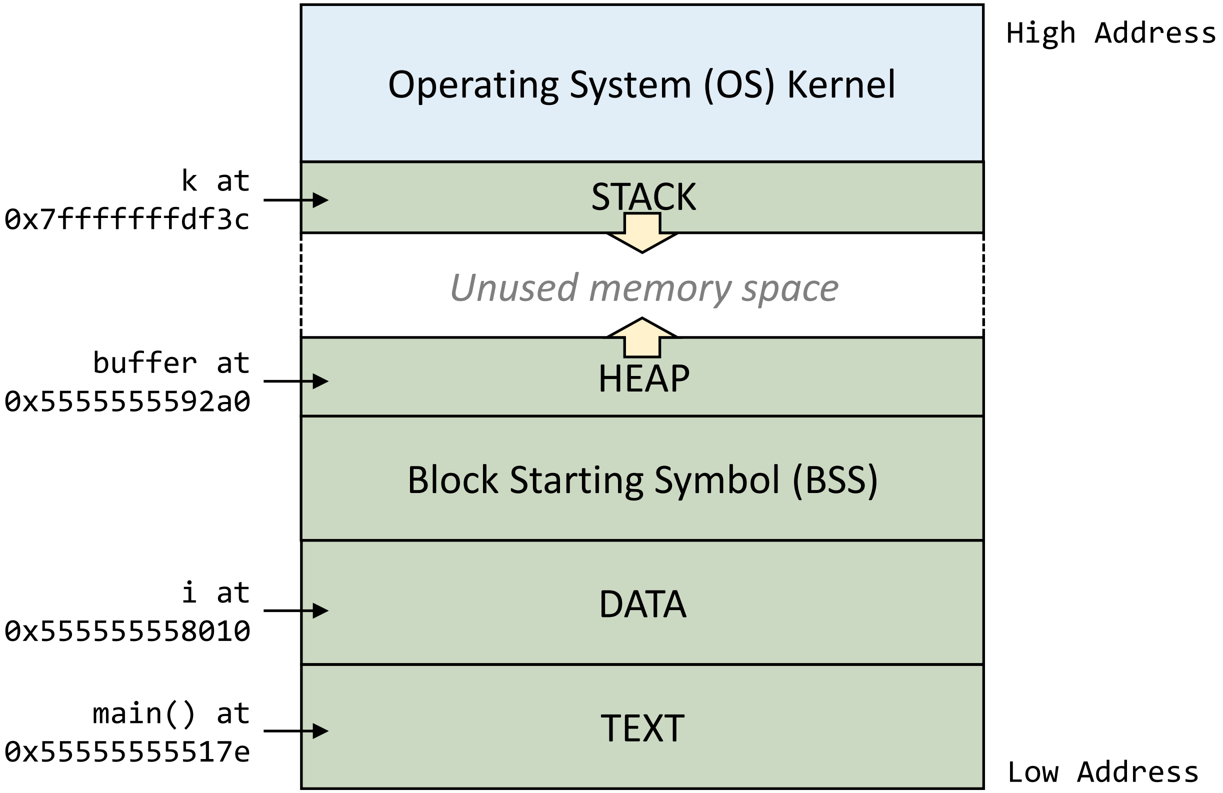
SCC131_example3.c

```
#include <stdio.h>
#include <stdlib.h>

int i = 5;

int func1(int a)
{
    int k = 5;
}

int main()
{
    char * buffer;
    buffer = (char*)malloc(i+1);
    if (buffer == NULL) exit(1);
    func1(i);
    return 0;
}
```



Confused?

Unfortunately, memory layout is anything but simple:

- Depends on architecture (Intel, AMD, ARM, ...)
- Depends on OS (macOS, Linux, Windows, ...)
- Depends on compiler and compiler version
- Depends on compiler flags

Summary

We focused on the memory layout of the x86-64 architecture and discussed about:

- The memory management unit, the translation lookaside buffer and the relationship between physical and virtual addresses.
- The main memory regions: Text, Data, BSS, Heap, Stack and OS Kernel.
- The stack layout: the base pointer, the instruction pointer, the stack pointer and the memory regions for function parameters and locally declared variables.
- GDB commands that can help you examine and visualise the stack of a running C program.

Resources

- MIT: Procedures and stacks
- MIT: Examining the stack
- MIT: X86-64 Architecture guide
- Stanford University: Runtime stack (Lab 7)
- Brown University: x64 cheat sheet (page 1)
- The Linux Kernel: Complete virtual memory map with 4-level page tables
- Stack frame layout on x86-64 by Eli Bendersky
- Visual GDB: GDB command reference