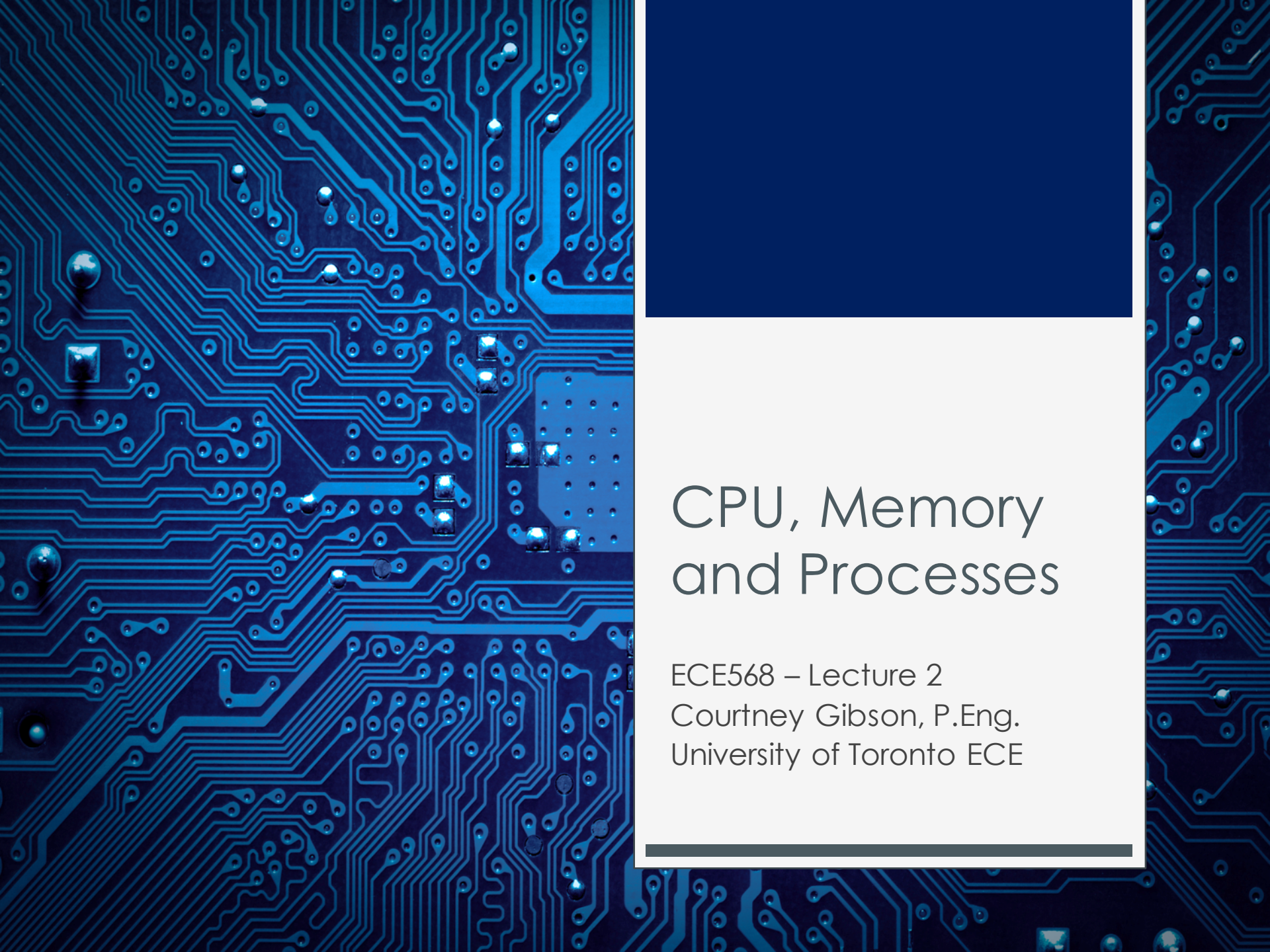


CPU, Memory and Processes

ECE568 – Lecture 2

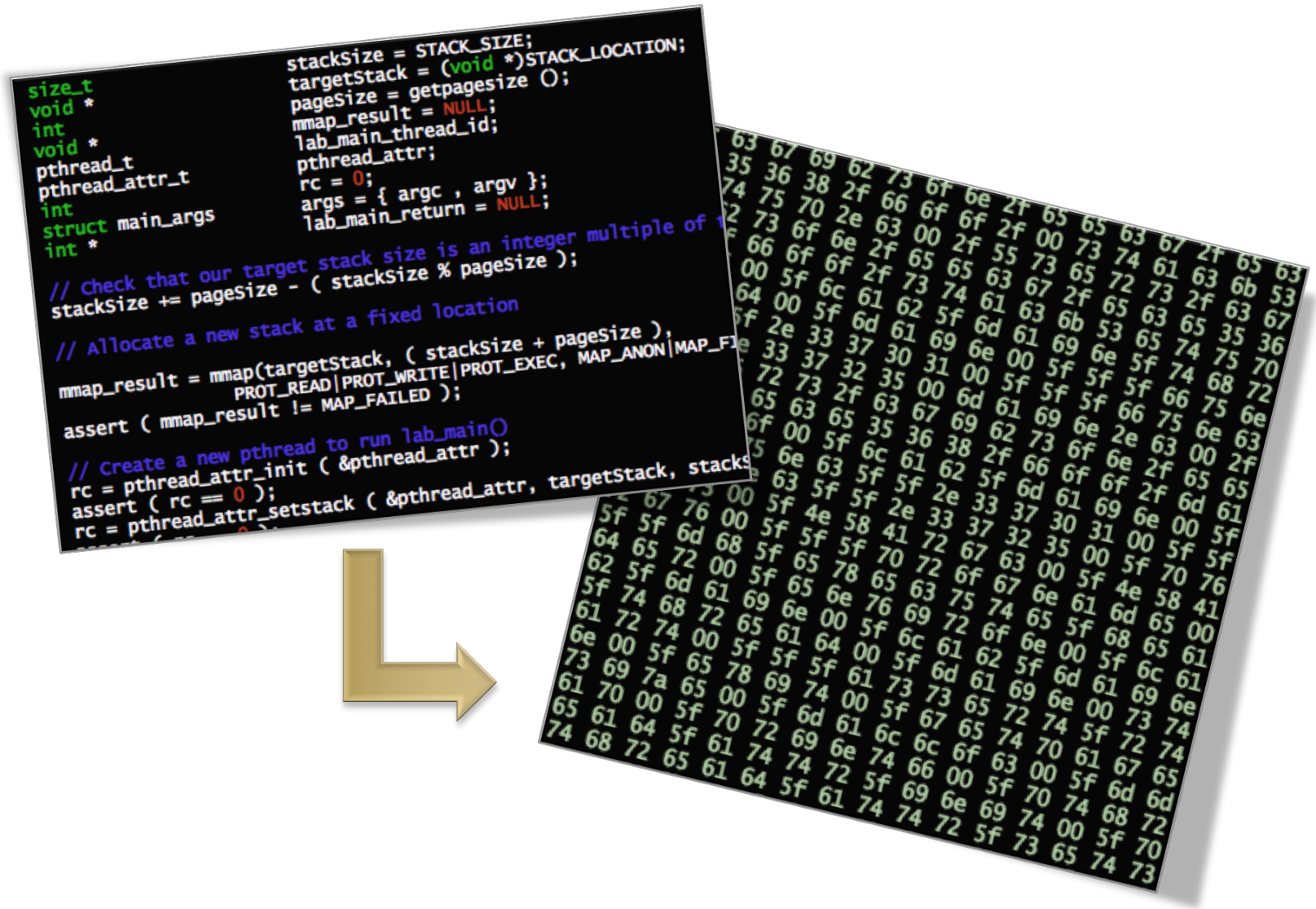
Courtney Gibson, P.Eng.
University of Toronto ECE

Memory

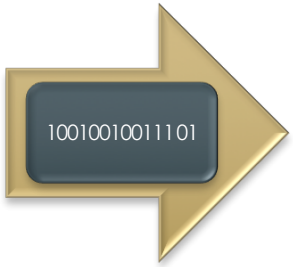
Data Representation

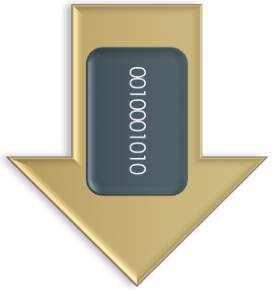
Data Representation

As developers, our primary job is to manipulate numbers.


size_t stackSize = STACK_SIZE;
void * targetStack = (void *)STACK_LOCATION;
int pageSize = getpagesize ();
void * mmap_result = NULL;
pthread_t lab_main_thread_id;
pthread_attr_t pthread_attr;
int rc = 0;
struct main_args args = { argc , argv };
int * lab_main_return = NULL;
// Check that our target stack size is an integer multiple of t
stackSize += pageSize - (stackSize % pageSize);
// Allocate a new stack at a fixed location
mmap_result = mmap(targetStack, (stackSize + pageSize),
PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANON|MAP_FI
assert (mmap_result != MAP_FAILED);
// Create a new pthread to run lab_main()
rc = pthread_attr_init (&pthread_attr);
assert (rc == 0);
rc = pthread_attr_setstack (&pthread_attr, targetStack, stacks

```
size_t          stackSize = STACK_SIZE;
void *          targetStack = (void *)STACK_LOCATION;
int             pageSize = getpagesize ();
void *          mmap_result = NULL;
pthread_t       lab_main_thread_id;
pthread_attr_t  pthread_attr;
int             rc = 0;
struct main_args args = { argc , argv };
int *           lab_main_return = NULL;

// Check that our target stack size is an integer multiple of t
stackSize += pageSize - ( stackSize % pageSize );

// Allocate a new stack at a fixed location

mmap_result = mmap(targetStack, ( stackSize + pageSize ),
                   PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANON|MAP_FI
assert ( mmap_result != MAP_FAILED );

// Create a new pthread to run lab_main()
rc = pthread_attr_init ( &pthread_attr );
assert ( rc == 0 );
rc = pthread_attr_setstack ( &pthread_attr, targetStack, stacks
```





Data Representation

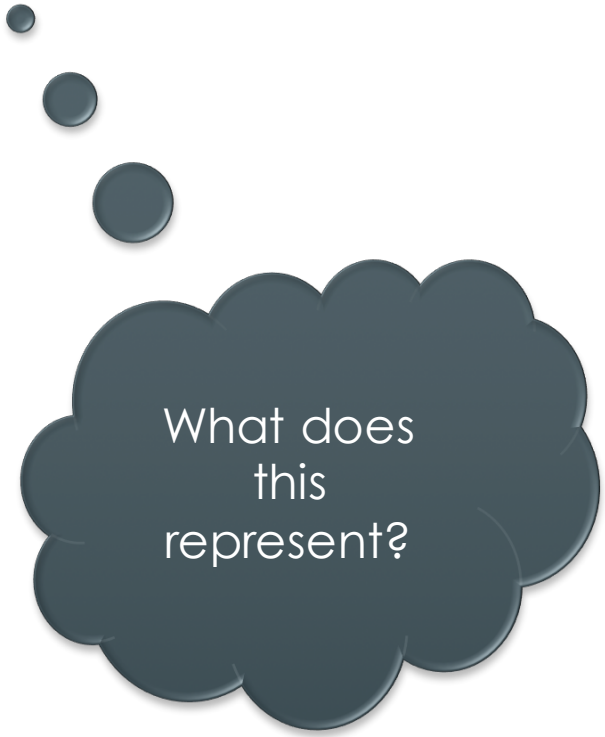
Because everything (instructions, input and output) is represented as numbers, we need to be careful.

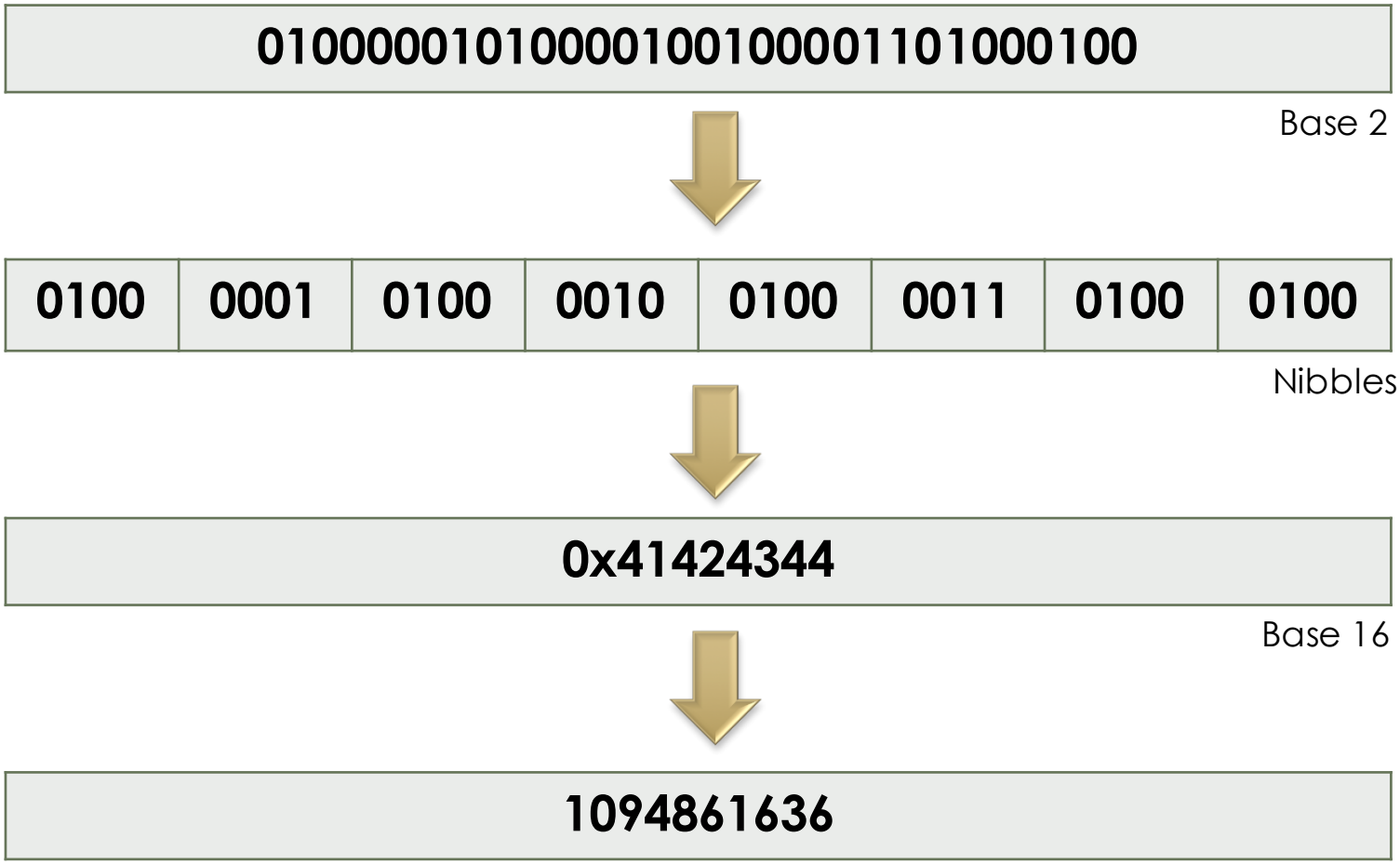
Try to avoid thinking of memory only in terms of how you define your variables:

```
int x = 1145258561;
```

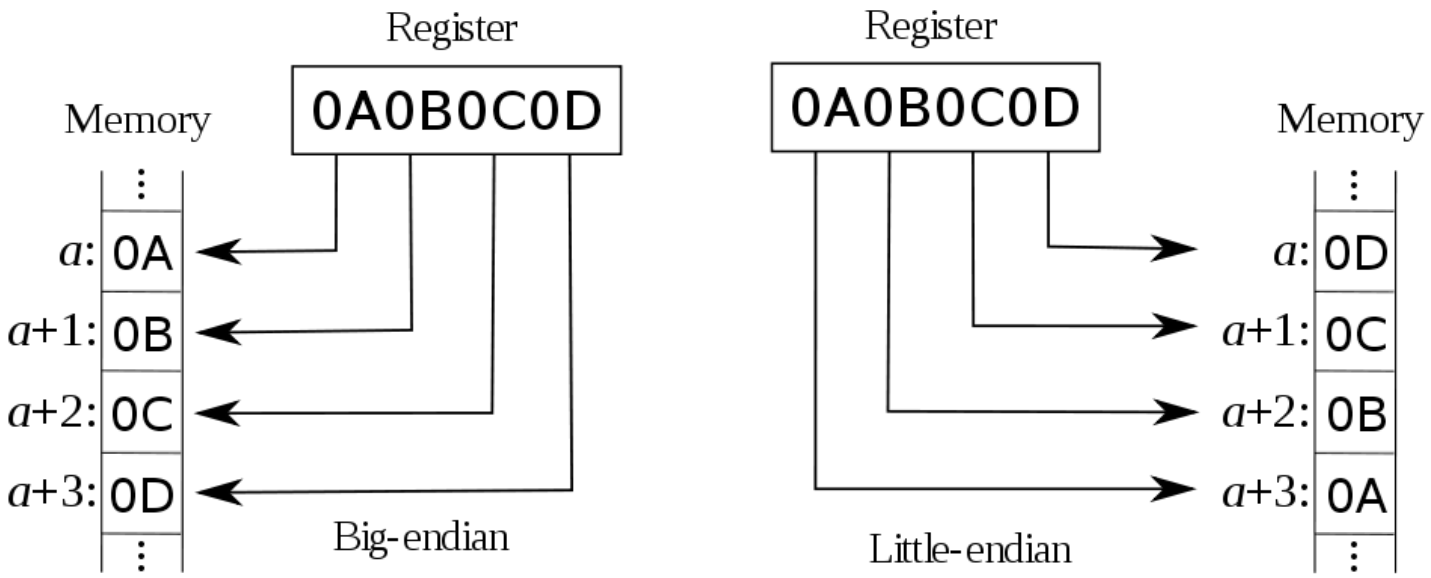
01000001010000100100001101000100

Base 2

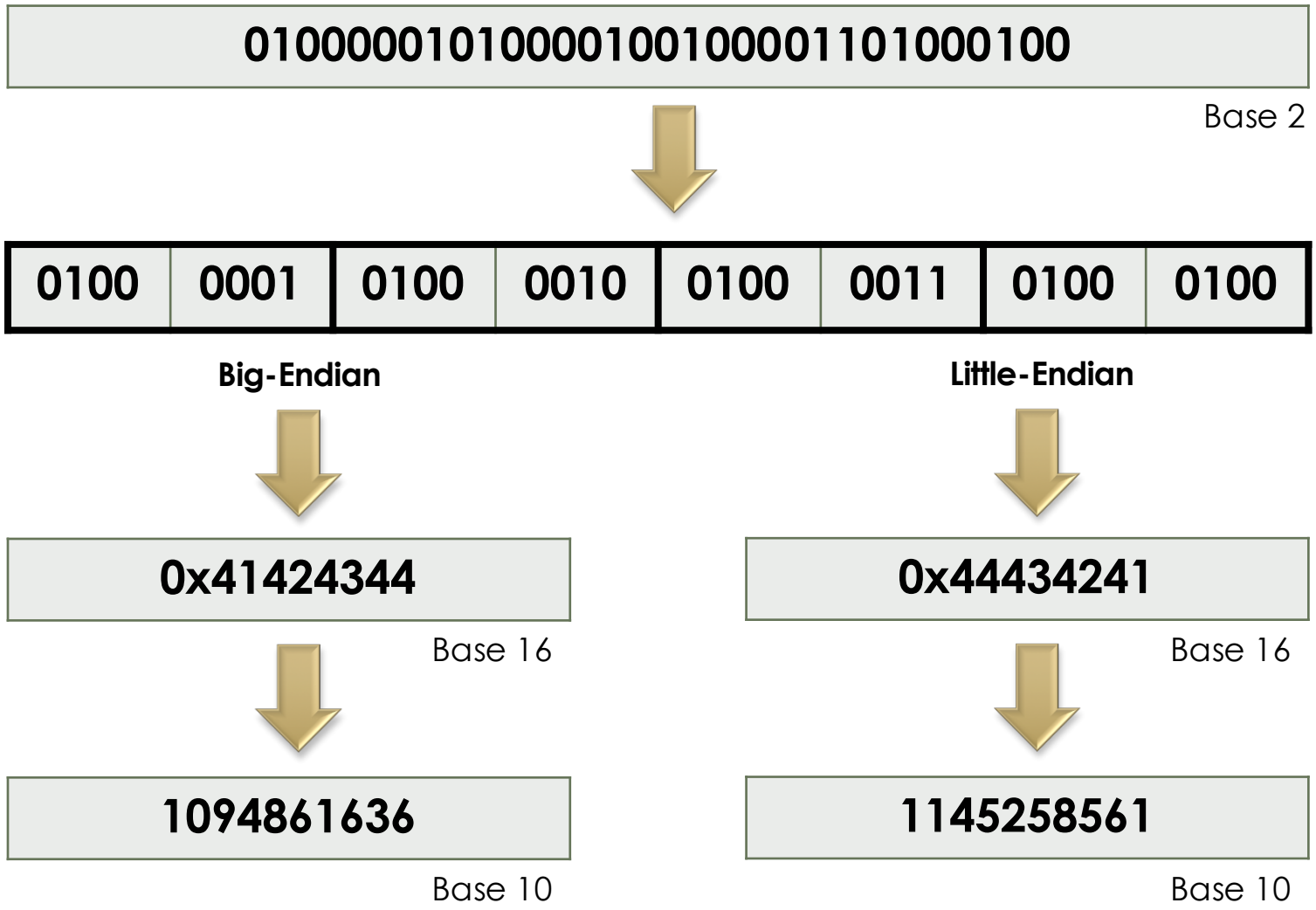



01000001010000100100001101000100
Base 2
0100 0001 0100 0010 0100 0011 0100 0100
Nibbles
0x41424344
Base 16
1094861636
Base 10

Big-Endian vs. Little-Endian



Intel is Little-Endian


01000001010000100100001101000100
Base 2
0100 0001 0100 0010 0100 0011 0100 0100
Big-Endian
Little-Endian
0x41424344
0x44434241
Base 16
Base 16
1094861636
1145258561
Base 10
Base 10

01000001010000100100001101000100

Base 2

0	1000001	01000010010000 1101000100

IEEE-754

781.035217

Base 10

01000001010000100100001101000100

Base 2

01000001	01000010	01000011	01000100

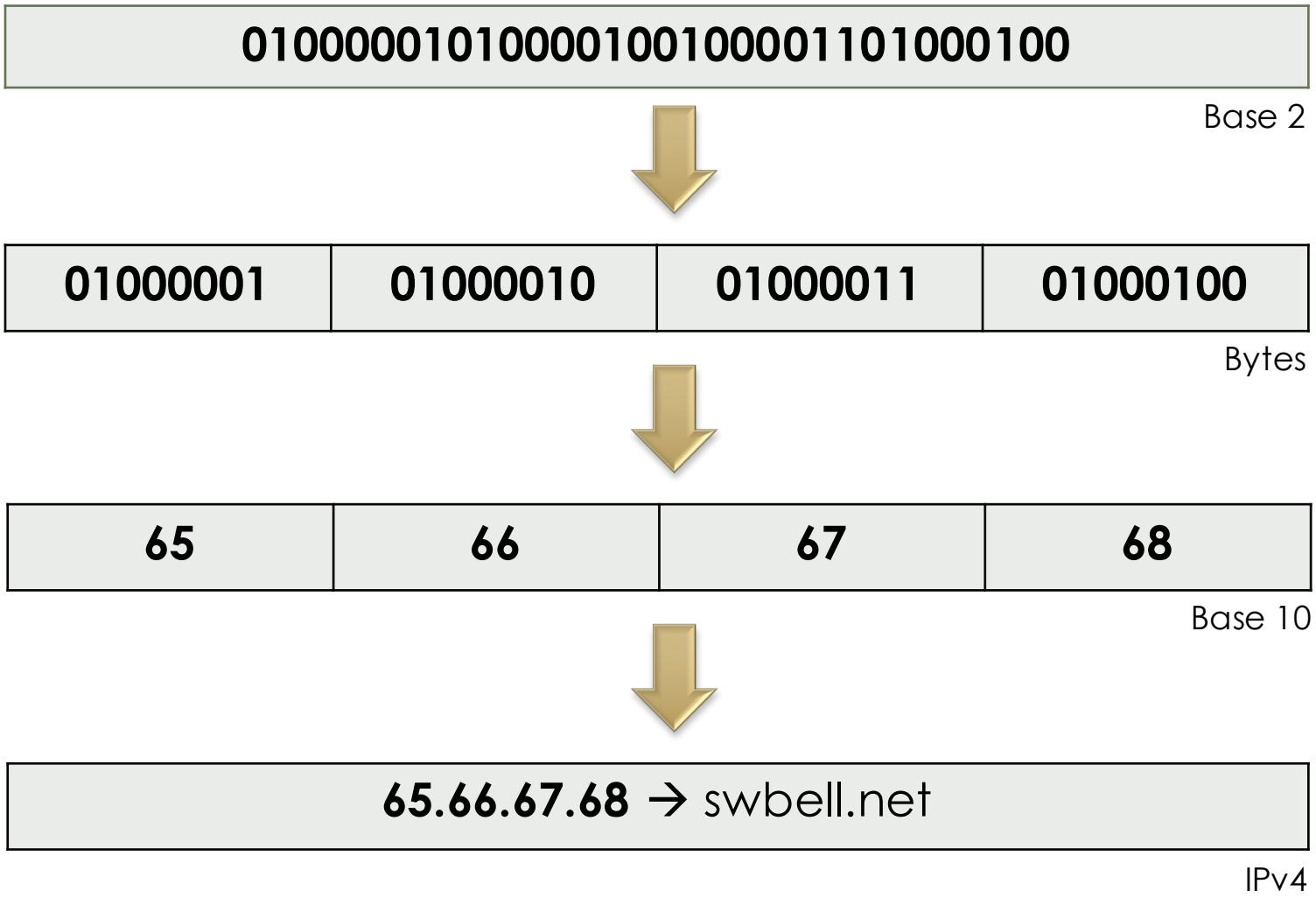
Bytes

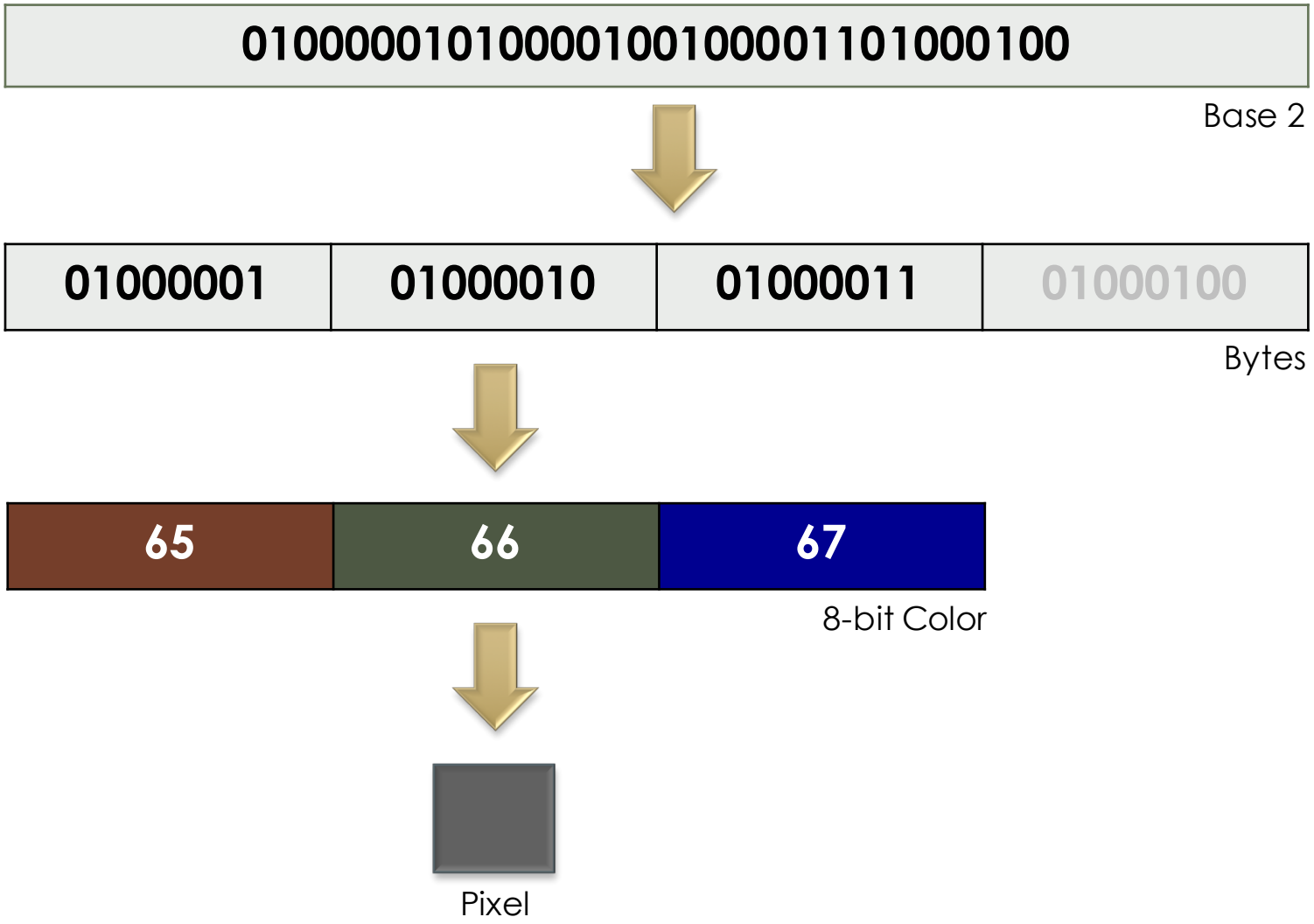
65	66	67	68

Base 10

'A'	'B'	'C'	'D'

ASCII


01000001010000100100001101000100
Base 2
01000001
01000010
01000011
01000100
Bytes
65
66
67
68
Base 10
65.66.67.68 → swbell.net
IPv4


01000001010000100100001101000100
Base 2
01000001
01000010
01000011
01000100
Bytes
65
66
67
8-bit Color
Pixel

01000001010000100100001101000100

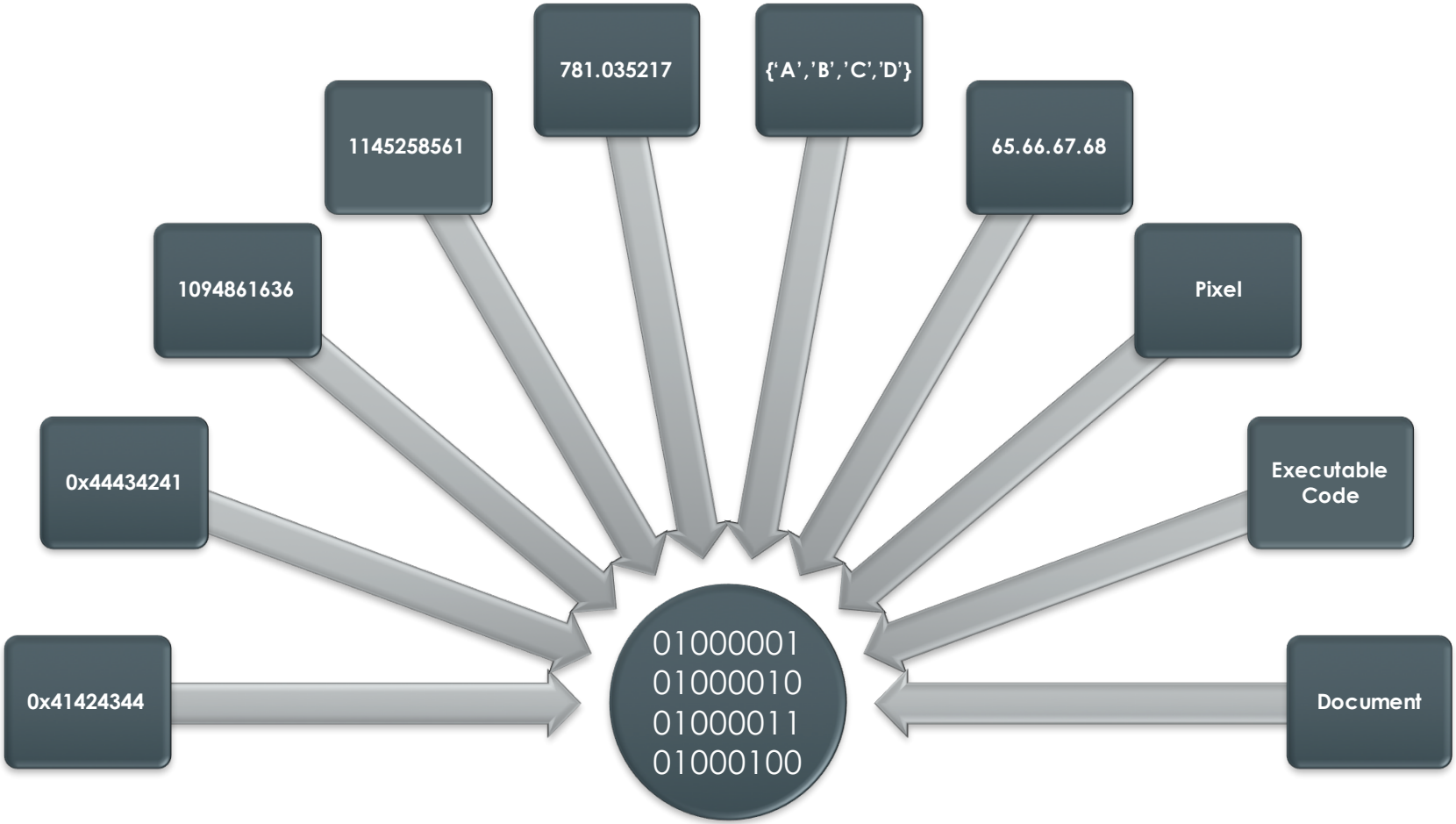
Base 2

01000001	01000010	01000011	01000100

Bytes

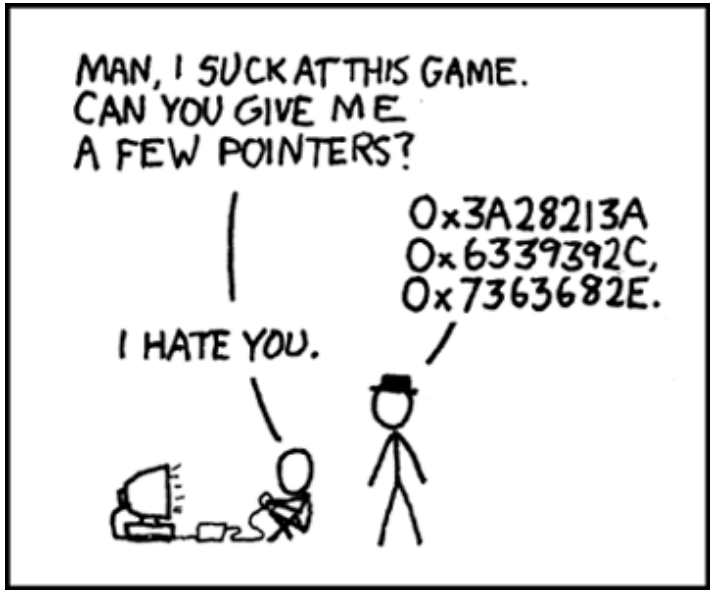
```
01000001  rex.B
01000010  rex.X
01000011  rex.XB
01000100  rex.R pop
```

Intel Assembly Code

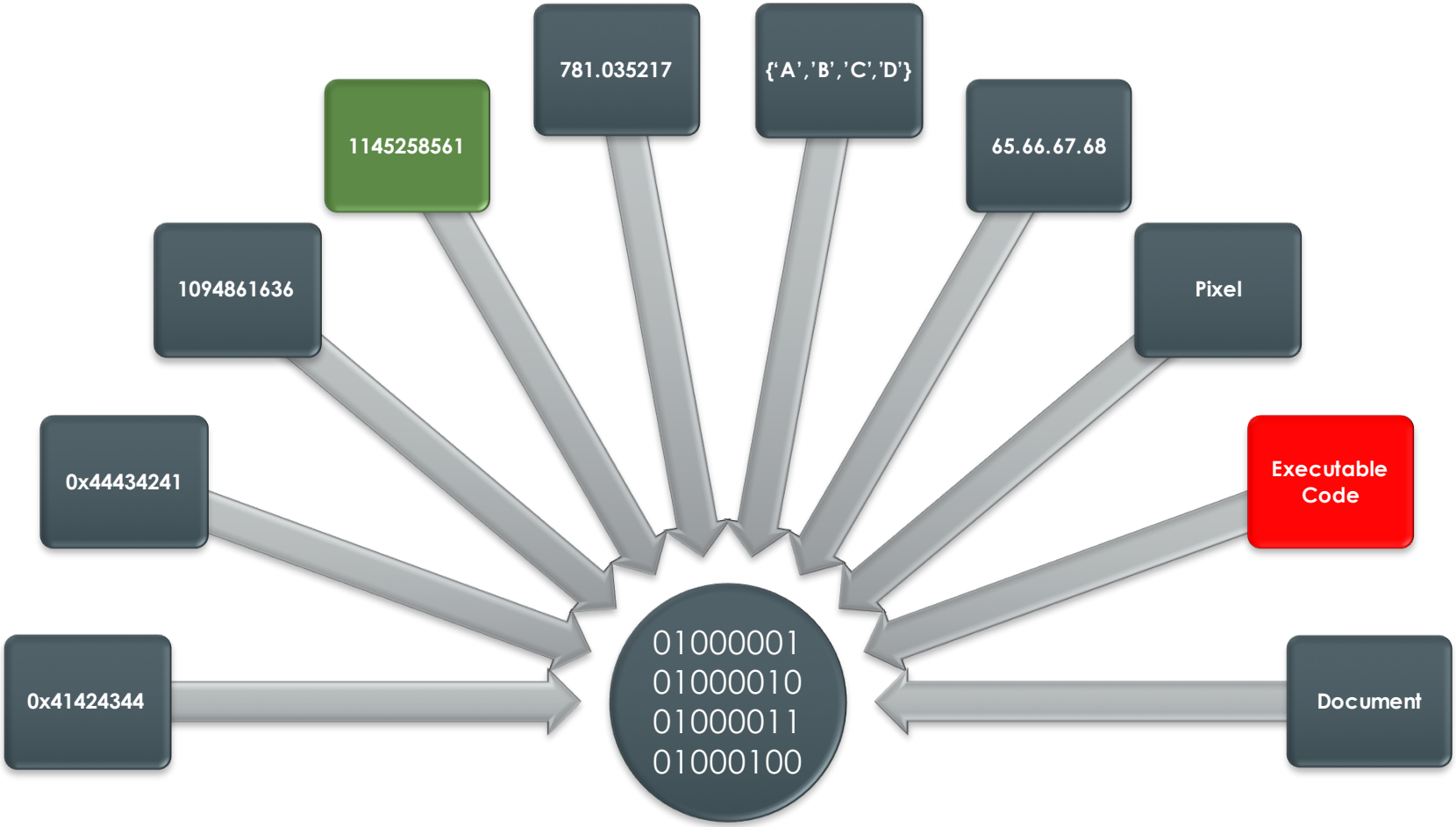

781.035217
{'A','B','C','D'}
1145258561
65.66.67.68
1094861636
Pixel
0x44434241
Executable Code
0x41424344
01000001
01000010
01000011
01000100
Document

Data Representation

What if our assumptions about how the data will be interpreted are wrong?



xkcd.com

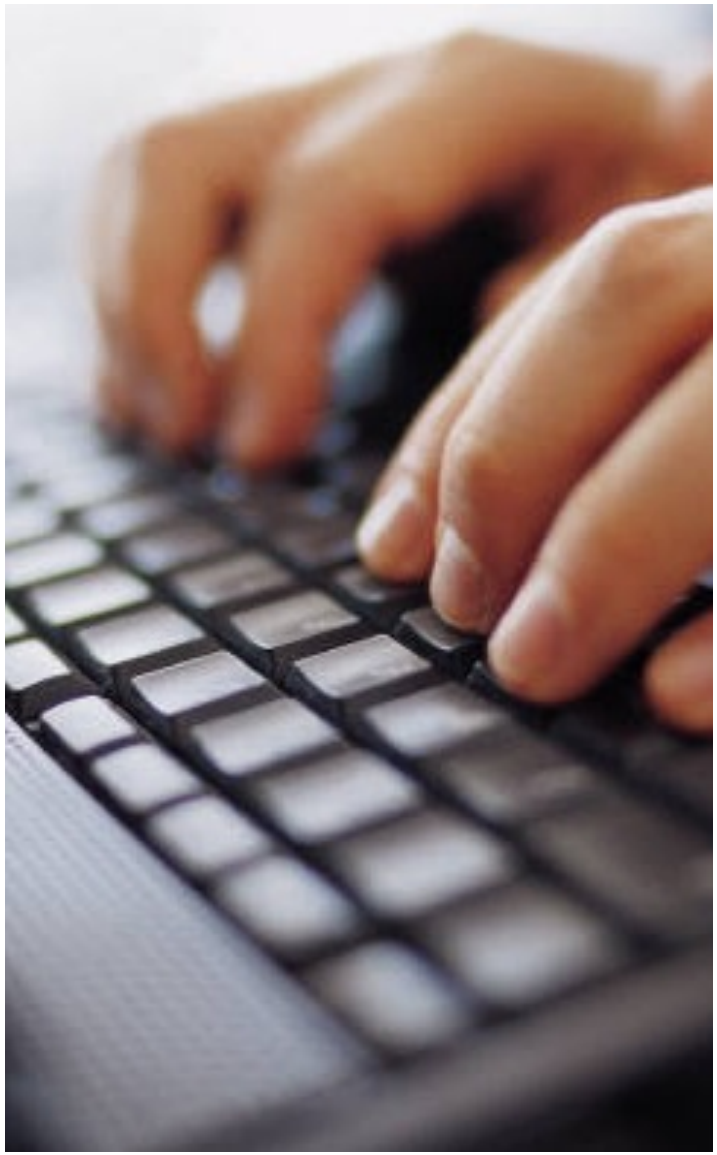


Many security exploits rely on data being interpreted in a different way than was originally intended.

```
unsigned long int i_hello = 0x6f6c6c6548;
unsigned long int i_world = 431316168567;

printf("%s %s", (char *)&i_hello, (char *)&i_world);
```

ASCII	Hex	Dec	ASCII	Hex	Dec
'A'	41	65	'a'	61	97
'B'	42	66	'b'	62	98
'C'	43	67	'c	63	99
'D'	44	68	'd'	64	100

Remember: Intel is Little-Endian; the bytes in an integer are stored in reverse order in memory.



Assembly Code

Intel Assembler, gdb

Intel Assembler

We will be making use of Intel assembler, both to discuss and to craft various security exploits. You will not need to be an expert, but should be comfortable in reading assembly code.

For reference, see:

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

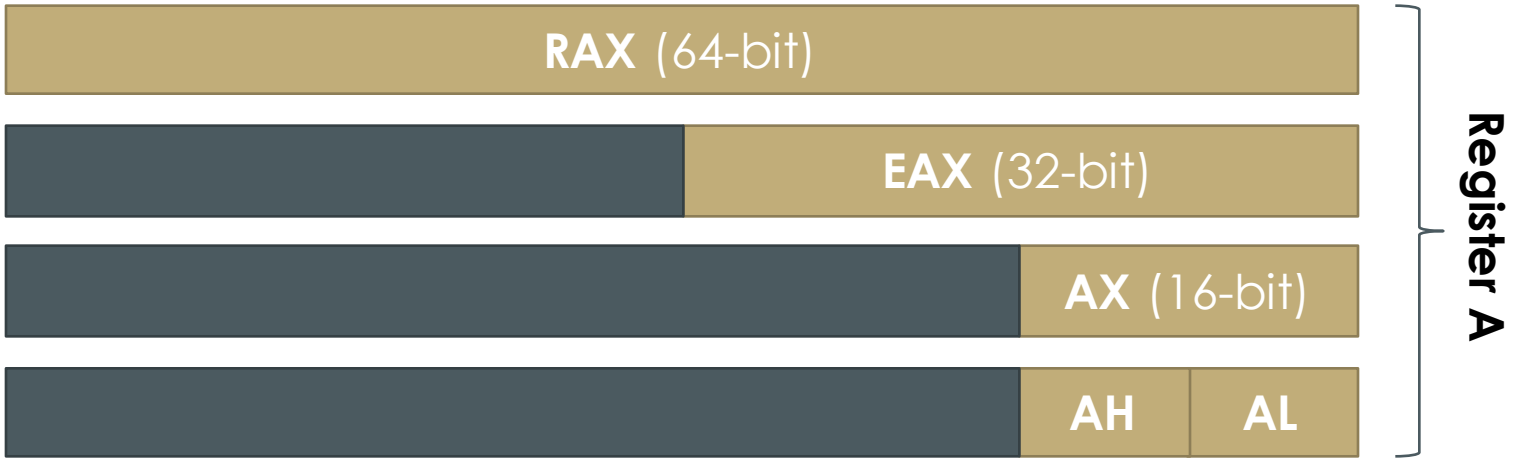
Intel Assembler

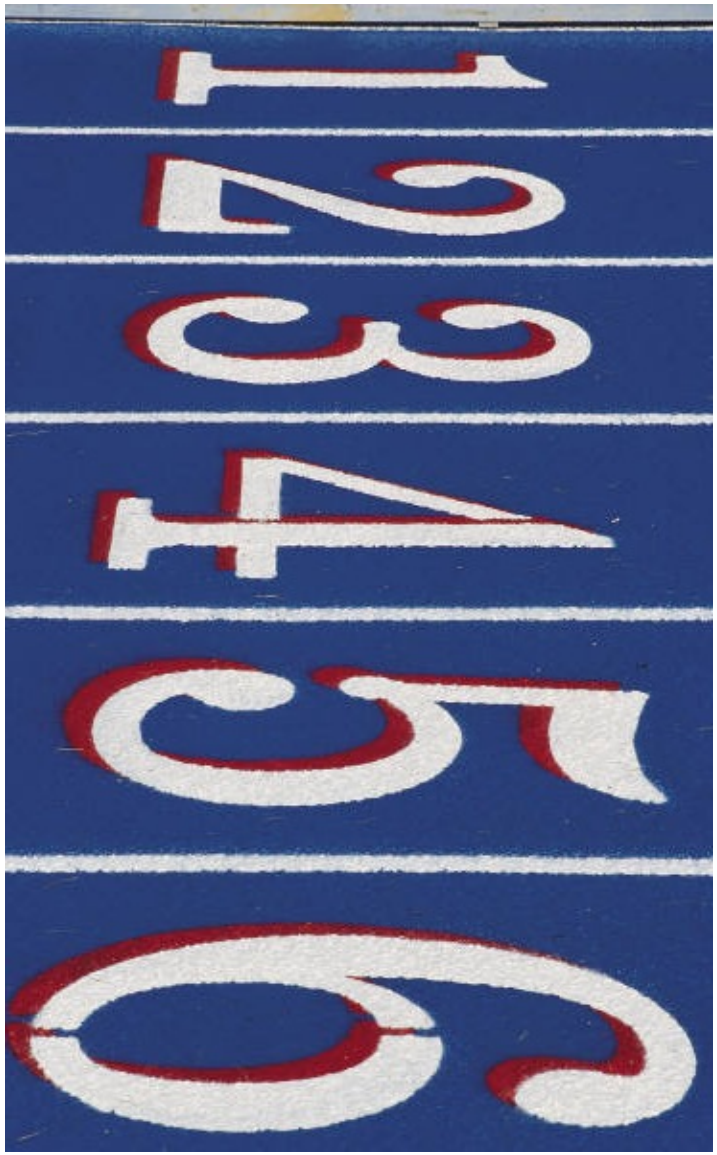
Intel assembly code looks very similar to most other assembly languages you may already be familiar with (e.g., Motorola 68k)

```
movl  -4(%rbp), %eax
addl  $1, %eax
movl  %eax, -16(%rbp)
```

Intel Registers

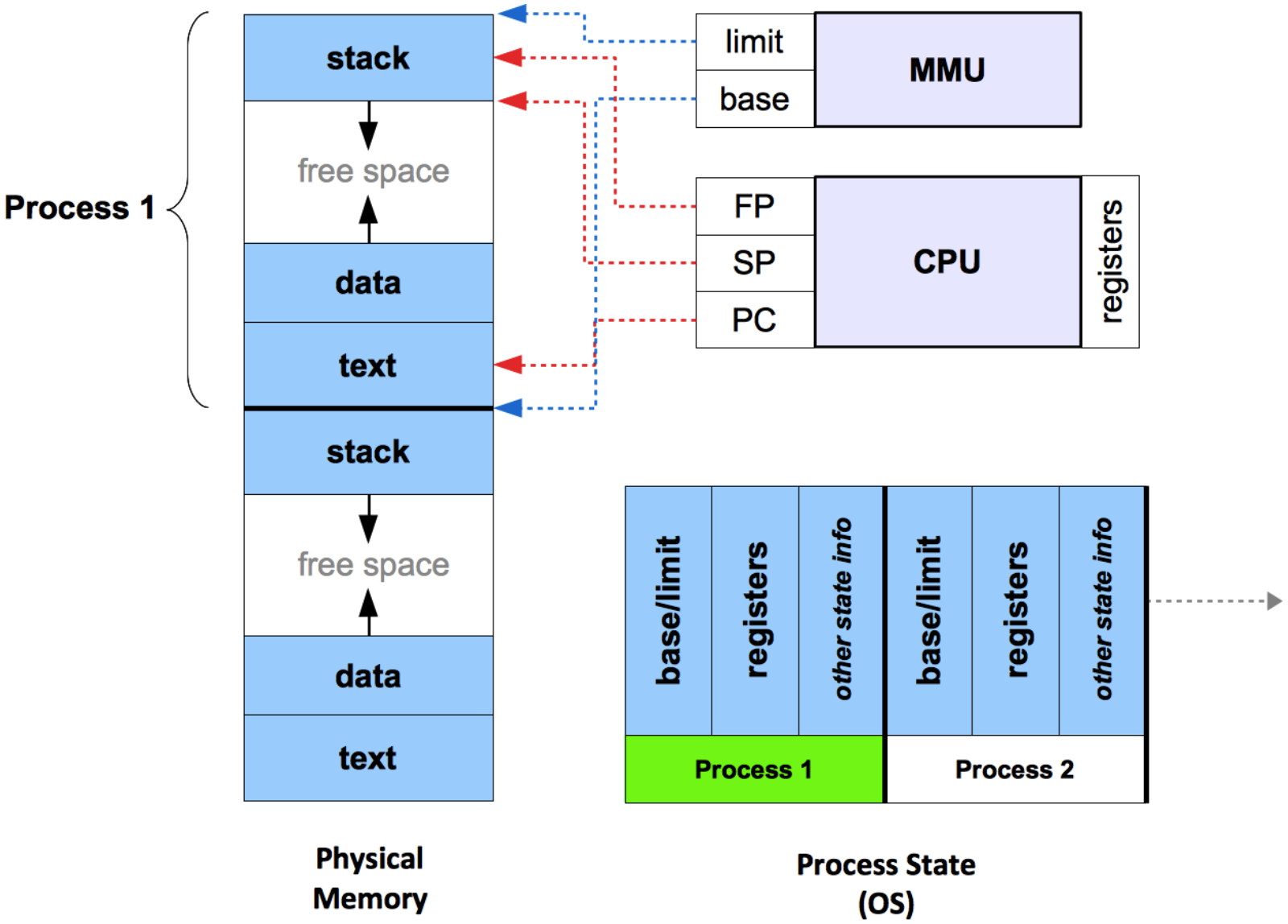
Current Intel CPUs have six general-purpose registers; the various bits in each register can be accessed in a number of ways:



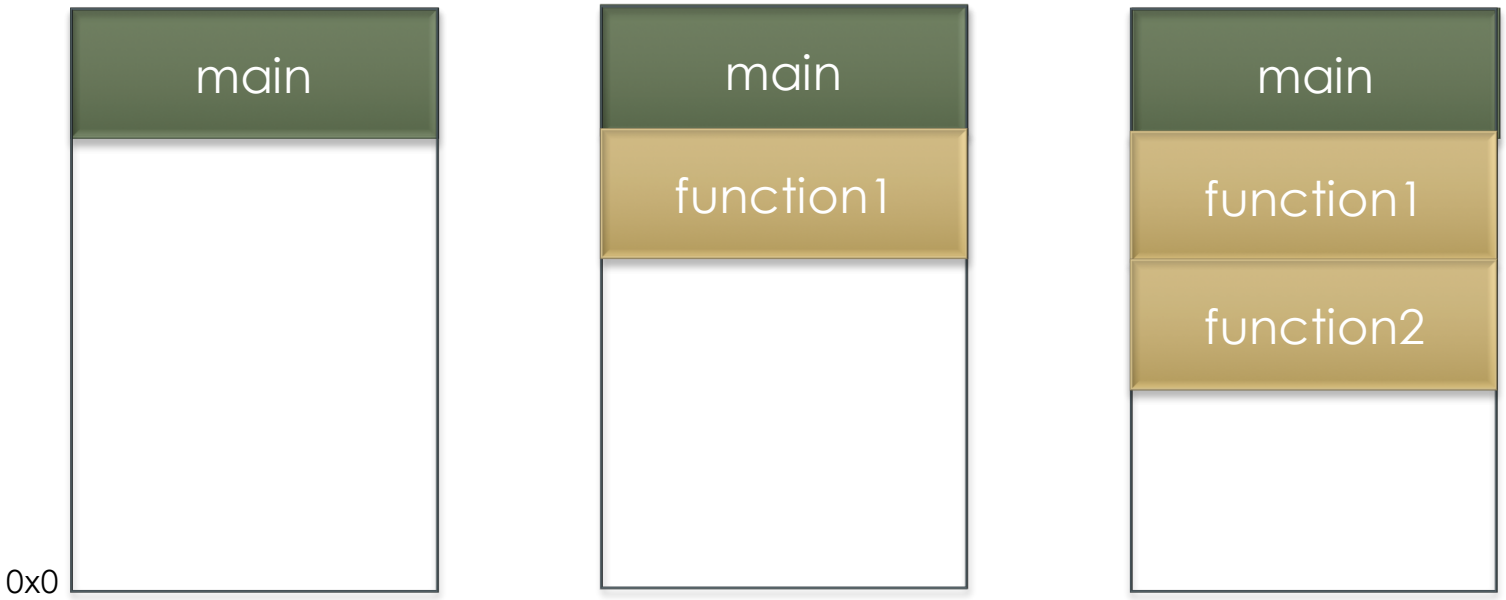


Stacks

Stack Frames, Function Calls


Process 1
stack
free space
data
text
stack
free space
data
text
limit
base
MMU
FP
SP
PC
CPU
registers
base/limit
registers
other state info
base/limit
registers
other state info
Process 1
Process 2
Physical Memory
Process State (OS)

Process Stack Growth



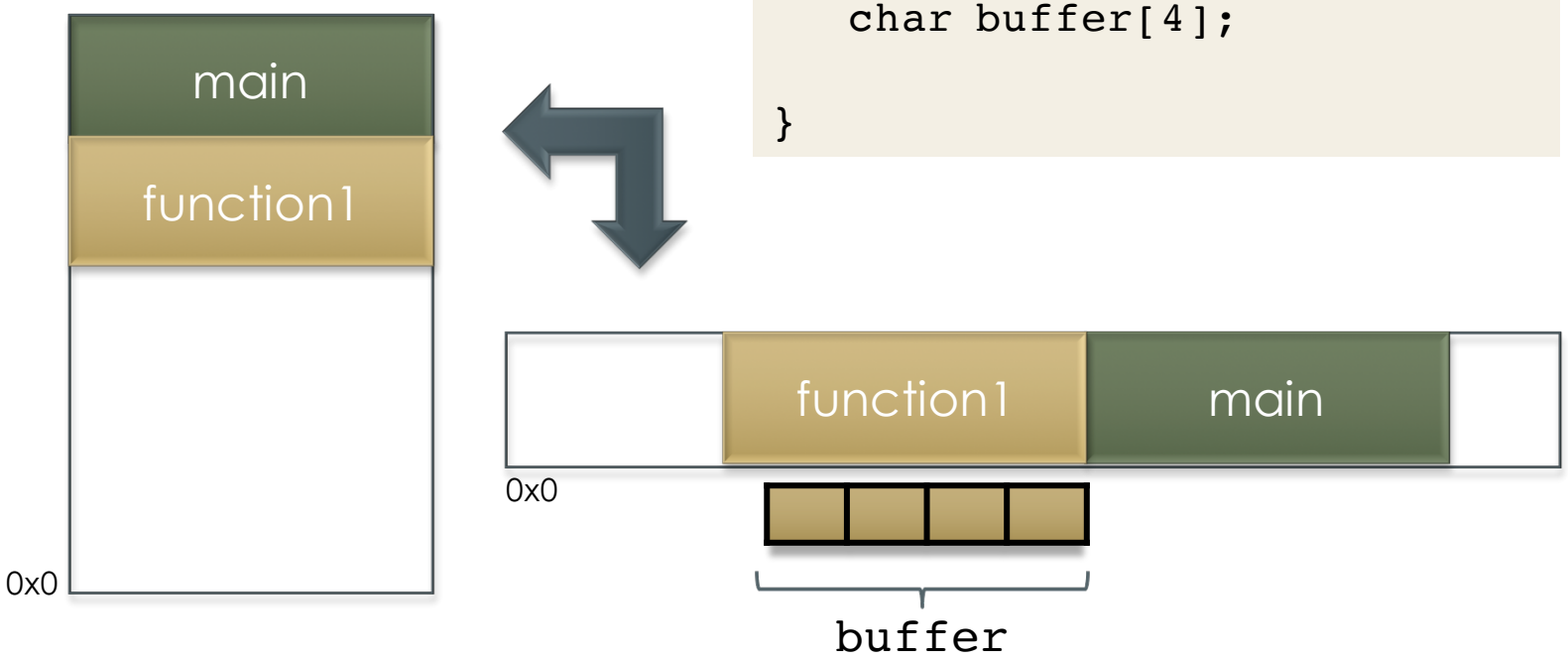
- The process stack grows downwards on Intel.

Stack Frames

Recall: What information does a stack frame contain?

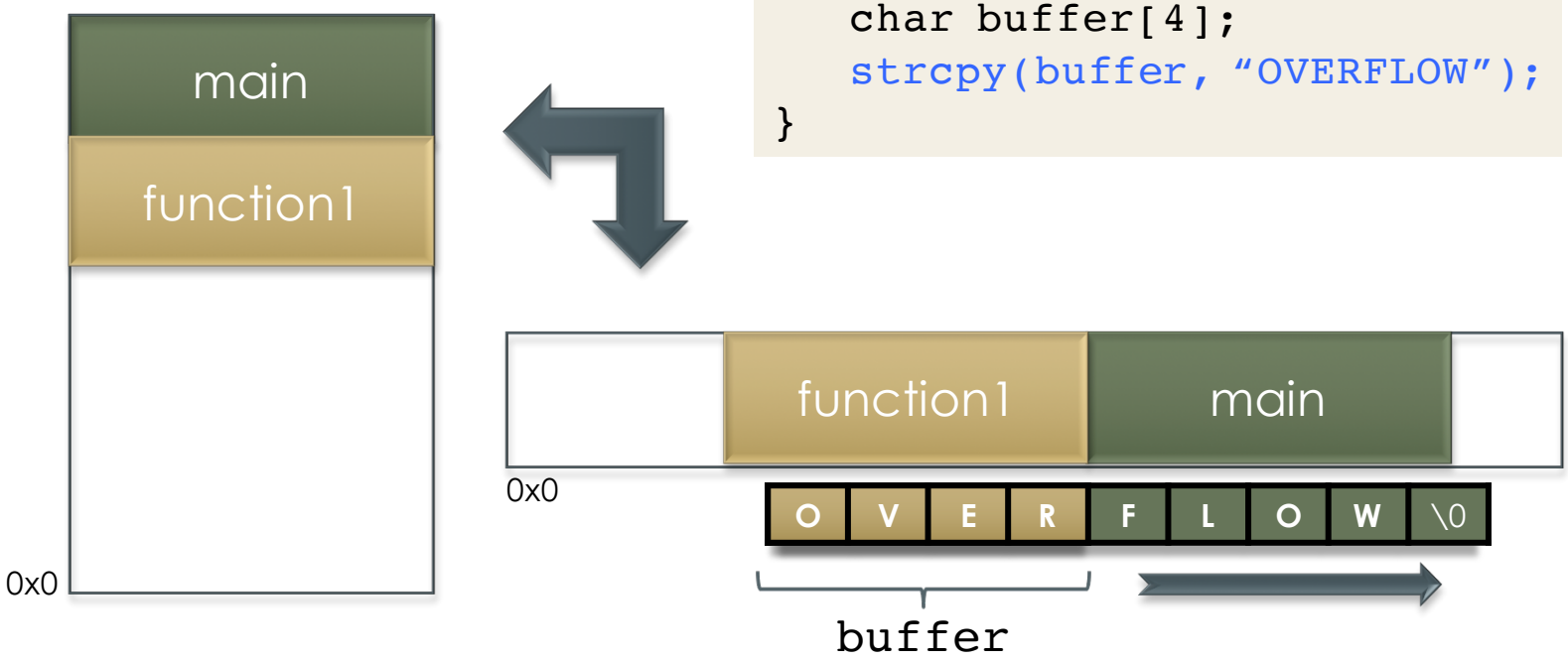
Process Stack Growth

```
void function1()
{
    char buffer[4];

}
```



Process Stack Growth

```
void function1()
{
    char buffer[4];
    strcpy(buffer, "OVERFLOW");
}
```

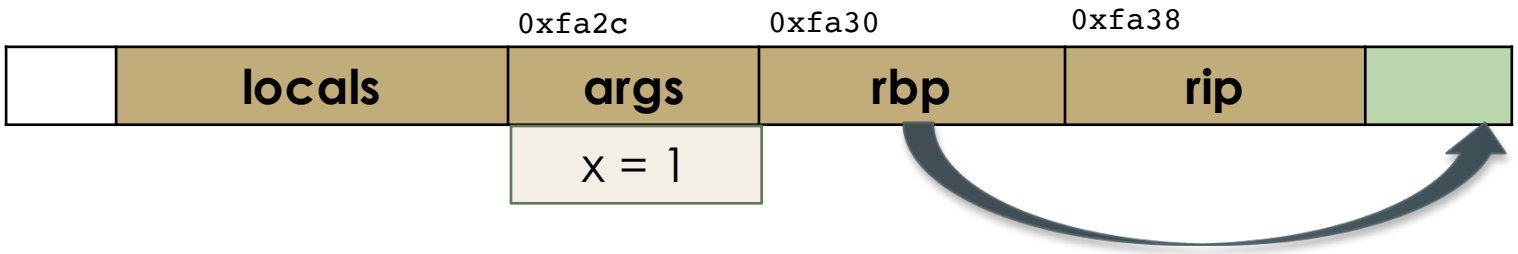


Using gdb to Examine Stacks

break	Defines a new breakpoint
run	Starts a new process
where	List of current stack frames
up / down	Move between frames
info frame	Display info on current frame
info args	List function arguments (“gcc –g”)
info locals	List all local variables (“gcc -g”)
print	Display a variable
x	Display the contents of memory

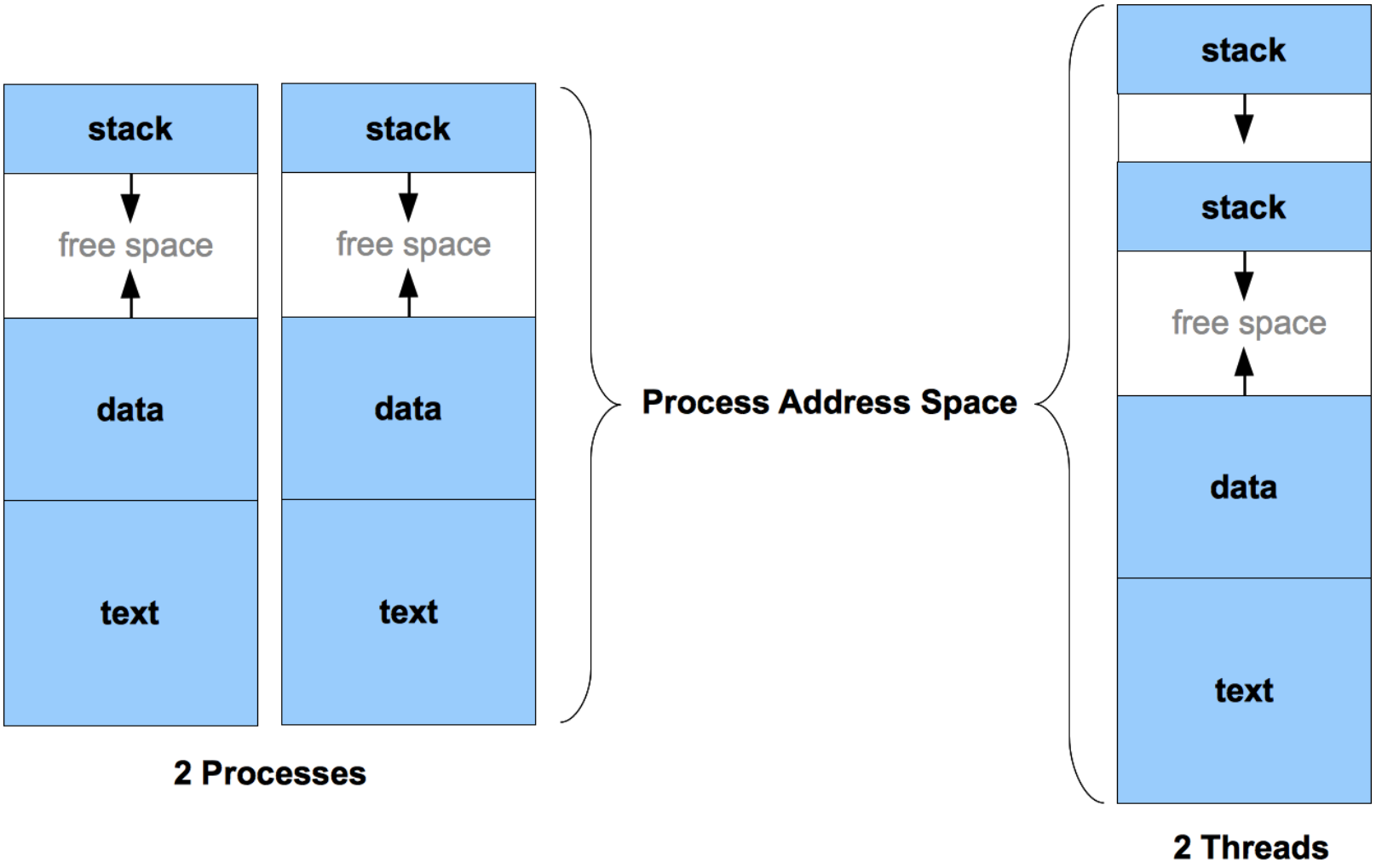
gdb: info frame

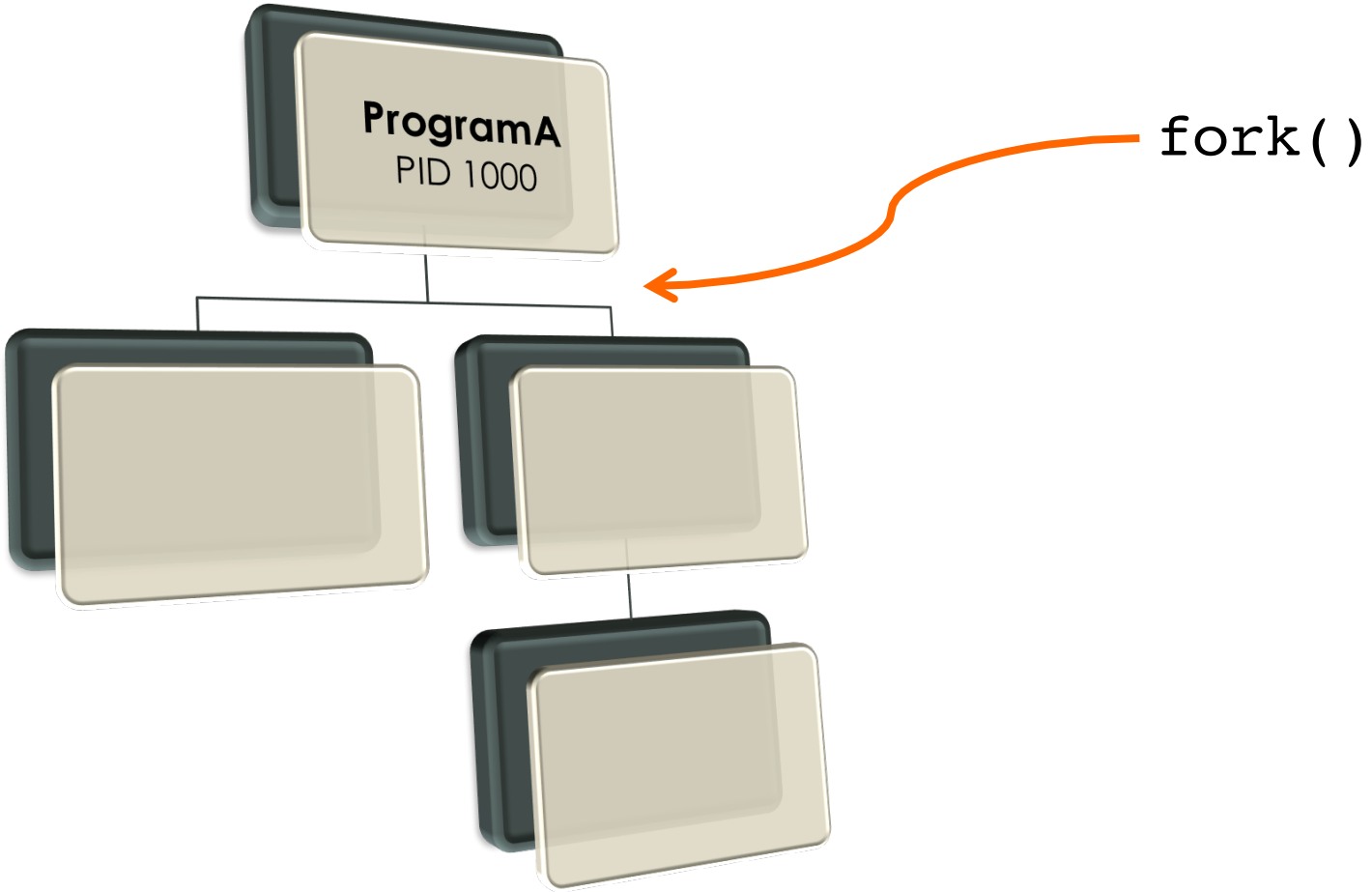
```
(gdb) info frame
Stack level 0, frame at 0xfa40:
 rip = 0x100000ec7 in foo (foo.c:3); saved rip 0x100000ef9
 called by frame at 0xfa70
 source language c.
 Arglist at 0xfa38, args: x=1
 Locals at 0xfa38, Previous frame's sp is 0xfa40
 Saved registers:
  rdi at 0xfa2c, rbp at 0xfa30, rip at 0xfa38
(gdb)
```

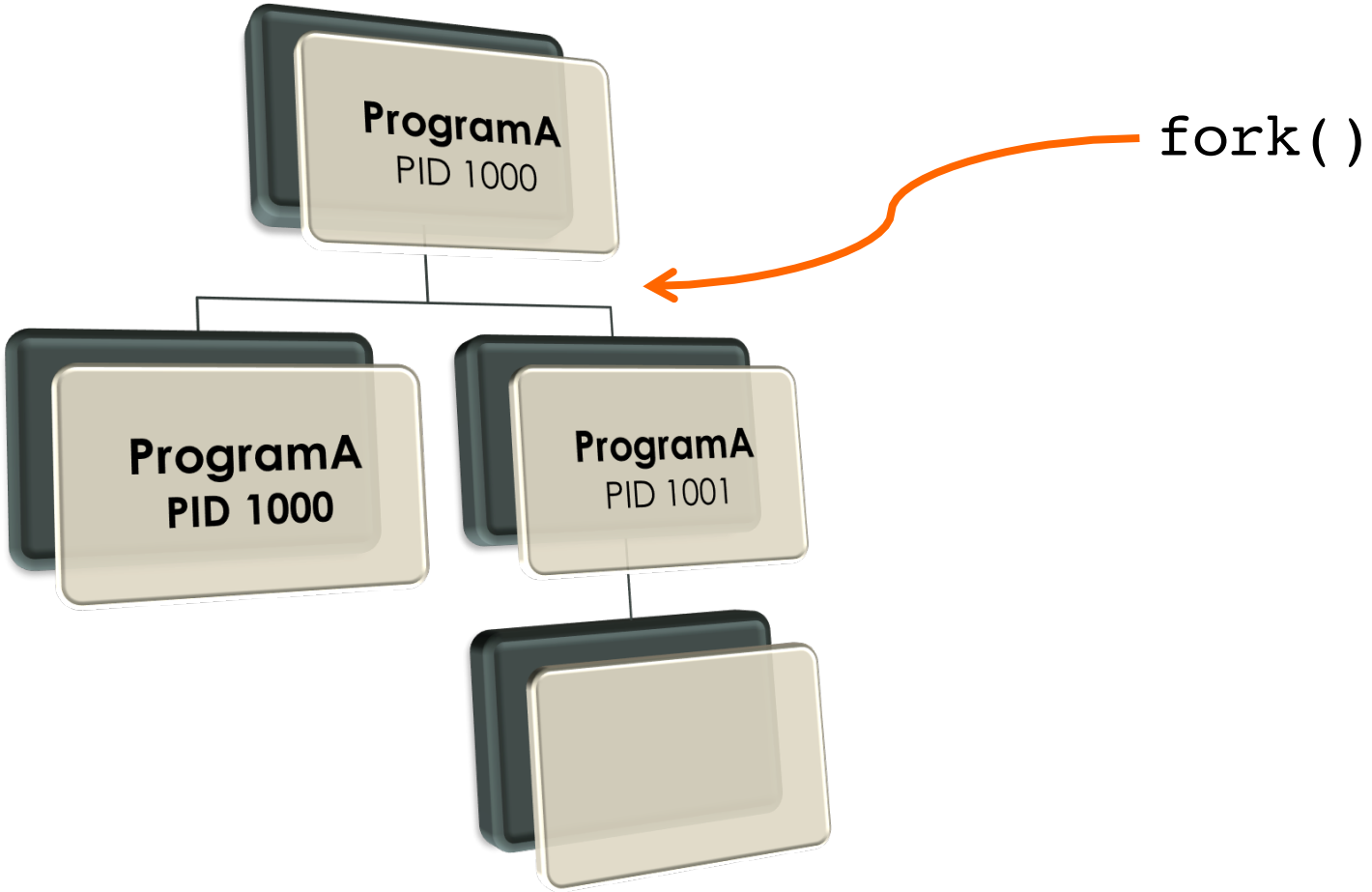


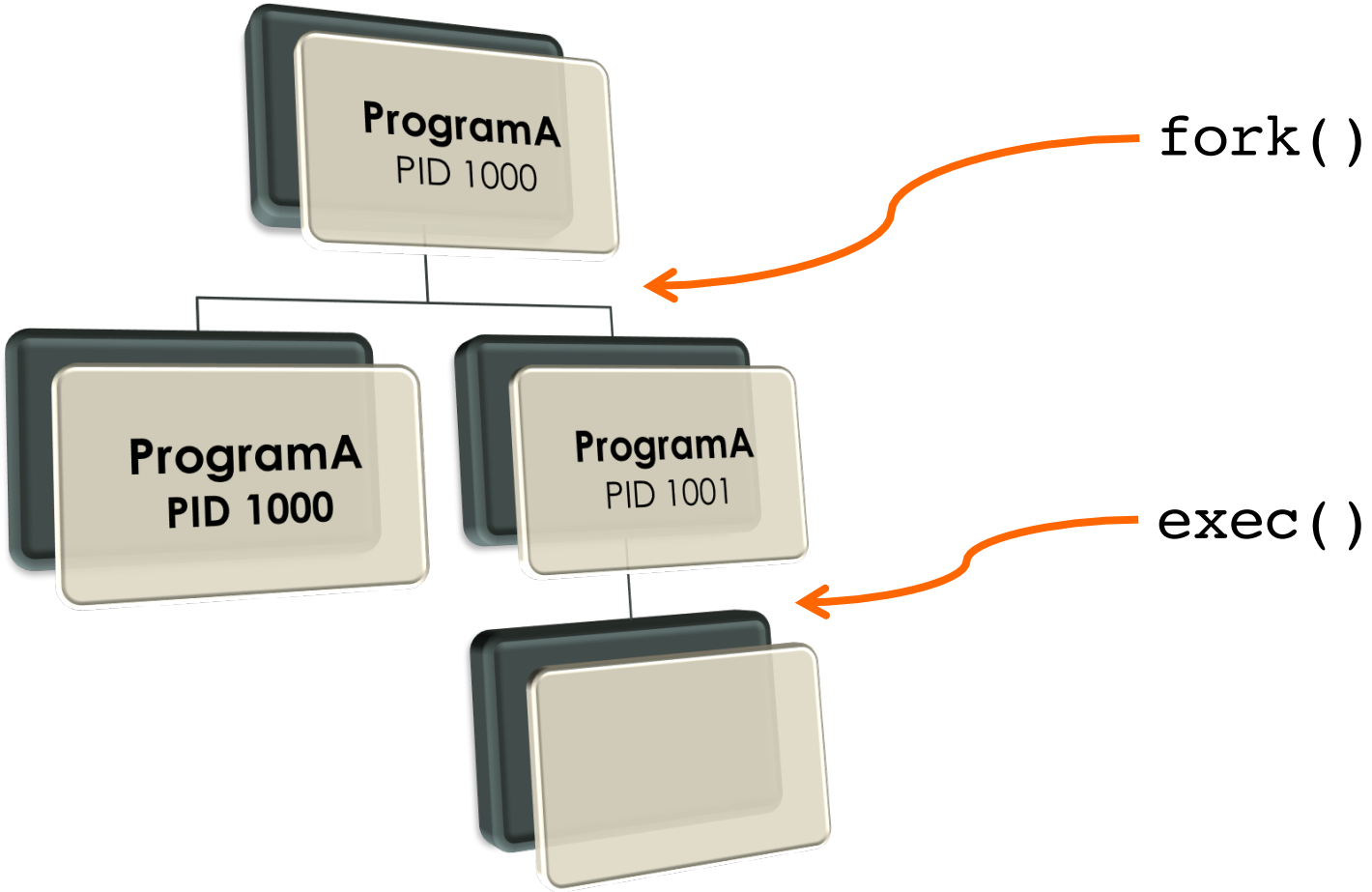
Processes

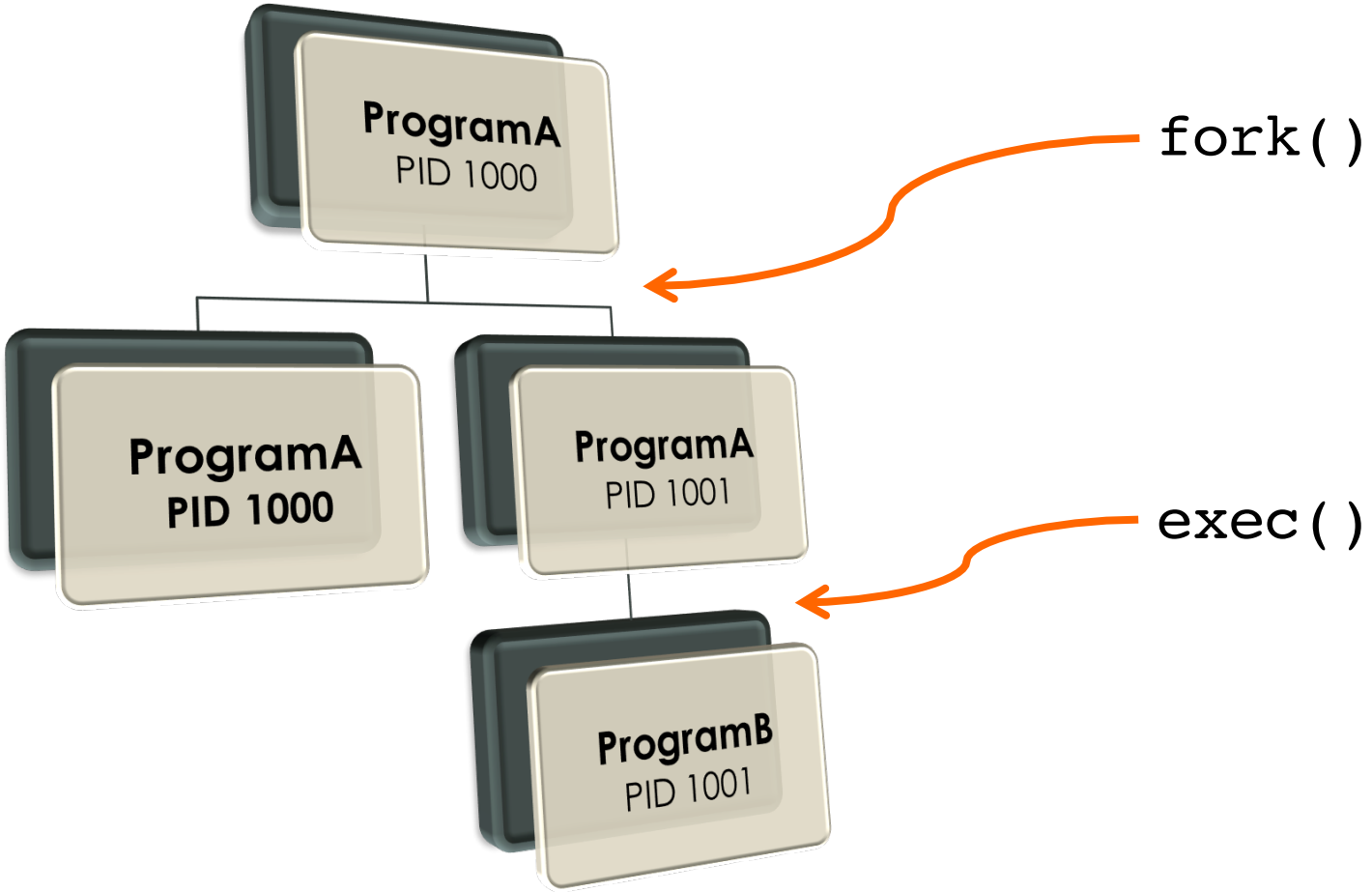
Threads, fork, exec


stack
free space
data
text
stack
free space
data
text
Process Address Space
2 Processes
stack
stack
free space
data
text
2 Threads


ProgramA
PID 1000
fork()


ProgramA
PID 1000
fork()
ProgramA
PID 1000
ProgramA
PID 1001


ProgramA
PID 1000
fork()
ProgramA
PID 1000
ProgramA
PID 1001
exec()


ProgramA
PID 1000
fork()
ProgramA
PID 1000
ProgramA
PID 1001
exec()
ProgramB
PID 1001