System Calls

David Ferry
CSCI 2510 – Principles of Computing Systems
Saint Louis University
St. Louis, MO 63103

Example of Policy & Mechanism

Policy describes how a system should work.

Mechanism implements policies.

- Mechanism should not dictate restrictions to policy

System calls:

- OS should provide safety and access control
- System call mechanism must support that

“System” call? As opposed to what?

Consider example program:

```
int max( int a, int b ){
    if ( a > b ) return a; else return b;
}

int main( int argc, char* argv[] ){
    int z = max( 5, 10 );
    char buffer[ bufferSize ];
    int bytes = snprintf( buffer, bufferSize, “Max is %d”, z );
    write( STDOUT_FILENO, buffer, bytes);
}
```

“System” call? As opposed to what?

Consider example program:

```
int max( int a, int b ){
    if ( a > b ) return a; else return b;
}

int main( int argc, char* argv[] ){
    int z = max( 5, 10 );
    char buffer[ bufferSize ];
    int bytes = snprintf( buffer, bufferSize, “Max is %d”, z );
    write( STDOUT_FILENO, buffer, bytes);
}
```

Implemented by our program.

“System” call? As opposed to what?

Consider example program:

```
int max( int a, int b ){
    if ( a > b ) return a; else return b;
}

int main( int argc, char* argv[] ){
    int z = max( 5, 10 );
    char buffer[ bufferSize ];
    int bytes = snprintf( buffer, bufferSize, “Max is %d”, z );
    write( STDOUT_FILENO, buffer, bytes);
}
```

Implemented by our program.

Implemented by C standard library.

“System” call? As opposed to what?

Consider example program:

```
int max( int a, int b ){
    if ( a > b ) return a; else return b;
}

int main( int argc, char* argv[] ){
    int z = max( 5, 10 );
    char buffer[ bufferSize ];
    int bytes = snprintf( buffer, bufferSize, “Max is %d”, z );
    write( STDOUT_FILENO, buffer, bytes);
}
```

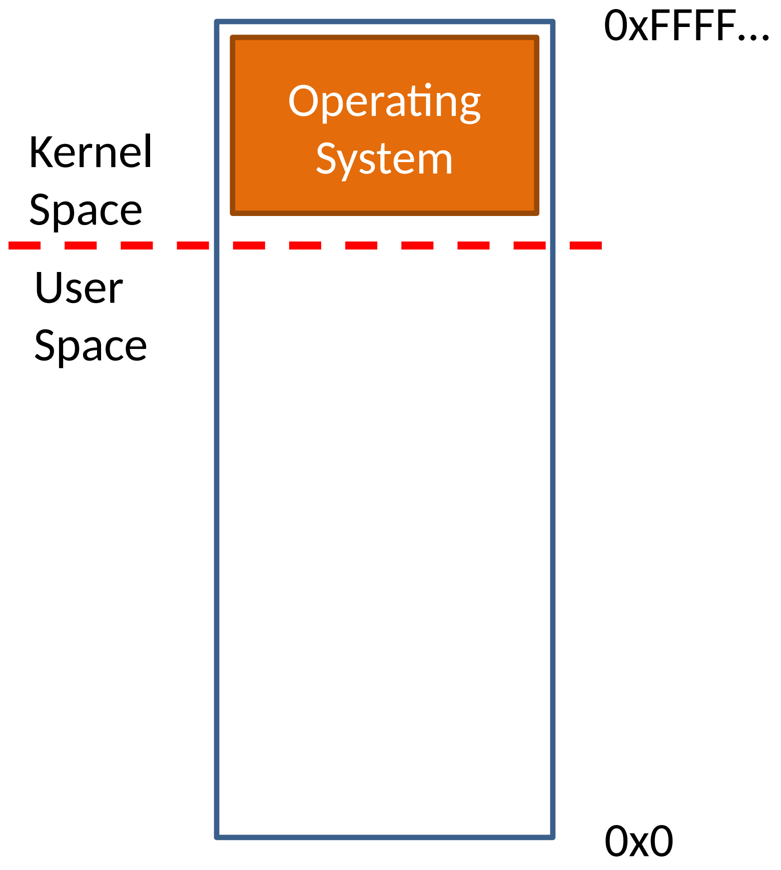
Implemented by our program.

Implemented by C standard library.

Implemented by the operating system
- Needs OS cooperation to work
- AKA “System call” – call to the operating system

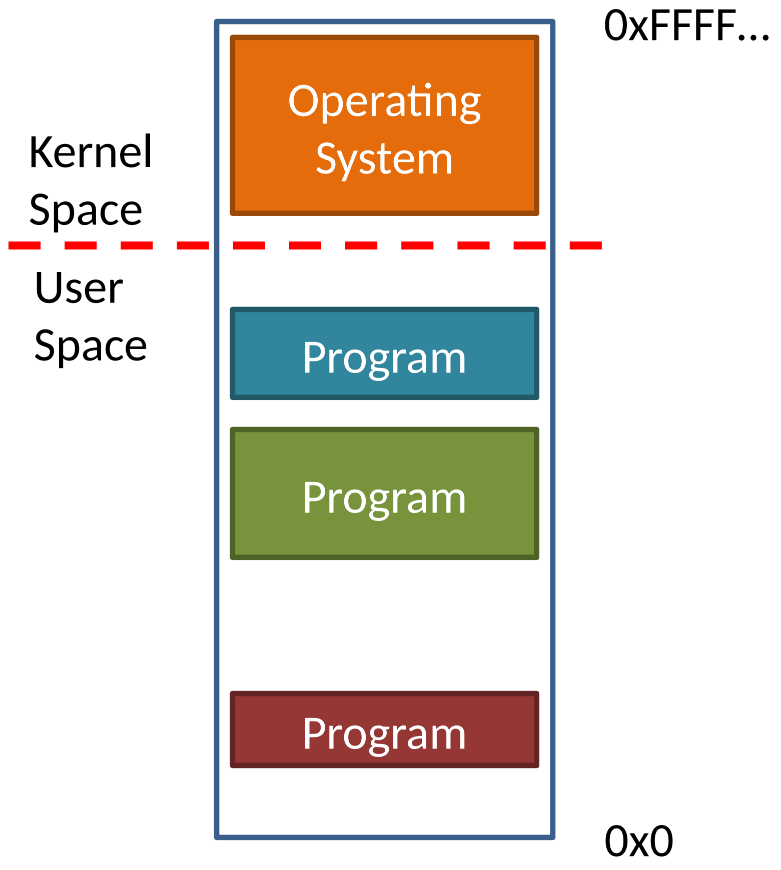
Where does each code fragment reside?

Imagine the entire memory of a machine:



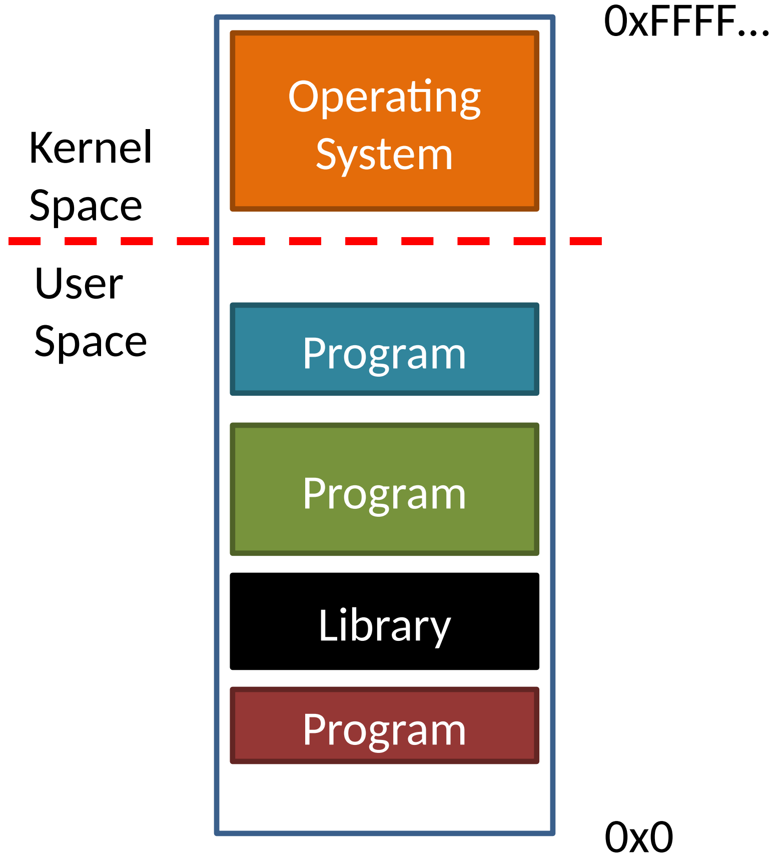
Where does each code fragment reside?

Imagine the entire memory of a machine:



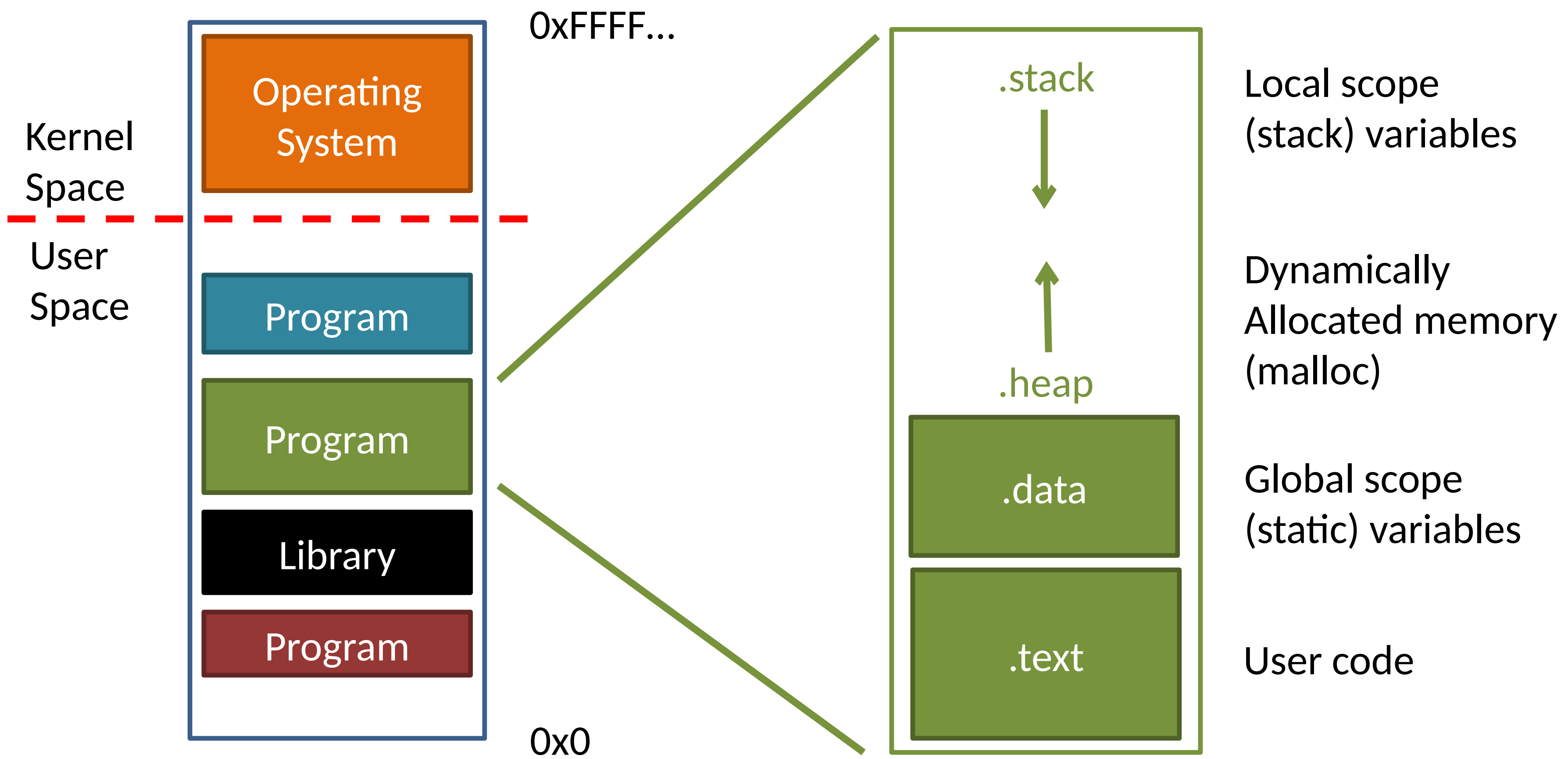
Where does each code fragment reside?

Imagine the entire memory of a machine:

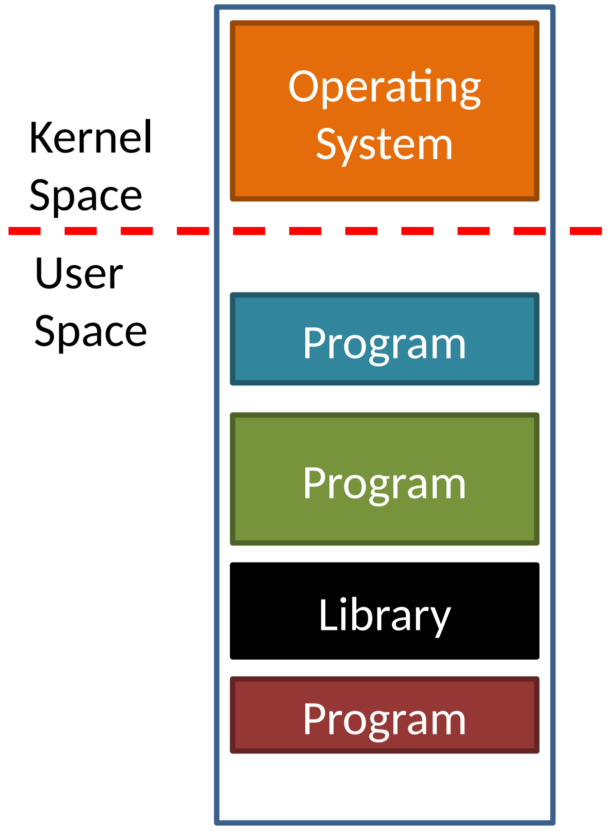


Where does each code fragment reside?

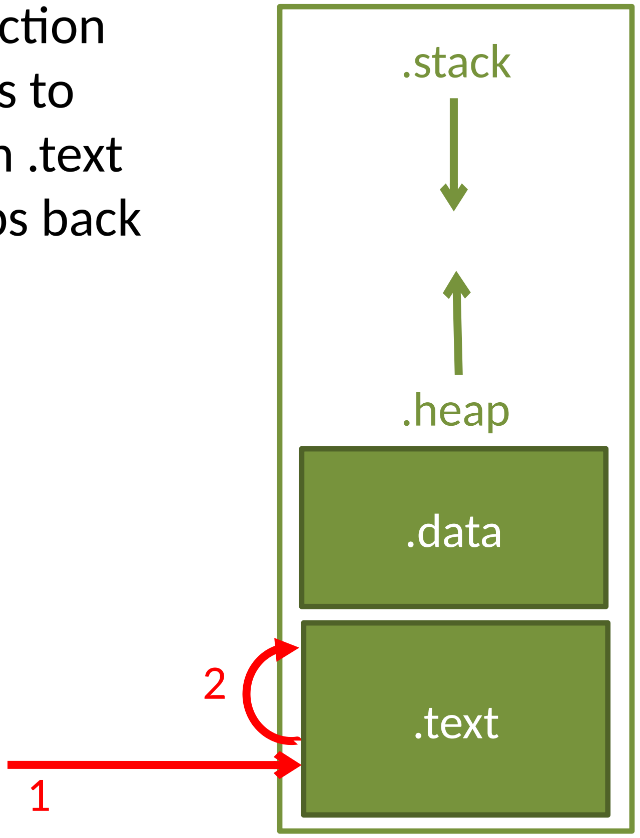
Imagine the entire memory of a machine:



Single-program function calls

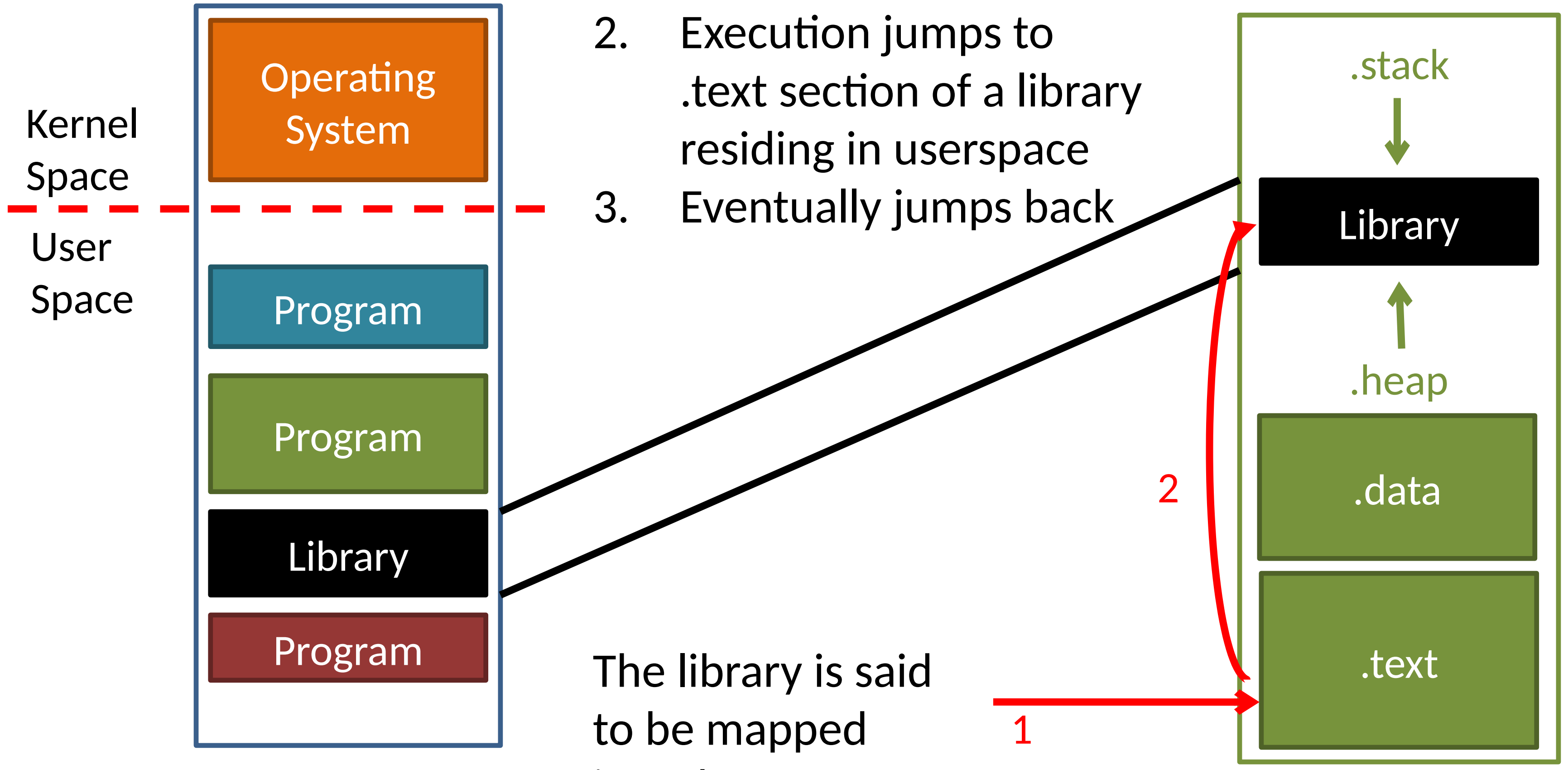


1. Code executing at some place in .text section
2. Execution jumps to another place in .text
3. Eventually jumps back



Library function calls

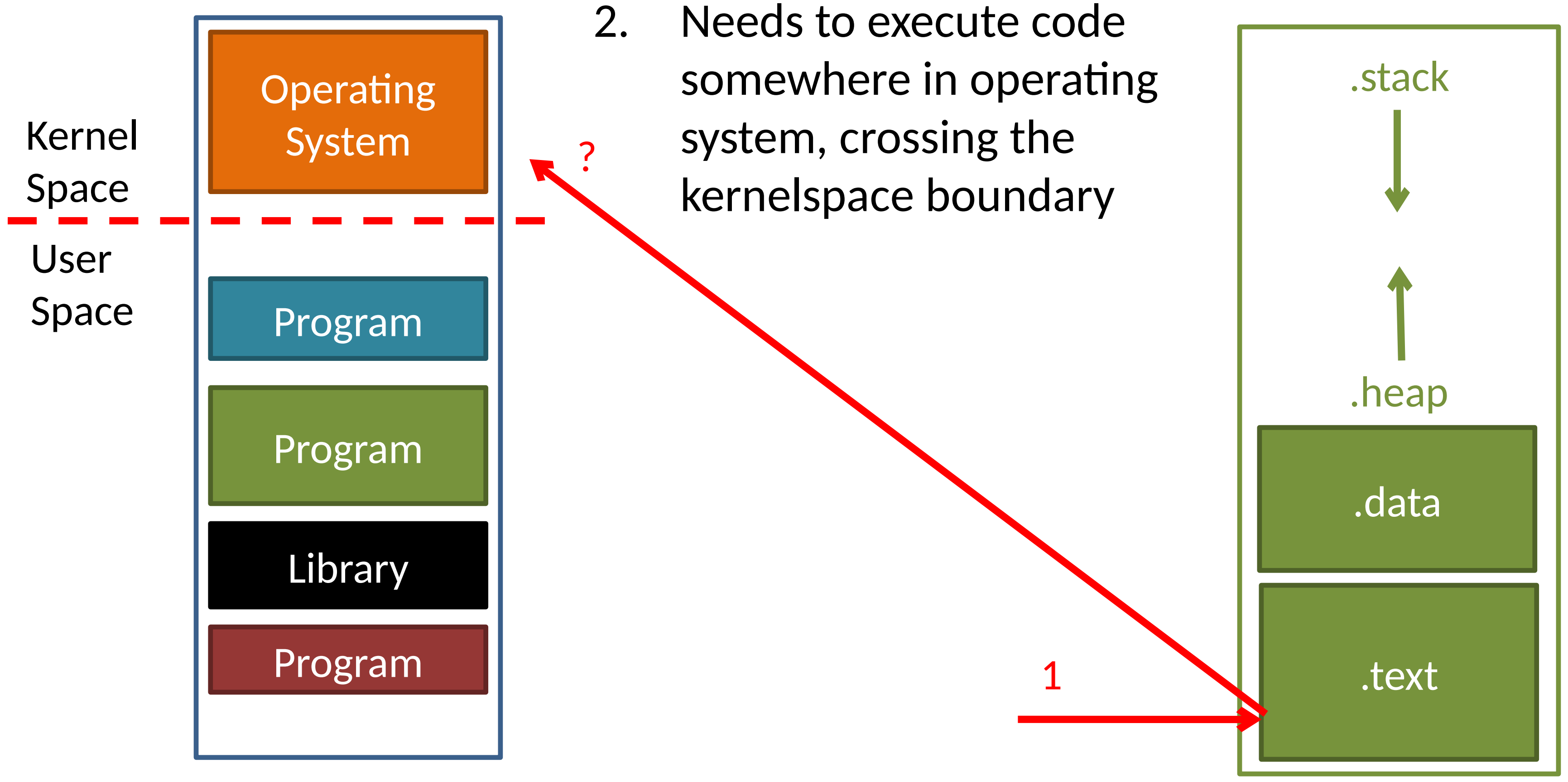
1. Code executing at some place in .text section
2. Execution jumps to .text section of a library residing in userspace
3. Eventually jumps back



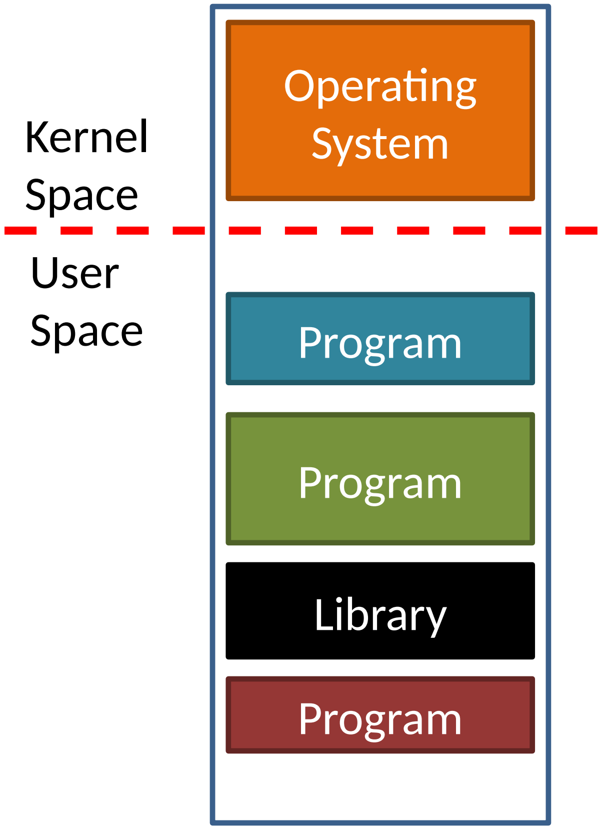
The library is said to be mapped into the program.

System calls

1. Code executing at some place in .text section
2. Needs to execute code somewhere in operating system, crossing the kernelspace boundary



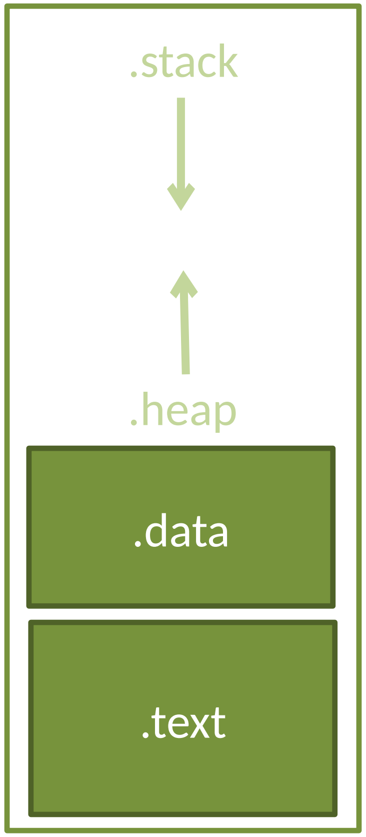
System calls



1. Code executing at some place in .text section
2. Needs to execute code somewhere in operating system, crossing the kernelspace boundary

Difficulties:

1) We can't give user programs access to OS memory like we can to system libraries
2) Need to protect system and other users from unauthorized access



Being Careful About Access to OS

What bad things can happen if a user program can read, write, or execute in the OS memory?

- The OS enforces access control (e.g. file read/write/delete, event permissions)
- The OS stores and authenticates secrets (e.g. password authentication)

Allowing unprivileged access circumvents OS security and policies.

Question: How does a user program achieve OS tasks without getting access to the OS?

System Call Mechanism

1. User program requests OS services
2. User program stops executing, hands execution over to OS
3. OS determines whether request is valid and allowable for user
4. OS performs service, if valid
5. OS returns a status code to user program
6. OS stops executing, hands execution back to user program

*Main point: OS code determines when/how to run.

Low Level Implementation

Low level mechanisms are finicky, for example, recall how a single function call is implemented in assembly.

int z = max(5, 10)

Making a function call at the low level…

1. Arguments stored in registers or on stack (depends on what ISA you're using)
2. Unconditional jump to function code
3. Update stack and base pointers
4. Execute function code
5. Store return value in known place
6. Unconditional jump back to calling code
7. Restore stack

32-bit x86 ISA Function Call

In x86:

```
pushl $10
pushl $5
```

int z = max(5, 10)

Making a function in x86…

1. Arguments stored on stack
2. Unconditional jump to function code
3. Update stack and base pointers
4. Execute function code
5. Store return value in known place
6. Unconditional jump back to calling code
7. Restore stack

32-bit x86 ISA Function Call

int z = max(5, 10)

In x86:

```
pushl $10
pushl $5
call  max
```

Making a function in x86…

1. Arguments stored on stack
2. Unconditional jump to function code
3. Update stack and base pointers
4. Execute function code
5. Store return value in known place
6. Unconditional jump back to calling code
7. Restore stack

32-bit x86 ISA Function Call

int z = max(5, 10)

Making a function in x86…

1. Arguments stored on stack
2. Unconditional jump to function code
3. Update stack and base pointers
4. Execute function code
5. Store return value in known place
6. Unconditional jump back to calling code
7. Restore stack

In x86:

```
pushl $10
pushl $5
call  max
pushl %ebp
movl  %esp, %ebp
```

32-bit x86 ISA Function Call

int z = max(5, 10)

Making a function in x86…

1. Arguments stored on stack
2. Unconditional jump to function code
3. Update stack and base pointers
4. Execute function code
5. Store return value in known place
6. Unconditional jump back to calling code
7. Restore stack

In x86:

```
pushl $10
pushl $5
call  max
pushl %ebp
movl  %esp, %ebp
<function code executes>
```

32-bit x86 ISA Function Call

int z = max(5, 10)

Making a function in x86…

1. Arguments stored on stack
2. Unconditional jump to function code
3. Update stack and base pointers
4. Execute function code
5. Store return value in known place
6. Unconditional jump back to calling code
7. Restore stack

In x86:

```
pushl $10
pushl $5
call  max
pushl %ebp
movl  %esp, %ebp
<function code executes>
<move return val to eax>
```

32-bit x86 ISA Function Call

int z = max(5, 10)

Making a function in x86…

1. Arguments stored on stack
2. Unconditional jump to function code
3. Update stack and base pointers
4. Execute function code
5. Store return value in known place
6. Unconditional jump back to calling code
7. Restore stack

In x86:

```
pushl $10
pushl $5
call  max
pushl %ebp
movl  %esp, %ebp
<function code executes>
<move return val to eax>
popl  %ebp
return
popl  %edx
popl  %edx
```

Implementing System Call Mechanism

Particulars vary by OS convention, processor instruction set architecture, and over time.

1. User program requests OS services
2. User program stops executing, hands execution over to OS
3. OS determines whether request is valid and allowable for user
4. OS performs service, if valid
5. OS returns a status code to user program
6. OS stops executing, hands execution back to user program

Implementing System Call Mechanism

Particulars vary by OS convention, processor instruction set architecture, and over time.

1. User program requests OS service
   - Need to specify what service and with what arguments, e.g. open() needs to know what file to open and with what permissions
   - Done by placing a specific system call number in processor register, and arguments in other registers
   - E.g. on Linux see “man 2 syscall” to see architecture calling conventions listed explicitly, see “man 2 syscalls” to see a comprehensive list of system calls

Implementing System Call Mechanism

Particulars vary by OS convention, processor instruction set architecture, and over time.

2. User program stops executing, hands execution over to OS
   - Executes a special assembly instruction to initiate the system call, see "man 2 syscall" for details. Called lots of different things but "software interrupt" and "trap" are common
   - Stops execution of user program, and transfers execution to a specific known starting point in operating system
3. OS determines whether request is valid and allowable for user
   - OS checks what service is requested by looking at processor registers, and the OS determines if the user program is allowed to do what it wants to do

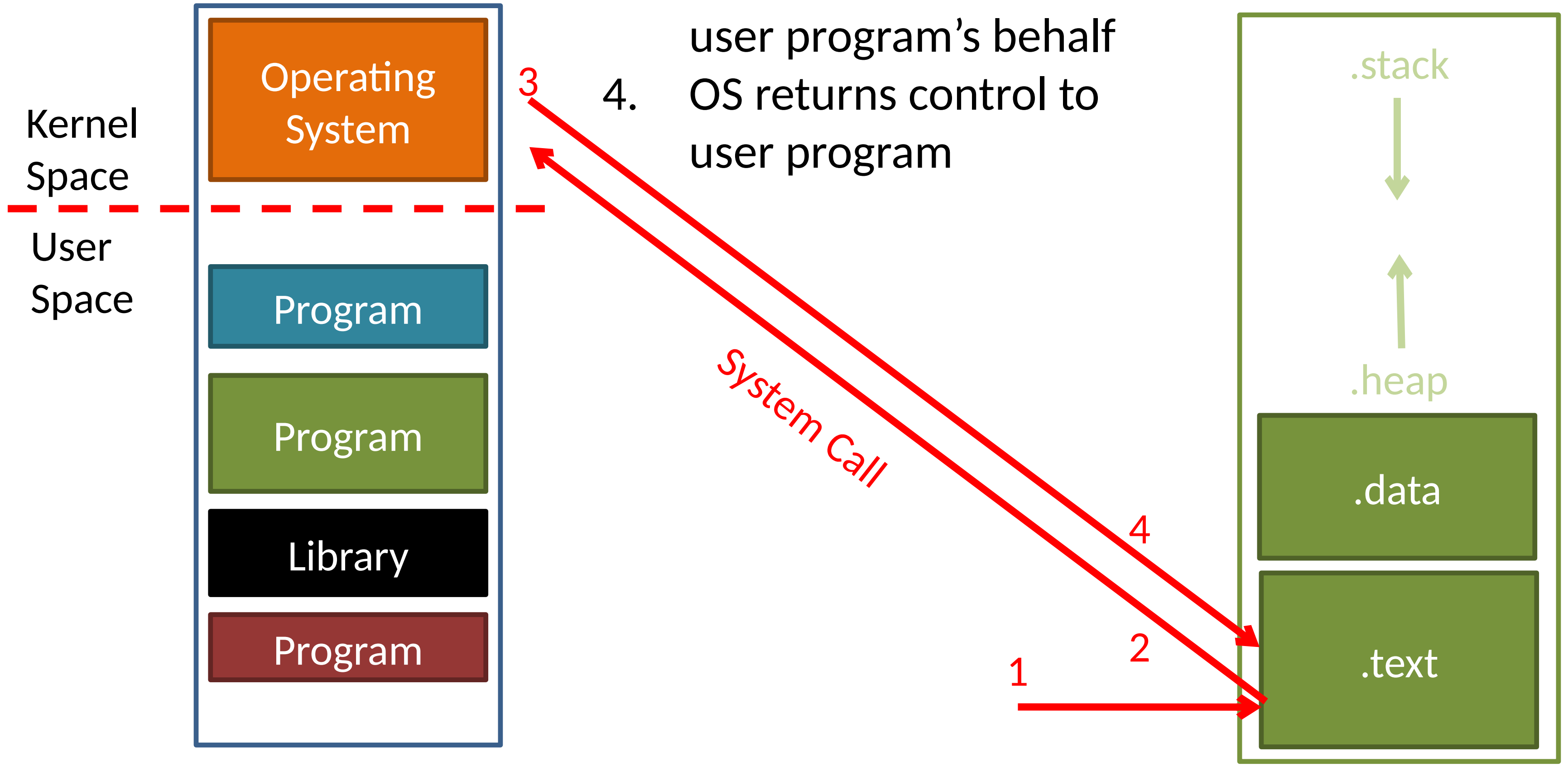
Implementing System Call Mechanism

Particulars vary by OS convention, processor instruction set architecture, and over time.

4. OS performs service, if valid
   - OS is in total control at this point, executing OS code only. This only happens if the OS system call entry point has determined that the actions are valid.
5. OS returns a status code to user program
   - Loads a single integer value in a specific register
6. OS stops executing, hands execution back to user program
   - Execution resumes in user program

System calls

1. Code executing at some place in .text section
2. Code requests system call
3. OS executes service on user program's behalf
4. OS returns control to user program



Kernel Space
User Space
Operating System
Program
Program
Library
Program
3
System Call
4
2
1
.stack
.heap
.data
.text