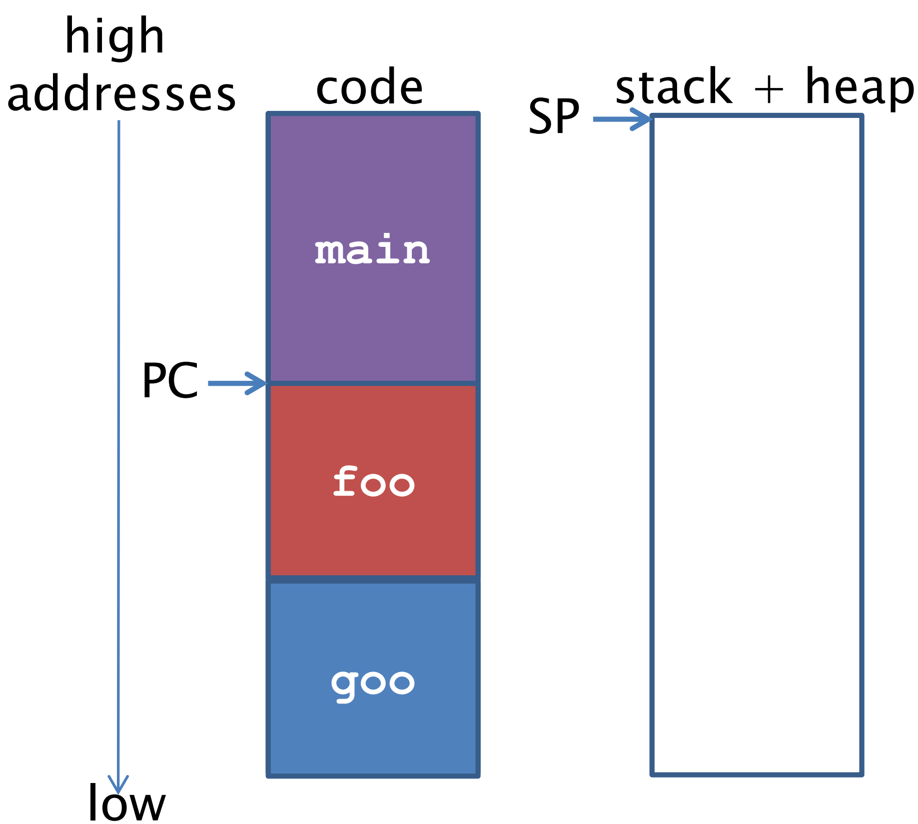
Threads

Instructor: William Zheng
Email:
william.zheng@digipen.edu
PHONE EXT: 1745

Lecture Outline

- Review on stack
- Intro to Multithreading
- Multithreading Models
- Multithreading APIs

Stack Pointer and Program Counter



Consider a code with the following functions:

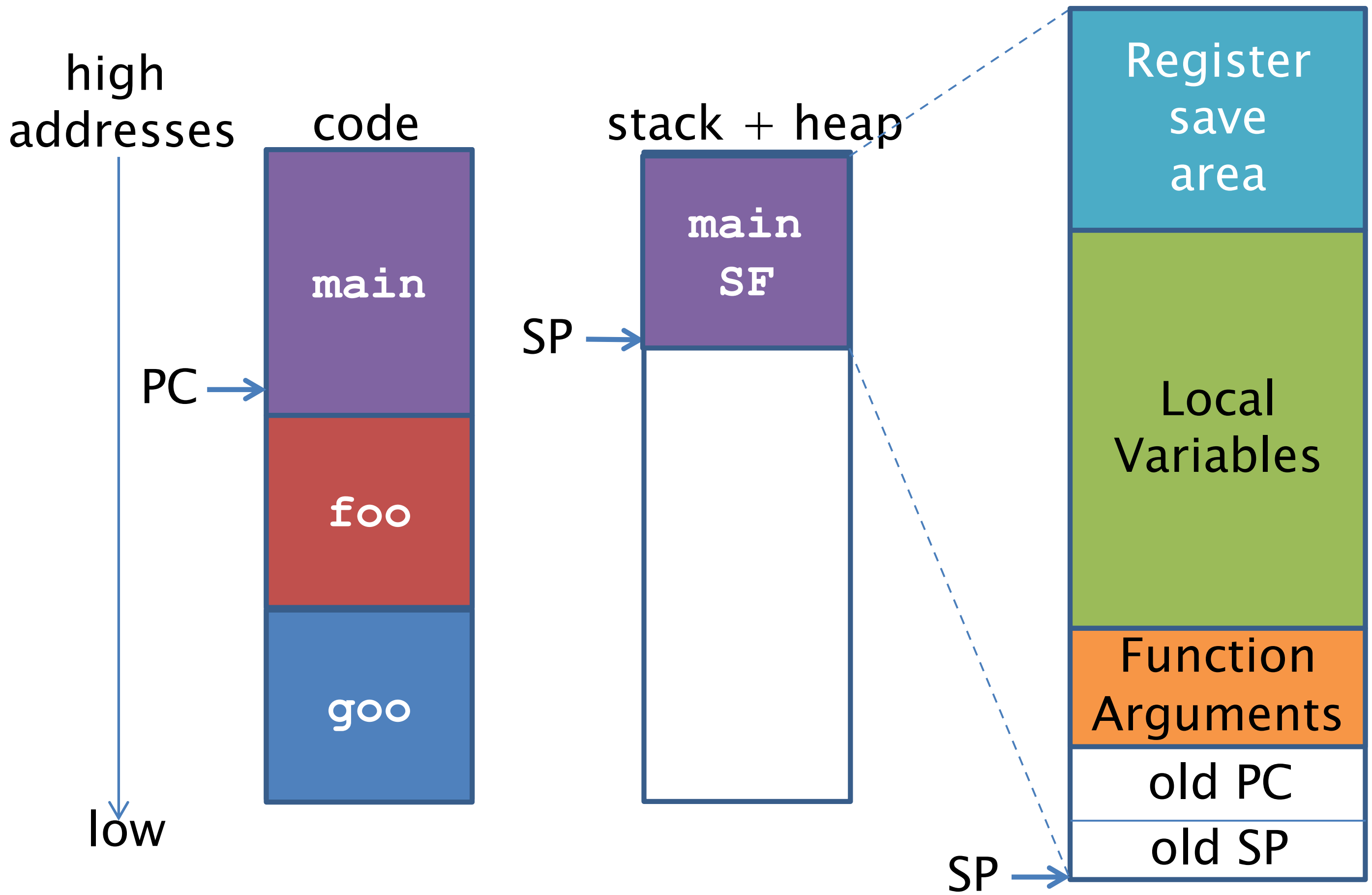
```
int main(int argc,
           char**argv);
char *foo(int, int,
  int);
int goo(double, int);
```

Assume that the functions are called:

```
main->foo->goo
```

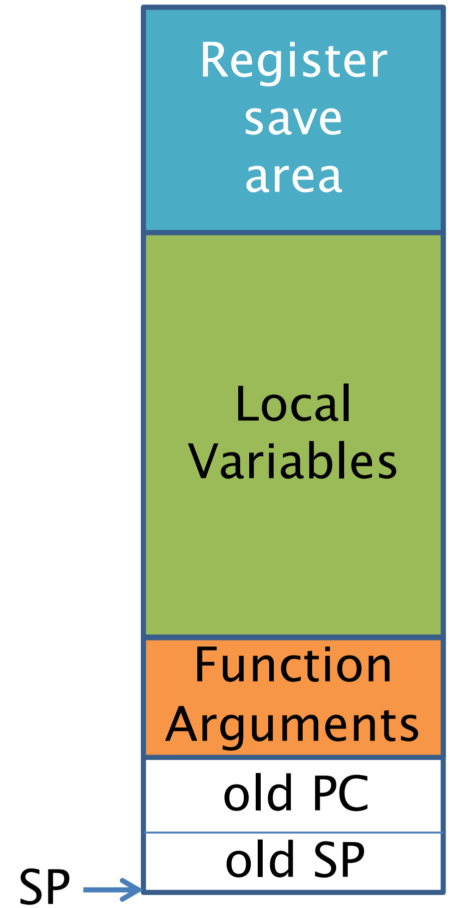
PC pointing to main
Stack is empty

Stack Frame


high
addresses
code
stack + heap
main
foo
goo
PC
main
SF
SP
low
Register
save
area
Local
Variables
Function
Arguments
old PC
old SP
SP

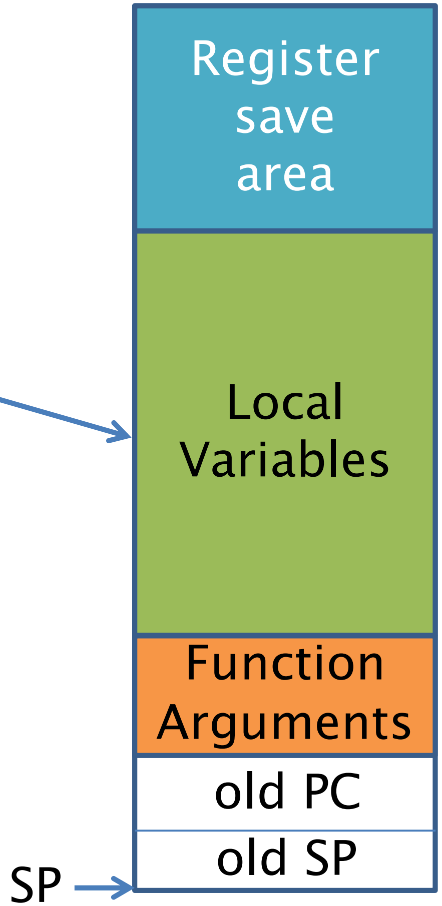
Stack Frame Organization – I

```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```



Stack Frame Organization – II

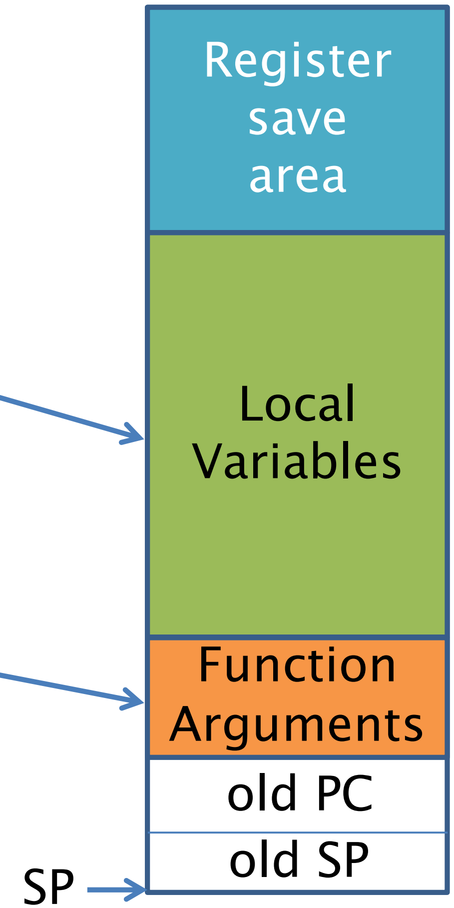
```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```



a and *d* could be in registers

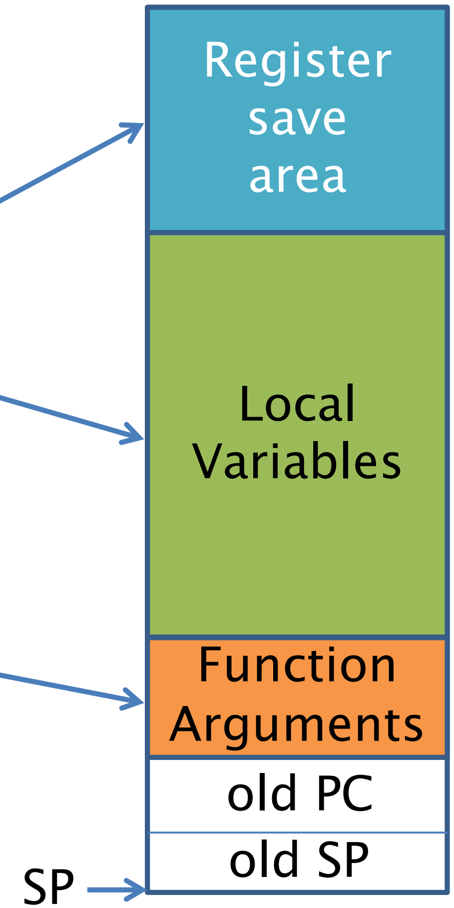
Stack Frame Organization – III

```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```

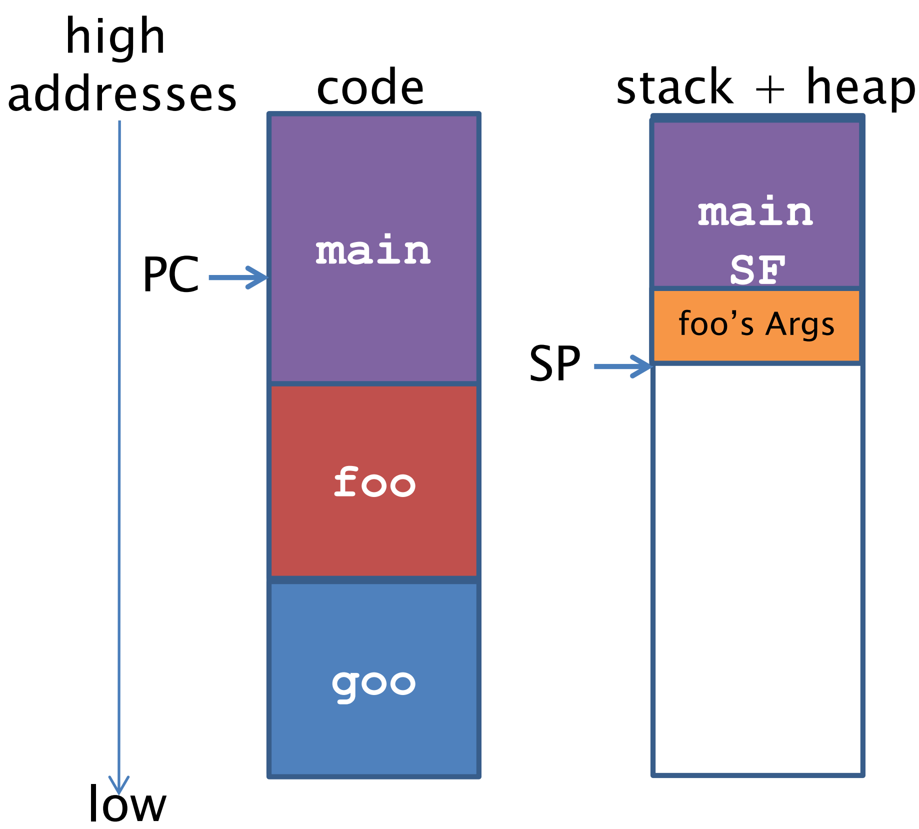


Stack Frame Organization – IV

```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```

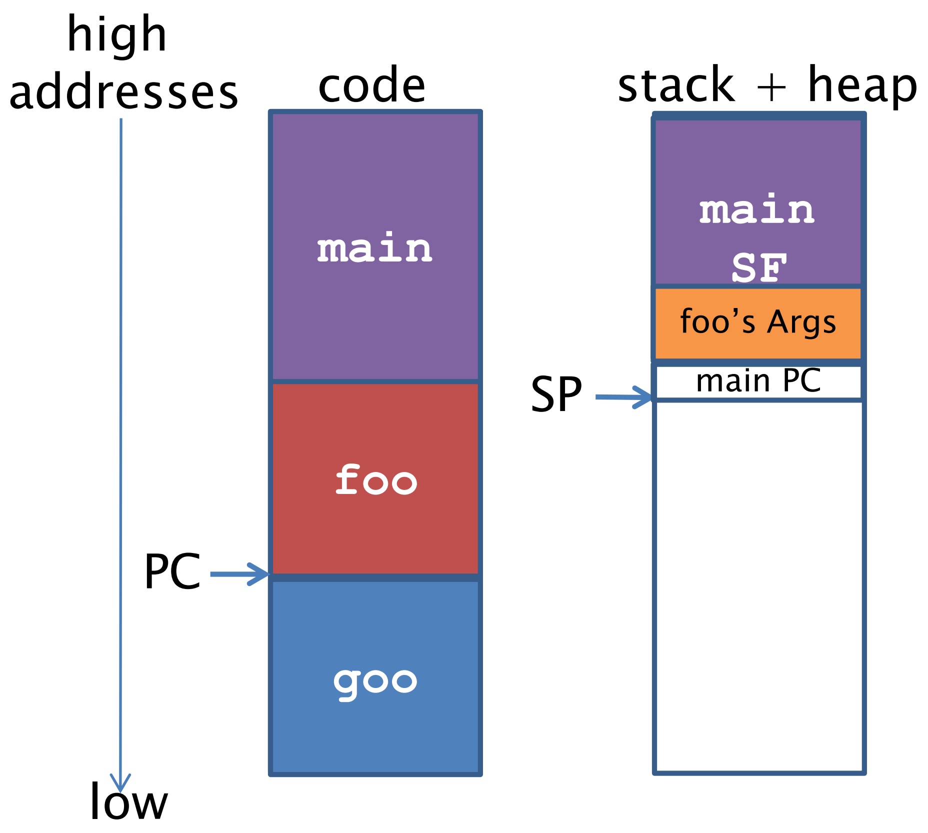


Function Call and SF Creation – I



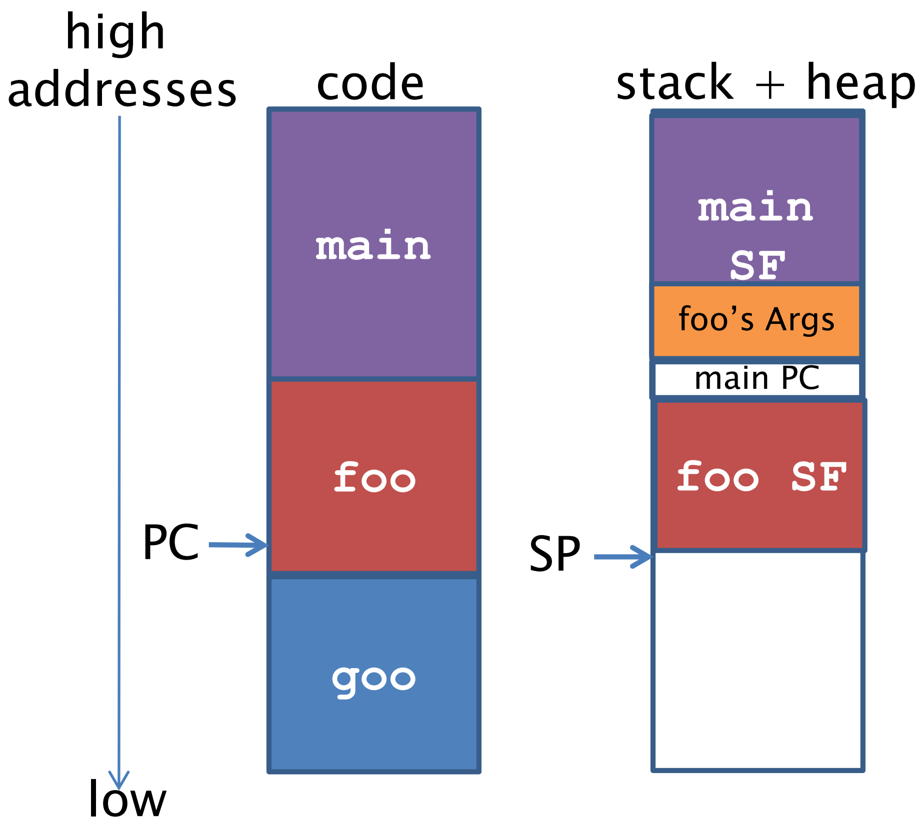
1. Push foo’s Args

Function Call and SF Creation – II



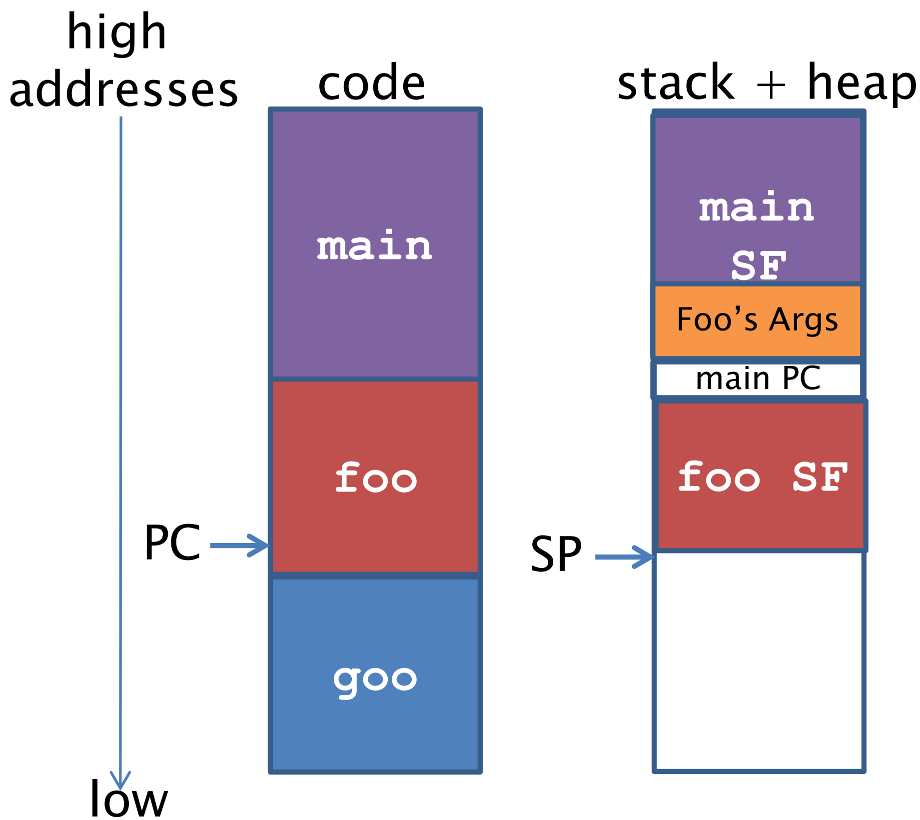
1. Push foo's Args
2. Call `foo`

Function Call and SF Creation –III



1. Push foo’s Args
2. Call `foo`
3. Save `main` SP and decrement SP

Function Call and SF Creation –IV



1. Push foo's Args
2. Call `foo`
3. Save `main` SP and decrement SP

Done in software
i.e., done by instructions generated by the compiler!

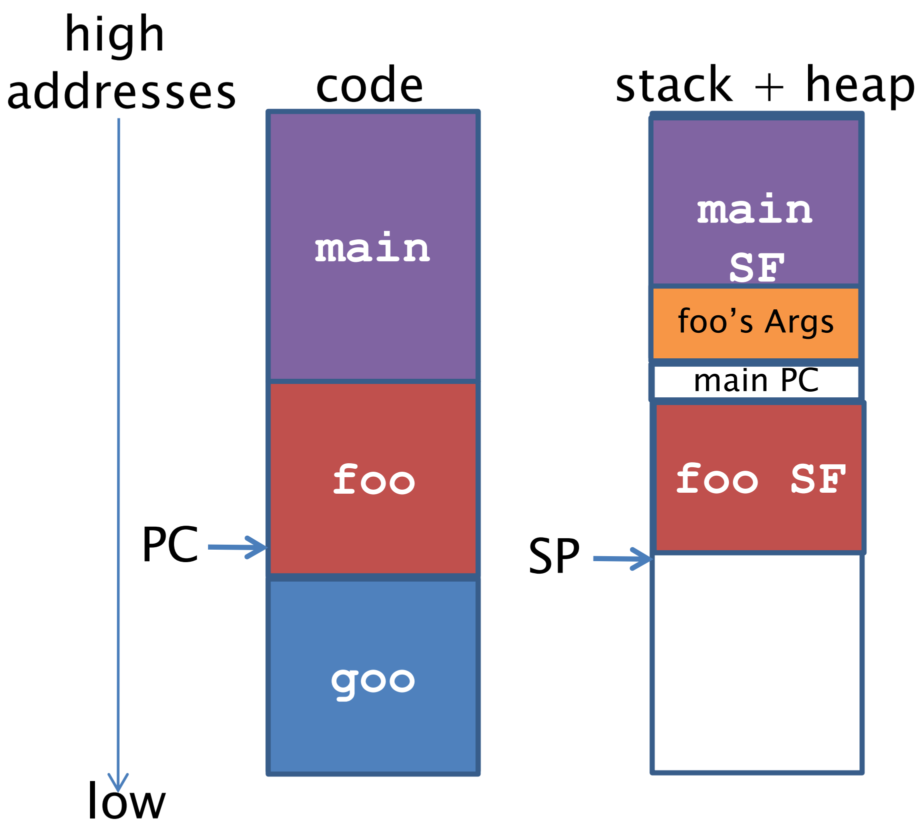
Q & A

- Does each function use the same stack frame size?
  - No. Depends on the size of local variables
- How and when is the size of stack frame determined?
  - Compiler determines by looking at the code. Compile-time.
- How is the stack frame allocated during run-time?
  - Decrementing the stack pointer

Q & A

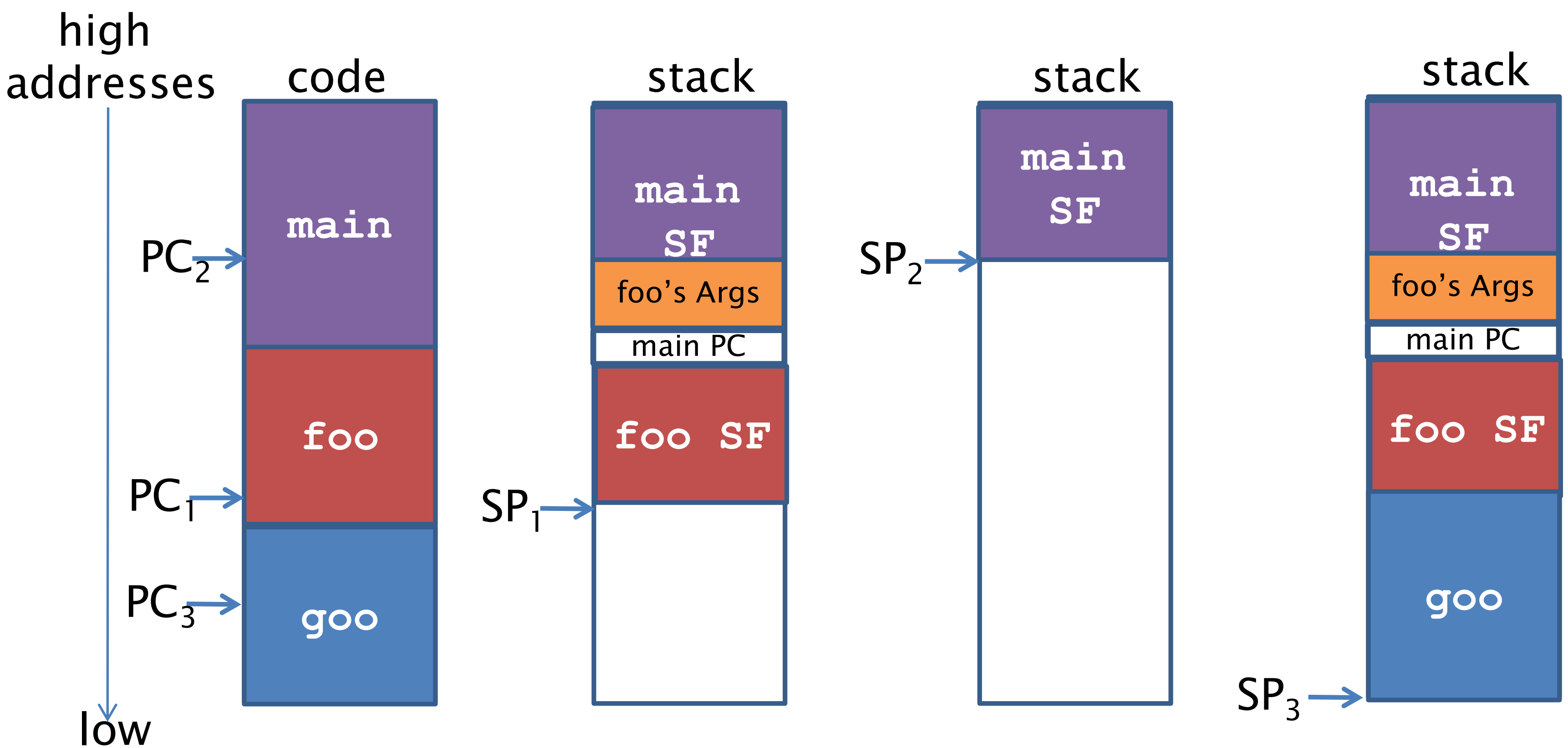
- What is the stack pointer?
  - A value stored in stack pointer register (x86/64: %esp (32bits), %rsp (64bits)) pointing to the beginning of the stack frame
- What is a program counter?
  - A value stored in program counter register (%eip (32bits), %rip (64bits))) pointing to a point in the text

Single-thread process

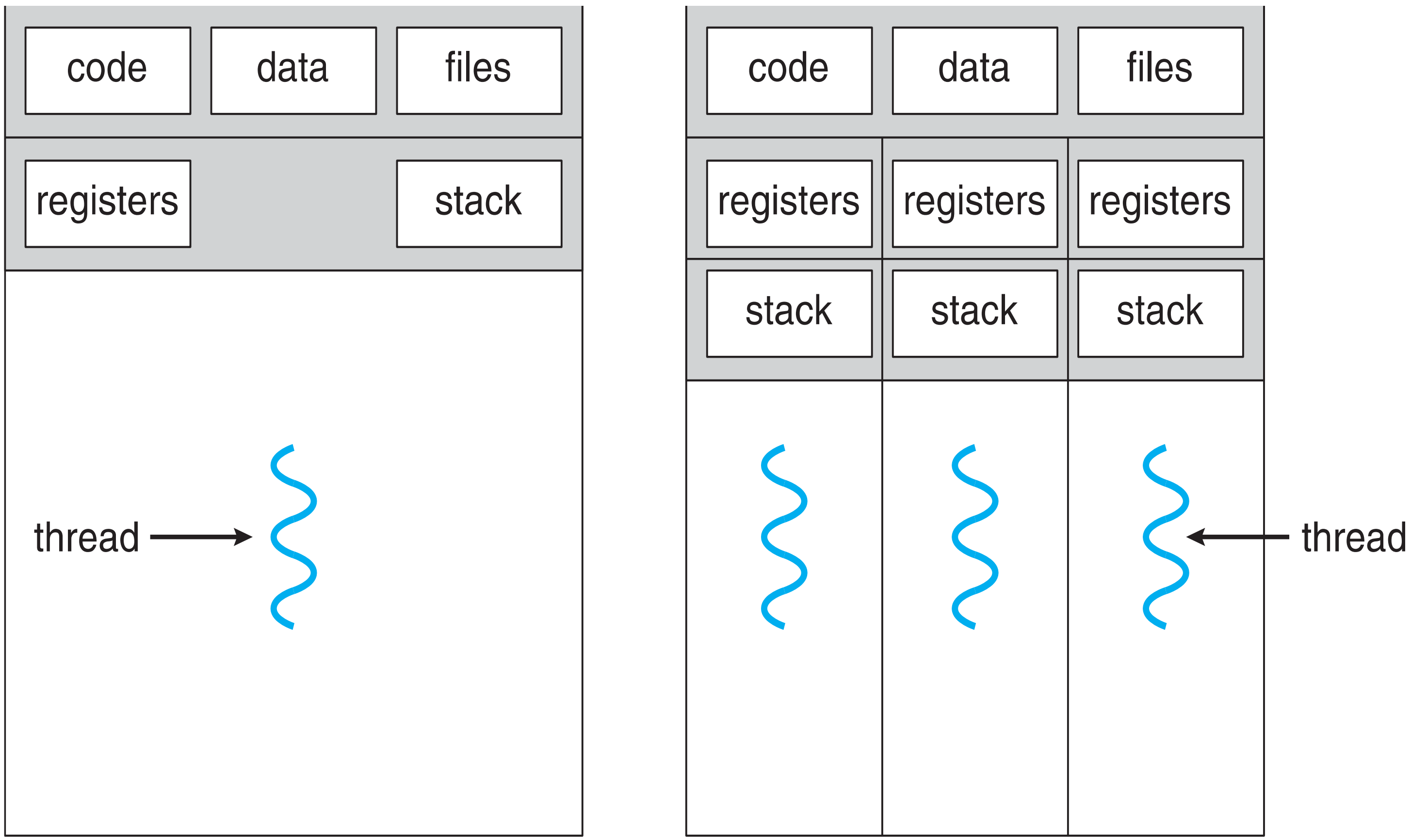


- Thread of control
  - PC
  - SP
  - Stack

Multi-threaded process



Multi-threaded versus single threaded



single-threaded process

multithreaded process

Why Multithreading?

- Responsiveness
- Resource Sharing
- Economy
- Scalability

Matrix multiplication

$$c_{ij} = \sum_{r=1}^{n} a_{ir} \times b_{rj}$$

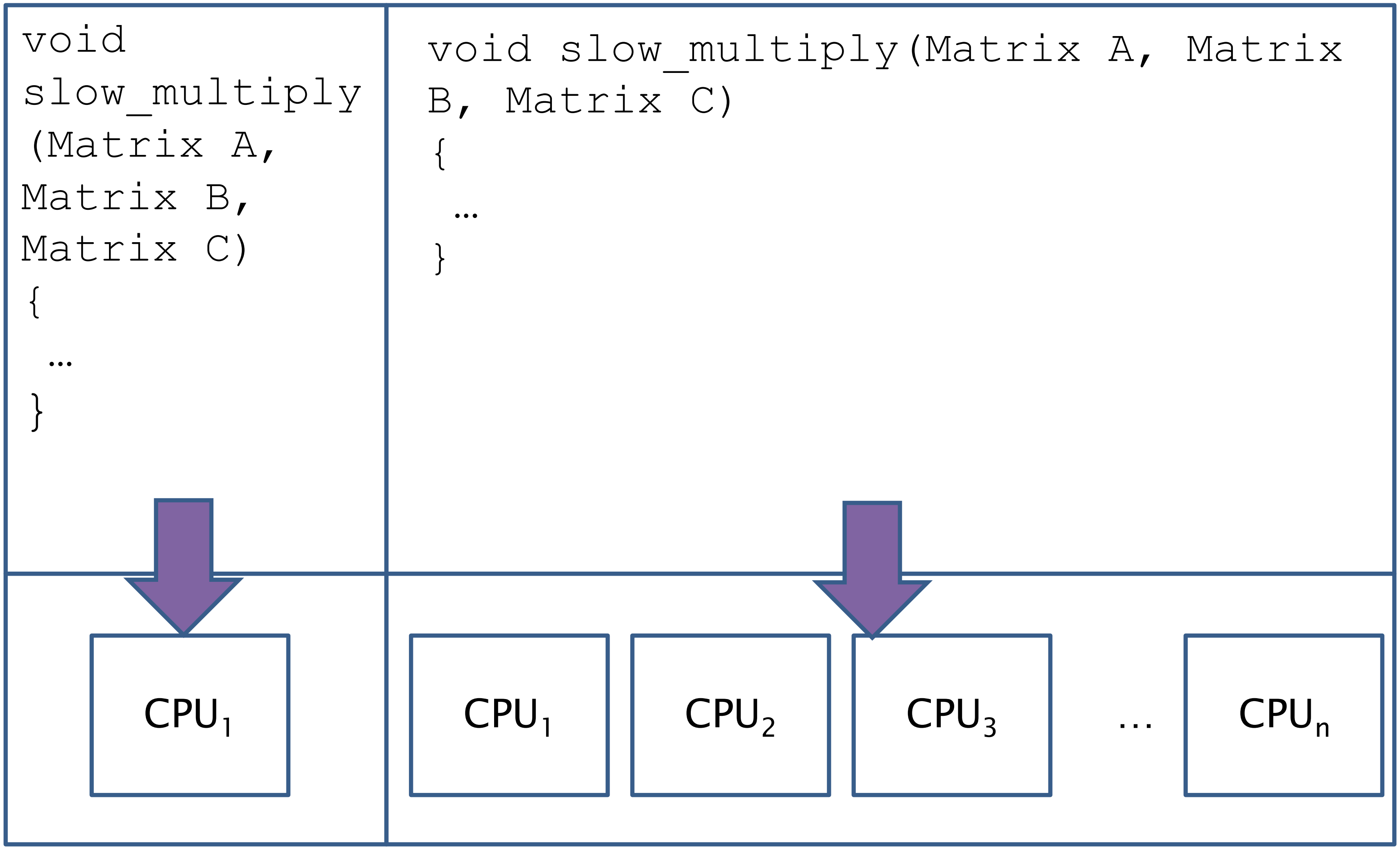
How many arithmetic operations ?

$$\begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \ldots & \ldots & \ldots \\ a_{m1} & \ldots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & \ldots & b_{1k} \\ \ldots & \ldots & \ldots \\ b_{n1} & \ldots & b_{nk} \end{pmatrix} = \begin{pmatrix} c_{11} & \ldots & c_{1k} \\ \ldots & \ldots & \ldots \\ c_{m1} & \ldots & c_{mk} \end{pmatrix}$$

An initial solution

```
void slow_multiply(Matrix A, Matrix B, Matrix C)
{
    for(int i=0; i<m; i++)
    {
        for(int j=0; j<k; j++)
        {
            int acc = 0;
            for(int r=0; r<n; r++)
            {
                acc += A[i][r]*B[r][j];
            }
            C[i][j] = acc;
        }
    }
}
```

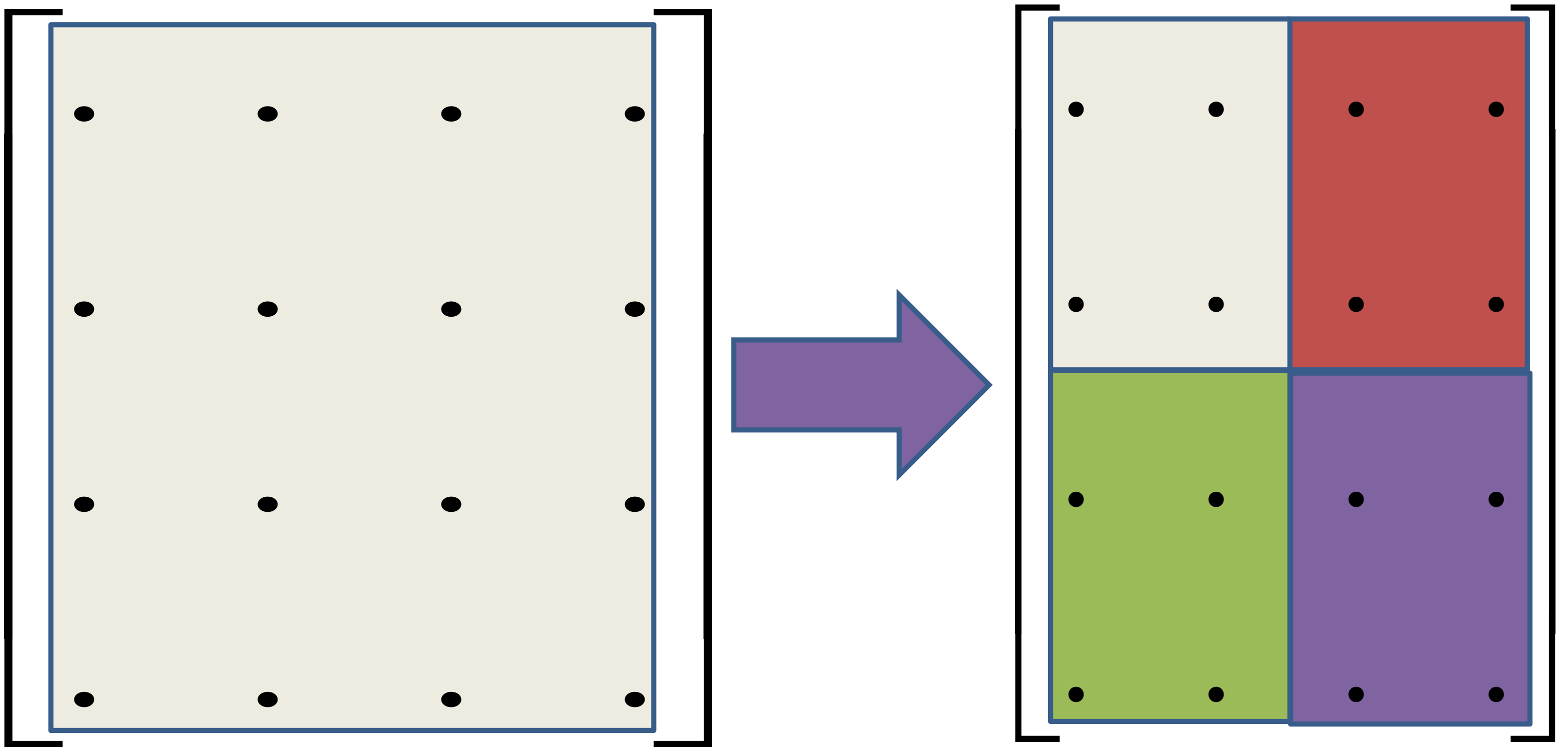
Resources Usage



Doing better

Instead of computing $C_{00}, C_{01}, C_{02}, C_{03}, \ldots.$

Why don't we …. split up the computation?



Multithreads of Multiplication

Procedure multiply(C, A, B):

•Base case: if $n = 1$, set $c_{11} \leftarrow a_{11} \times b_{11}$ (or multiply a small block matrix).

•Otherwise, allocate space for a new matrix T of shape $n \times n$, then:

- Partition A into A_{11}, A_{12}, A_{21}, A_{22}.
- Partition B into B_{11}, B_{12}, B_{21}, B_{22}.
- Partition C into C_{11}, C_{12}, C_{21}, C_{22}.
- Partition T into T_{11}, T_{12}, T_{21}, T_{22}.
- Parallel execution:
  - *Fork* multiply(C_{11}, A_{11}, B_{11}).
  - *Fork* multiply(C_{12}, A_{11}, B_{12}).
  - *Fork* multiply(C_{21}, A_{21}, B_{11}).
  - *Fork* multiply(C_{22}, A_{21}, B_{12}).
  - *Fork* multiply(T_{11}, A_{12}, B_{21}).
  - *Fork* multiply(T_{12}, A_{12}, B_{22}).
  - *Fork* multiply(T_{21}, A_{22}, B_{21}).
  - *Fork* multiply(T_{22}, A_{22}, B_{22}).
- *Join* (wait for parallel forks to complete).
- add(C, T).
- Deallocate T.

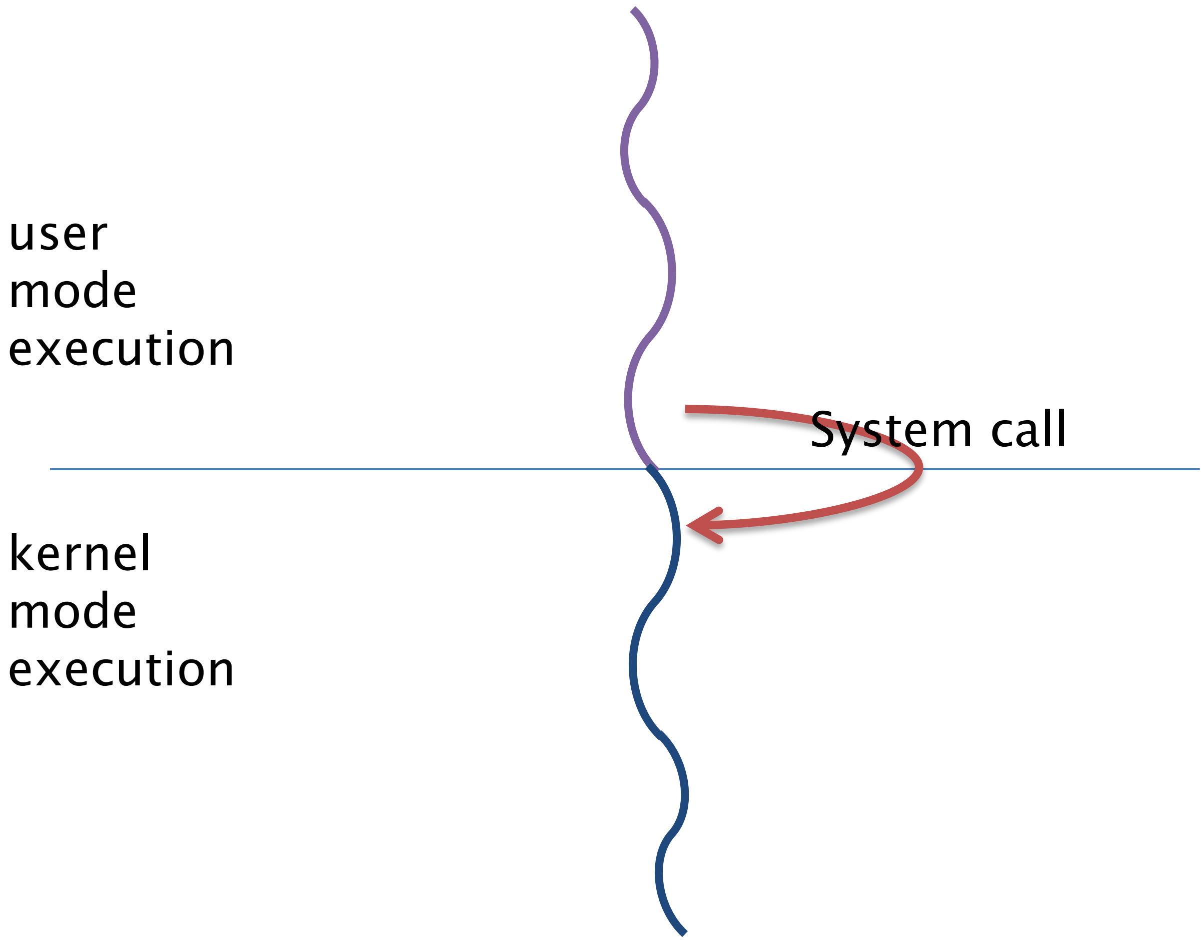
Two kinds of stack

- User Stack
  - Used for user-level programs
- Kernel Stack
  - Used by system-calls

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X

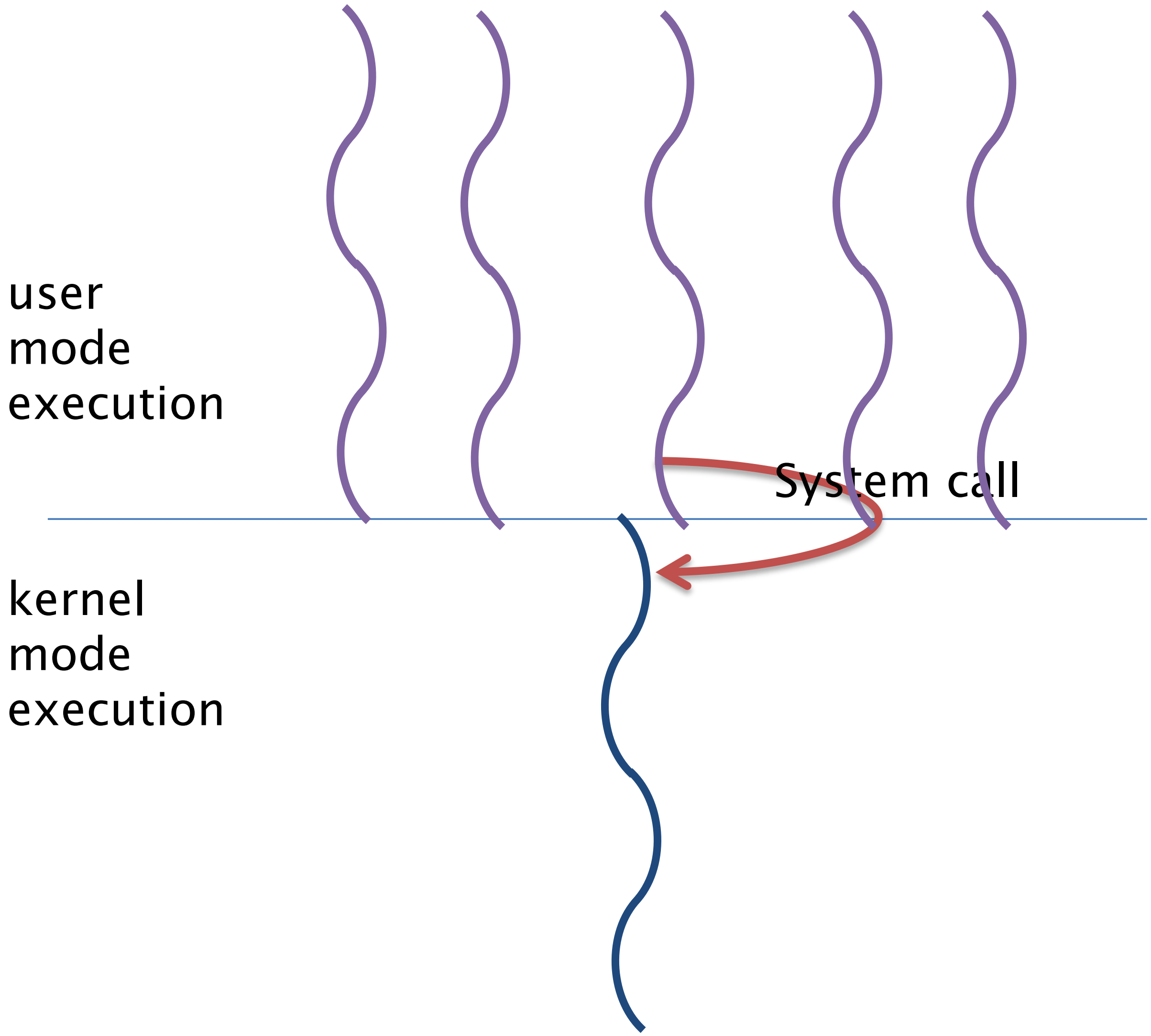
Single thread execution



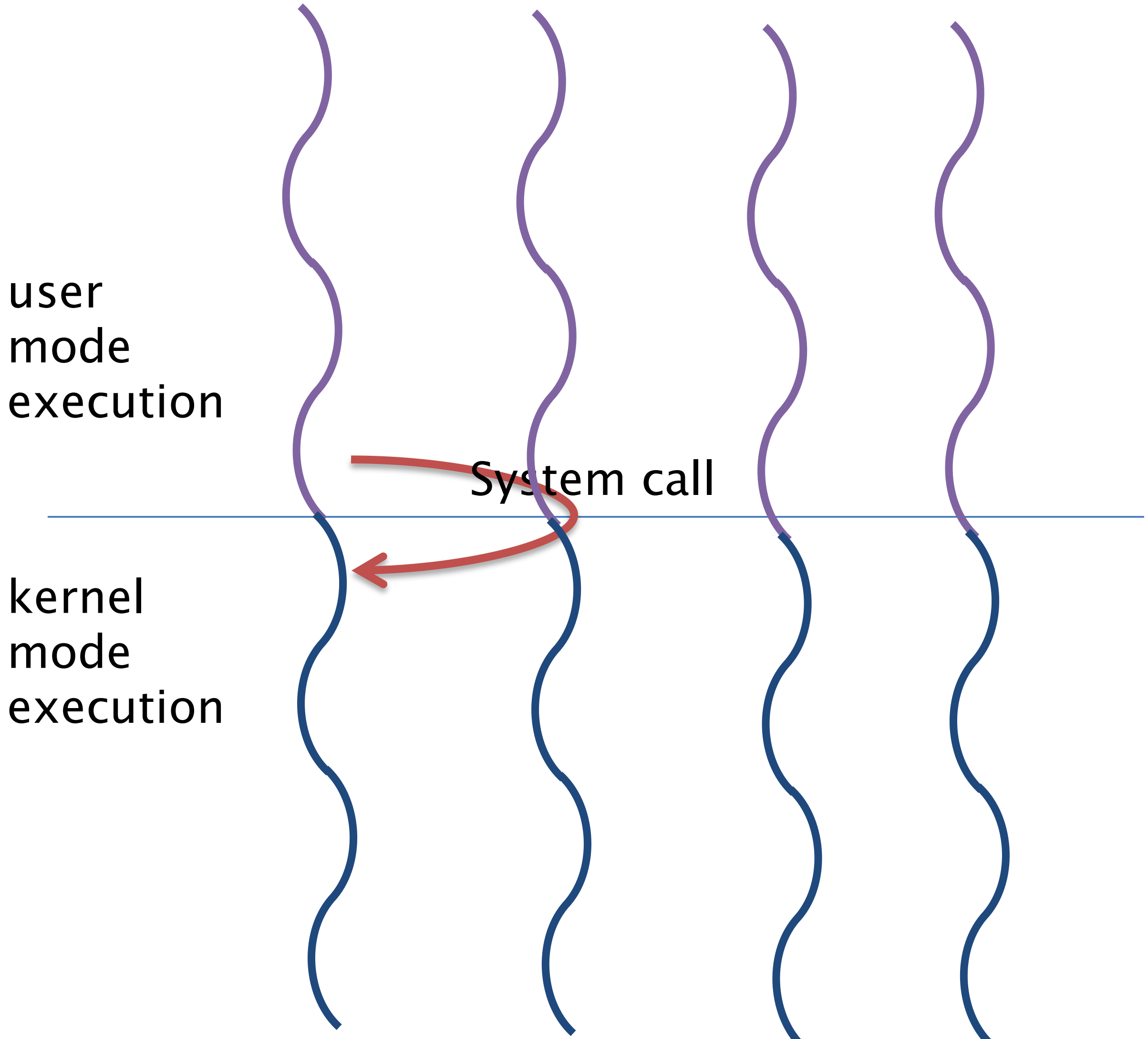
Multithreading Model

- Ratio of User Level Threads to Kernel Level Threads in a Process
  - M : 1
  - 1 : 1
  - M : N

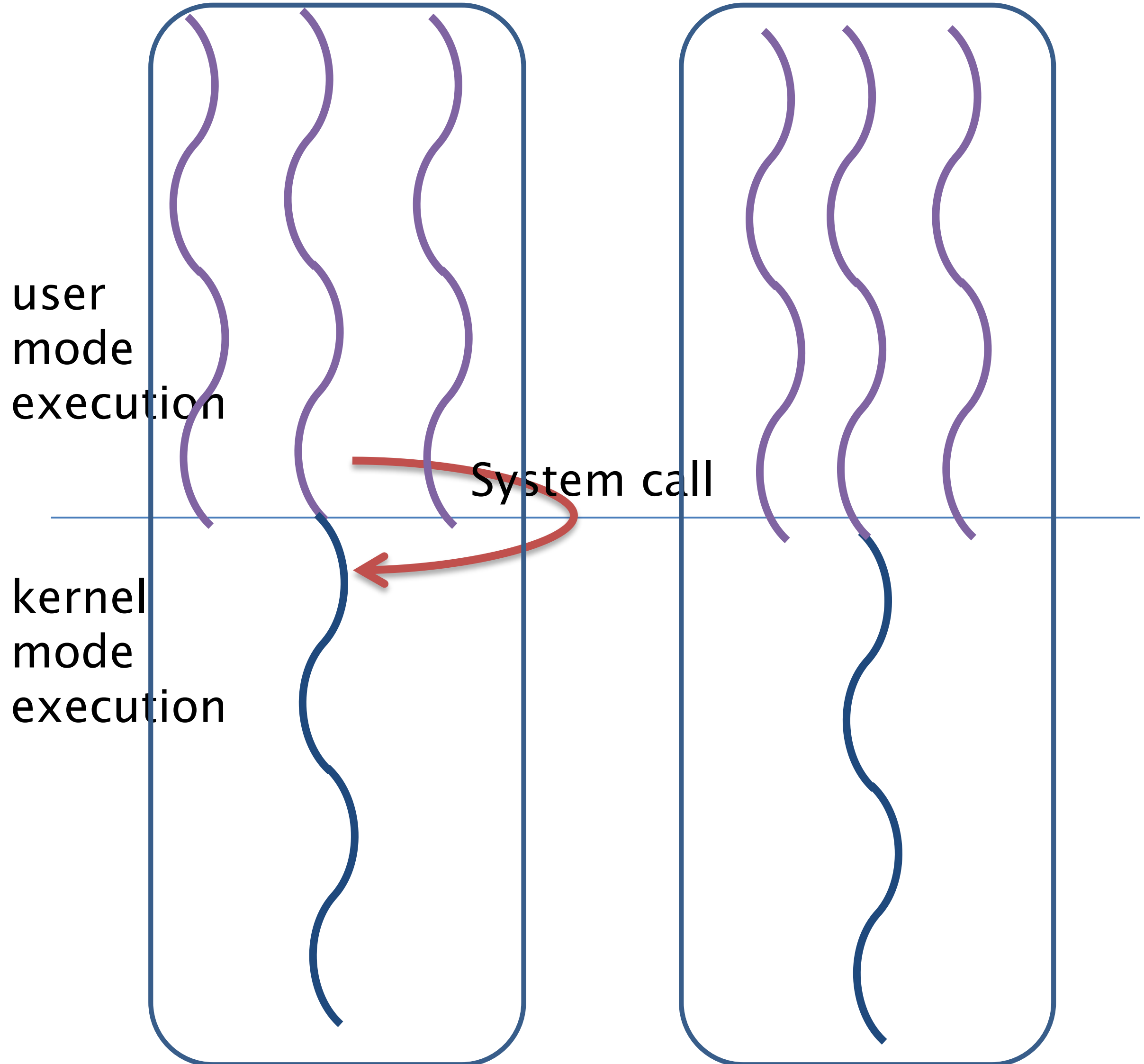
M : 1 Model



1:1 Thread Execution



M:N Thread Execution



Threads API

- Basic
  - Thread Creation
  - Thread Joining & Exit
- Advanced
  - ProcessorAffinity
  - Yield CPU

Threads Creation

- Thread ID
- Passing Arguments to thread
- Starting function for thread
  - pthread_create in Linux/Unix
  - CreateThread in Win32

Threads Joining and Exit

- Linux
  - pthread_join and pthread_exit
- Win32
  - WaitForSingleObject
  - ExitThread