Systems and Networking I

Applied Computer Science and Artificial Intelligence

2025-2026

Gabriele Tolomei

Computer Science Department

Sapienza Università di Roma

tolomei@di.uniroma1.it

The Big Picture So Far

- We have presented a number of services the OS provides to
  - abstract from actual physical (HW) resources
  - ease the interaction between users and HW resources

The Big Picture So Far

- We have presented a number of services the OS provides to
  - abstract from actual physical (HW) resources
  - ease the interaction between users and HW resources
- Different OS designs depending on how those services are implemented
  - monolithic, layered, microkernel, hybrid, etc.

Part II: Process Management

Program vs. Process

- A **program** is an executable file which resides on the persistent memory (e.g., disk),
  - contains only the set of instructions to accomplish a specific job
  - e.g., the ls program is an executable file stored at /bin/ls on the disk of a UNIX-like OS

Program vs. Process

- A **program** is an executable file which resides on the persistent memory (e.g., disk),
  - contains only the set of instructions to accomplish a specific job
  - e.g., the ls program is an executable file stored at /bin/ls on the disk of a UNIX-like OS
- A **process** is a particular instance of a program when loaded to main memory
  - e.g., multiple instances of the ls program above, thus multiple processes for the same program

Program vs. Process

- A **program** is an executable file which resides on the persistent memory (e.g., disk),
  - contains only the set of instructions to accomplish a specific job
  - e.g., the ls program is an executable file stored at /bin/ls on the disk of a UNIX-like OS
- A **process** is a particular instance of a program when loaded to main memory
  - e.g., multiple instances of the ls program above, thus multiple processes for the same program

program → "static/passive" vs. process → "dynamic/active"

Process

- A **process** is the OS abstraction of a running program (unit of execution)

Process

- A **process** is the OS abstraction of a running program (unit of execution)
- Process is dynamic, whilst a program is static (code and data only)

Process

- A **process** is the OS abstraction of a running program (unit of execution)
- Process is dynamic, whilst a program is static (code and data only)
- Several processes may run the same program (e.g., multiple Google Chrome instances) but each has its own state

Process

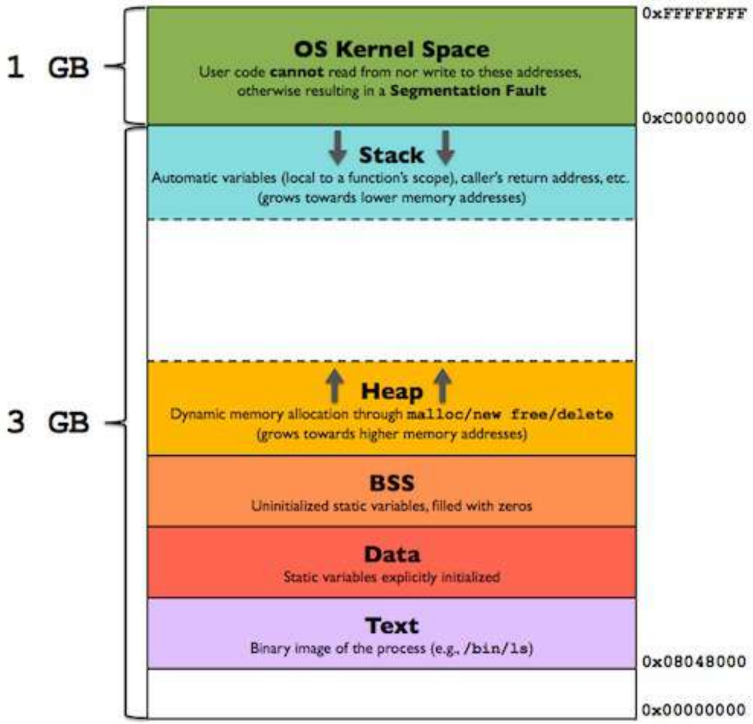
- A **process** is the OS abstraction of a running program (unit of execution)
- Process is dynamic, whilst a program is static (code and data only)
- Several processes may run the same program (e.g., multiple Google Chrome instances) but each has its own state
- A process executes one instruction at a time, sequentially

OS Process Management

- How are processes represented in the OS?
- What are the possible states a process may be in and how the system moves from one state to another?
- How are processes created in the OS?
- How do processes communicate with each other?

Process: Virtual Address Space Layout

The OS gives the same amount of virtual address space to each process

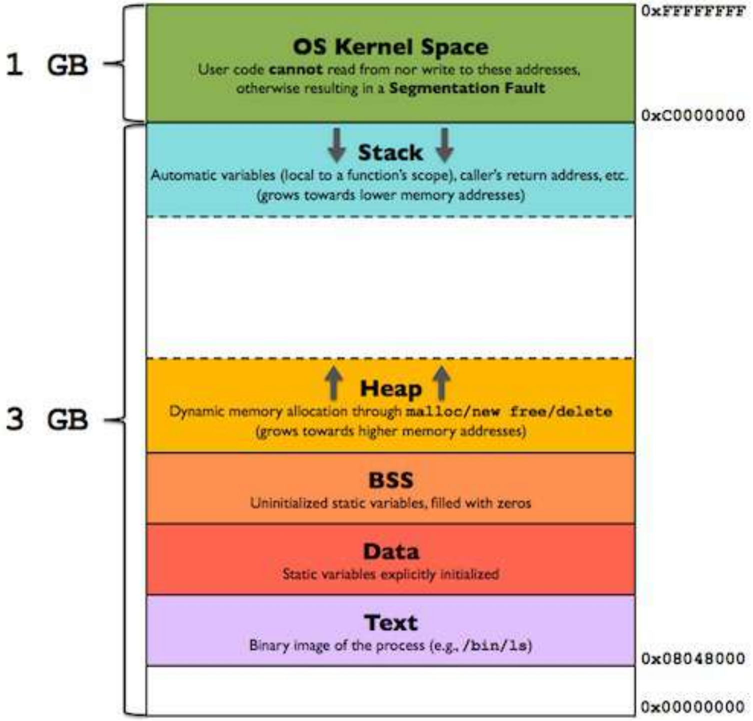


0xFFFFFFFF
1 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc.
(grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete
(grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

Process: Virtual Address Space Layout

The OS gives the same amount of virtual address space to each process

The virtual address space is an abstraction of the physical memory address space

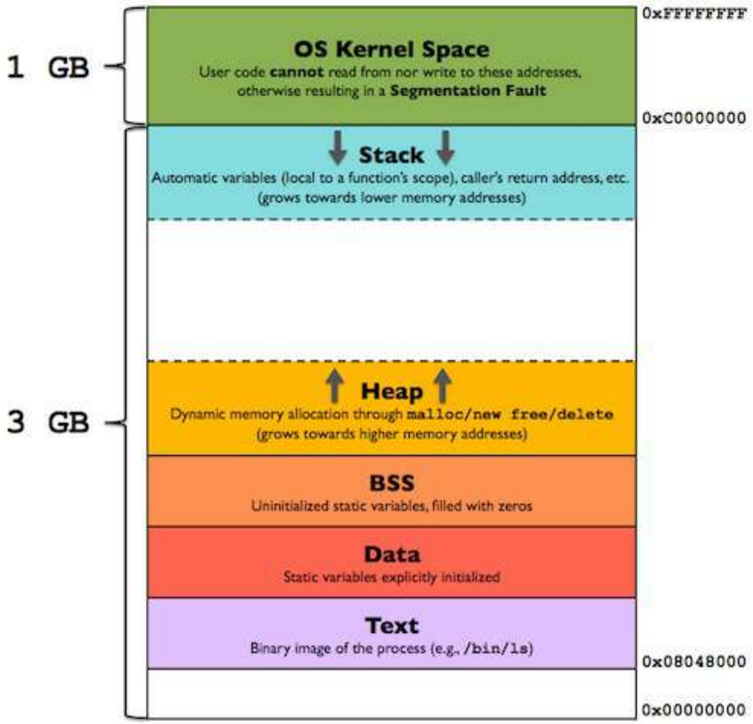

1 GB
3 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000

Process: Virtual Address Space Layout

The OS gives the same amount of virtual address space to each process

The virtual address space is an abstraction of the physical memory address space

The range of valid virtual addresses that a process can generate is machine-dependent


0xFFFFFFFF
1 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc.
(grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete
(grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

Process: Virtual Address Space Layout

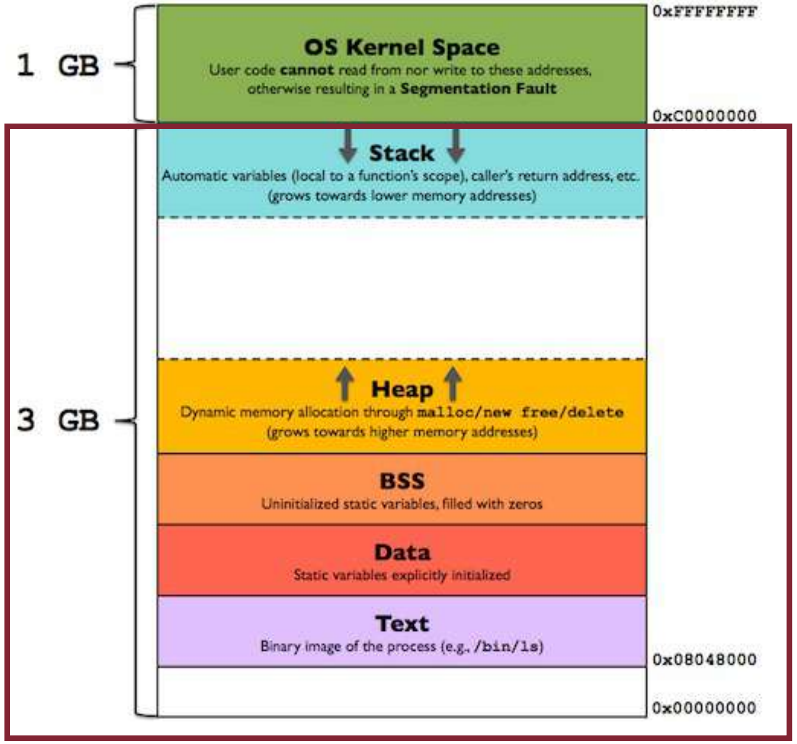
The OS gives the same amount of virtual address space to each process

The virtual address space is an abstraction of the physical memory address space

The range of valid virtual addresses that a process can generate is machine-dependent

For example, on a 32-bit architecture, the virtual addresses range from 0 to 2^{32} -1

(with the exception of some addresses reserved for the OS kernel)



1 GB

3 GB

0xFFFFFFFF

OS Kernel Space

User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**

0xC0000000

Stack

Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)

Heap

Dynamic memory allocation through `malloc/new free/delete` (grows towards higher memory addresses)

BSS

Uninitialized static variables, filled with zeros

Data

Static variables explicitly initialized

Text

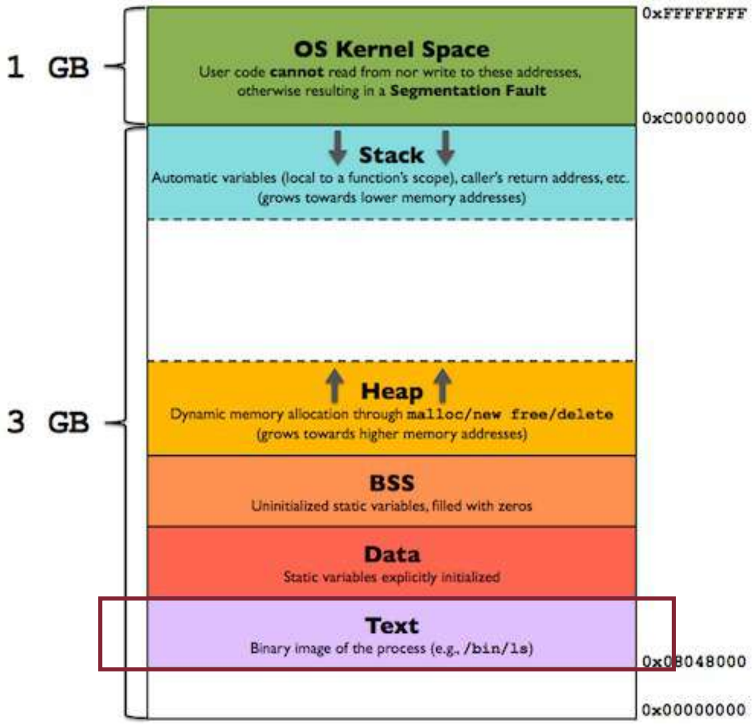
Binary image of the process (e.g., `/bin/ls`)

0x08048000

0x00000000

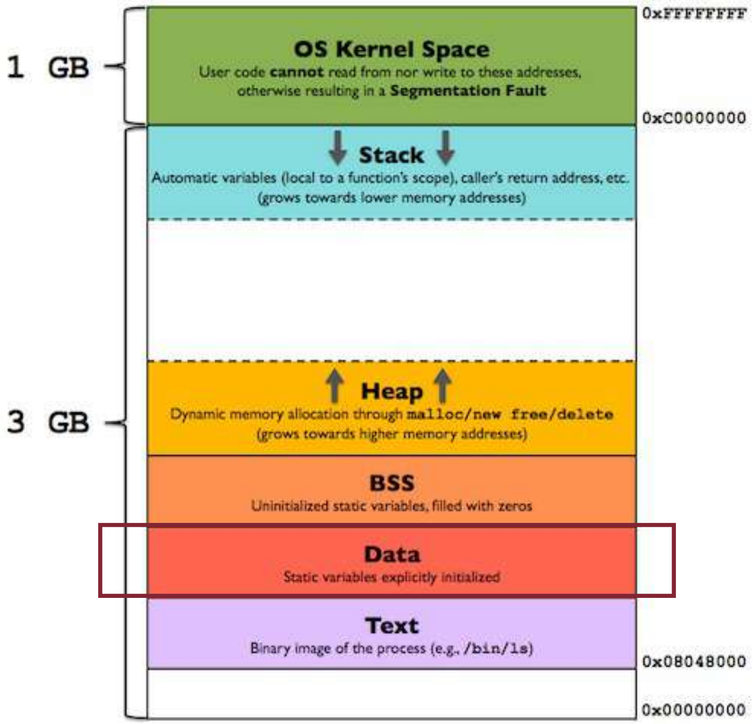
Process: Virtual Address Space Layout

- Text → contains executable instructions


1 GB
3 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through `malloc/new free/delete` (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., `/bin/ls`)
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000

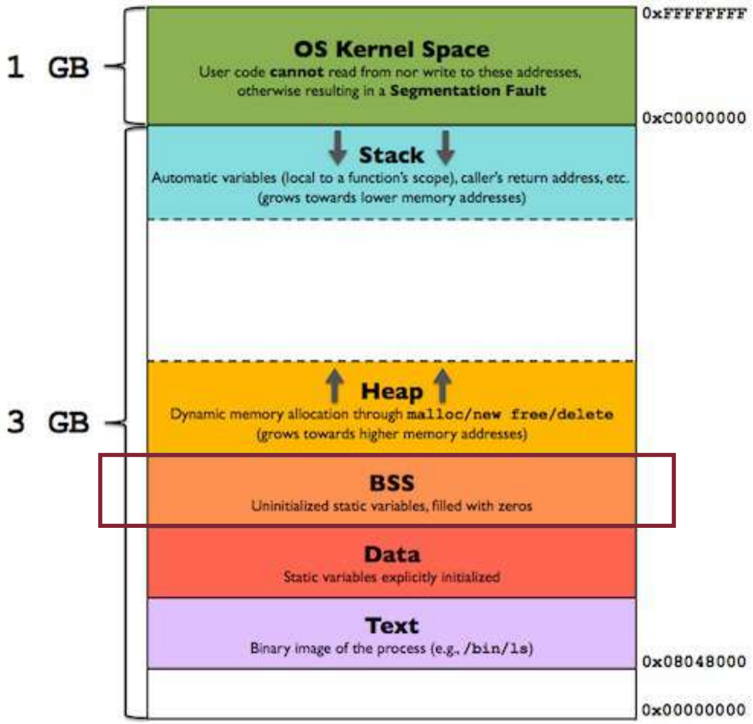
Process: Virtual Address Space Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)


0xFFFFFFFF
1 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

Process: Virtual Address Space Layout

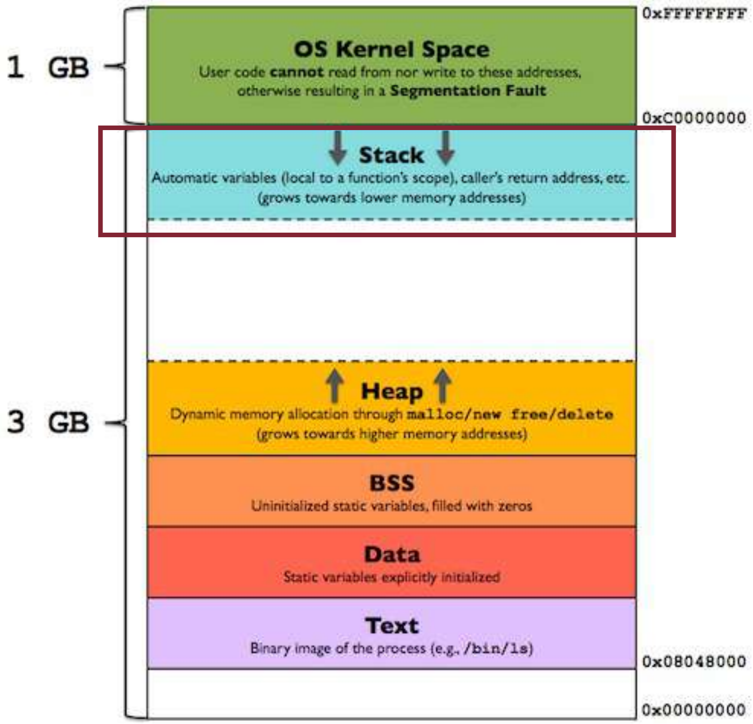
- Text → contains executable instructions
- Data → global and static variable (initialized)
- BSS → global and static variable (uninitialized or initialized to 0)



0xFFFFFFFF
1 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

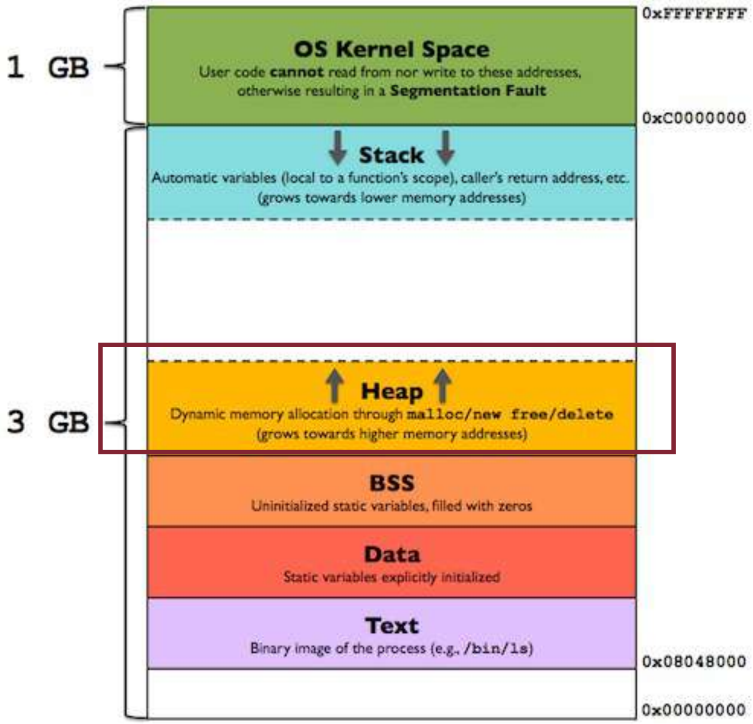
Process: Virtual Address Space Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)
- BSS → global and static variable (uninitialized or initialized to 0)
- Stack → LIFO structure used to store all the data needed by a function call (stack frame)


1 GB
3 GB
0xFFFFFFFF
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc.
(grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete
(grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

Process: Virtual Address Space Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)
- BSS → global and static variable (uninitialized or initialized to 0)
- Stack → LIFO structure used to store all the data needed by a function call (stack frame)
- Heap → used for dynamic allocation



1 GB
3 GB
0xFFFFFFFF
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc.
(grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete
(grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

Program vs. Process: Example

Program

```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```

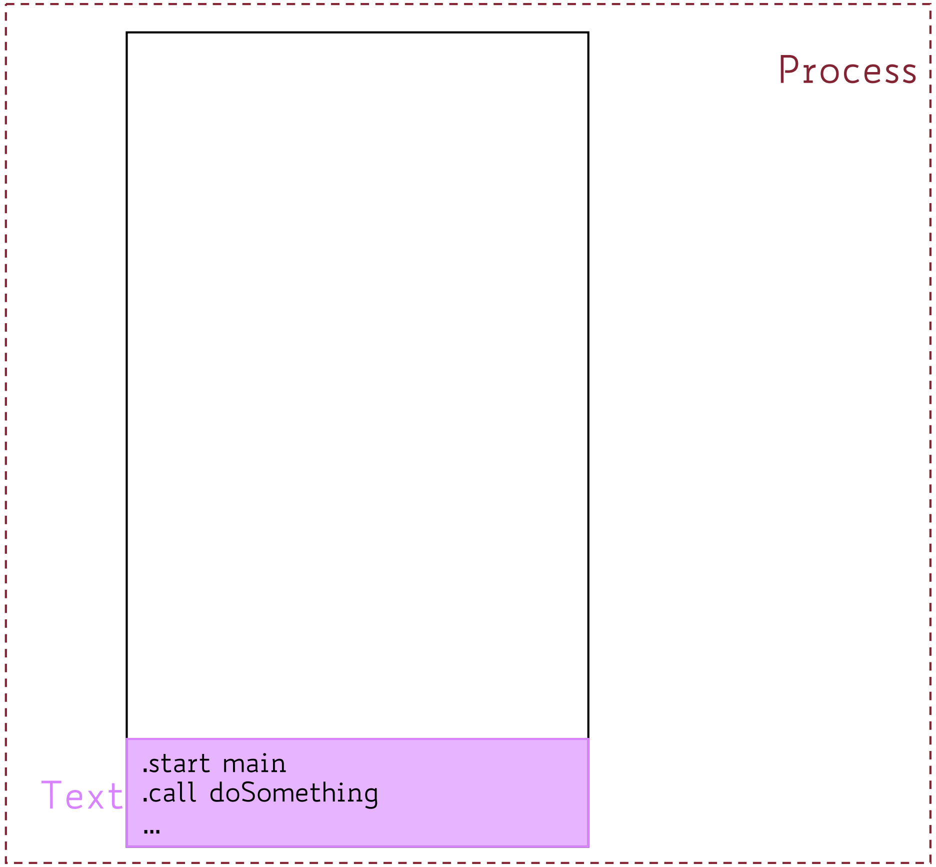
Program vs. Process: Example

Program

```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



```
.start main
.call doSomething
…
```

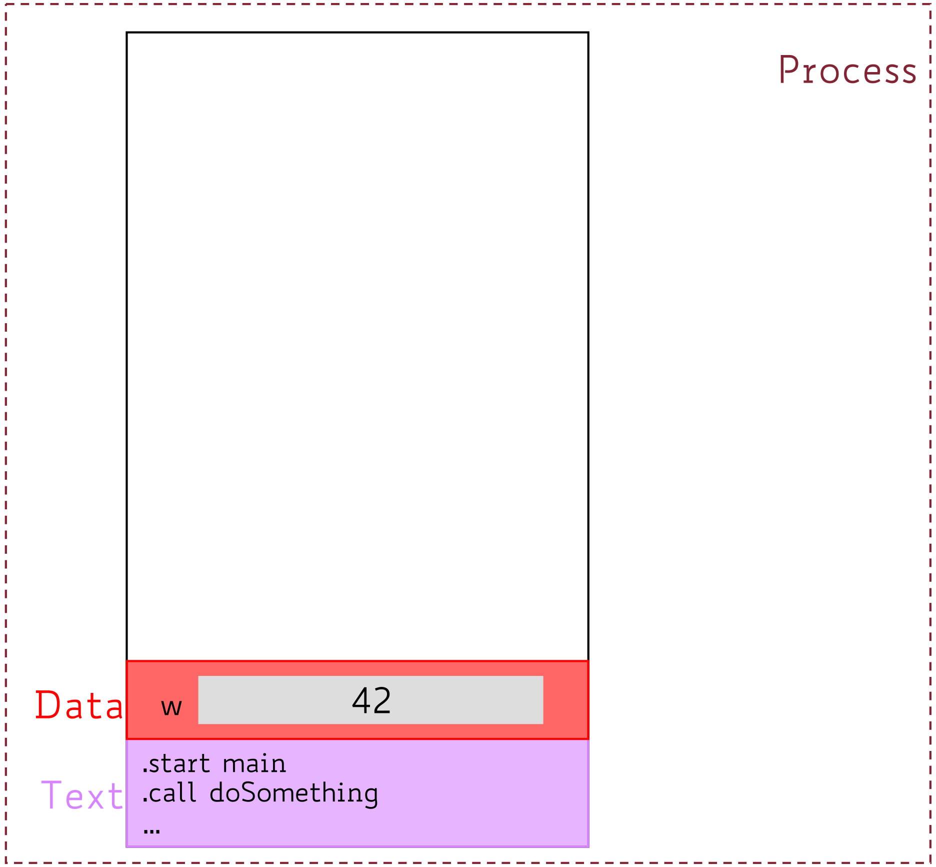
Program vs. Process: Example

Program

```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



```
.start main
.call doSomething
…
```

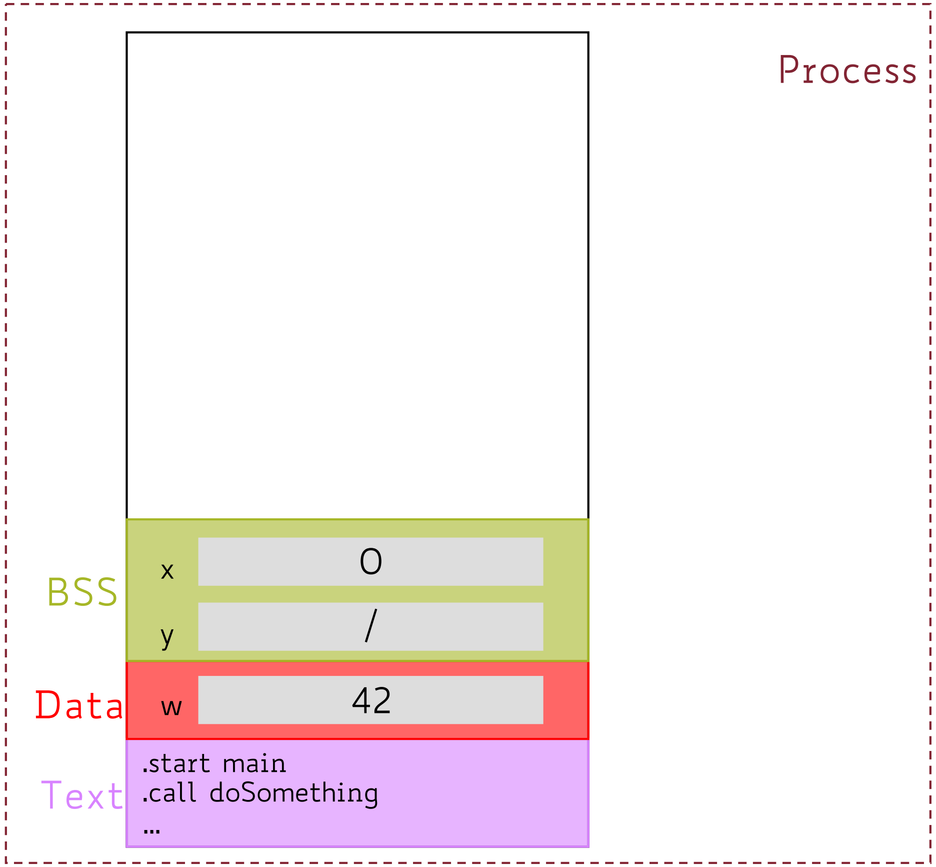
Program vs. Process: Example

Program

```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



```
.start main
.call doSomething
…
```

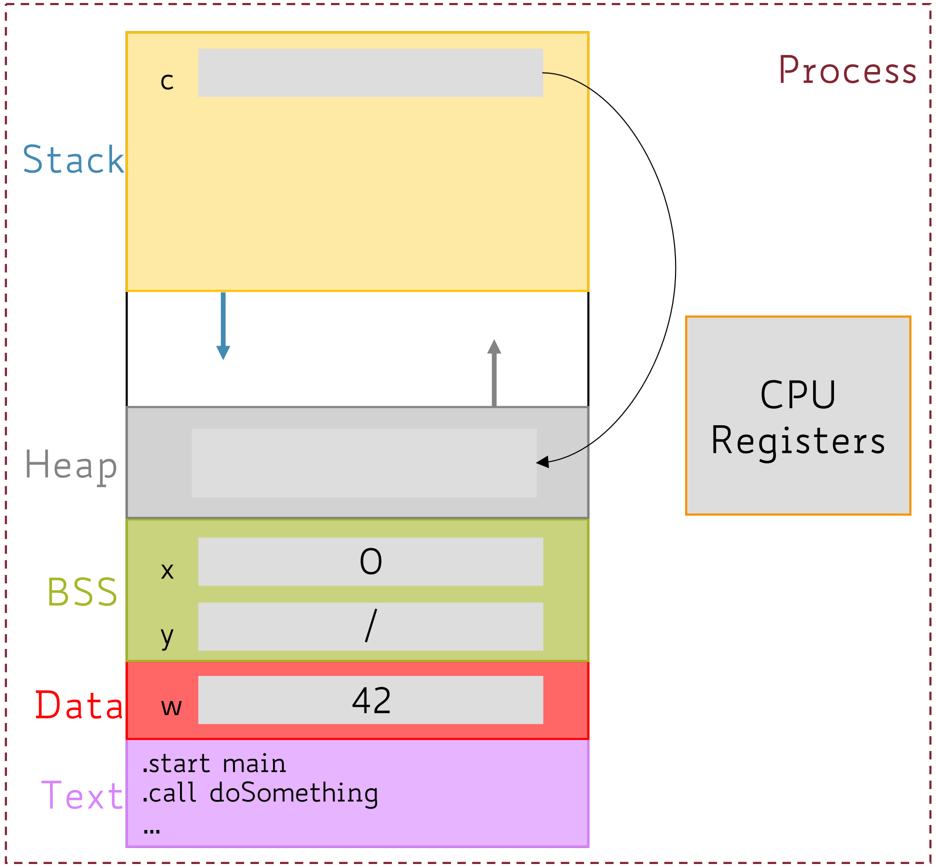
Program vs. Process: Example

Program

```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



```
.start main
.call doSomething
…
```

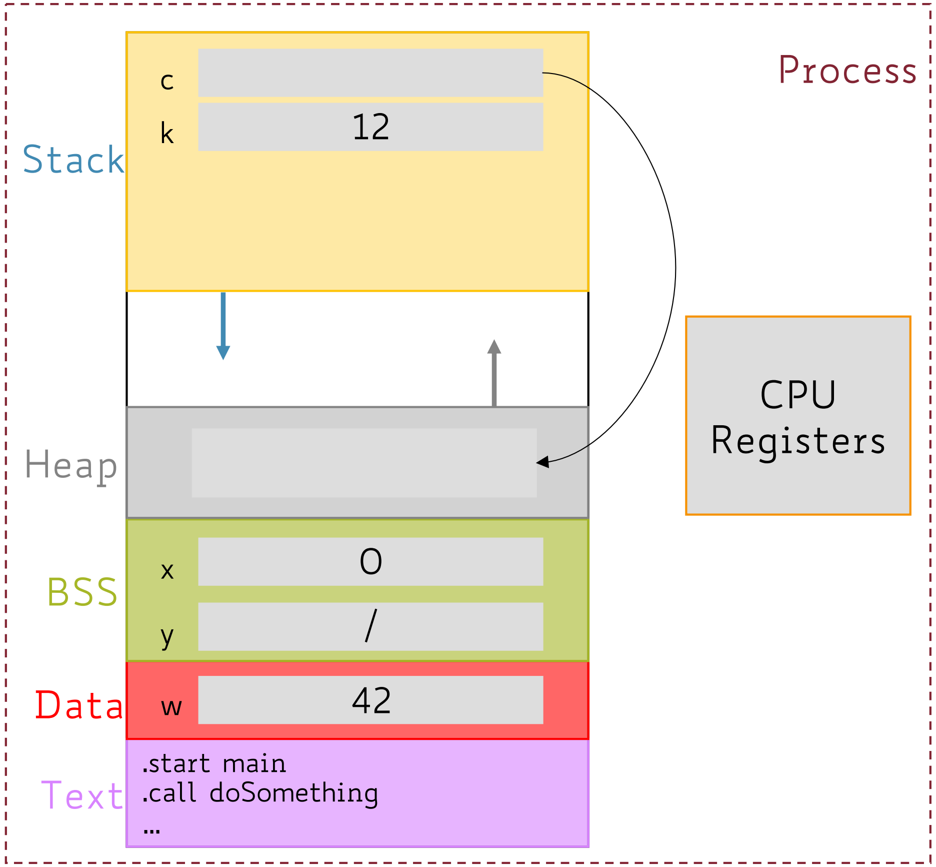
Program vs. Process: Example

Program

```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



```
.start main
.call doSomething
…
```

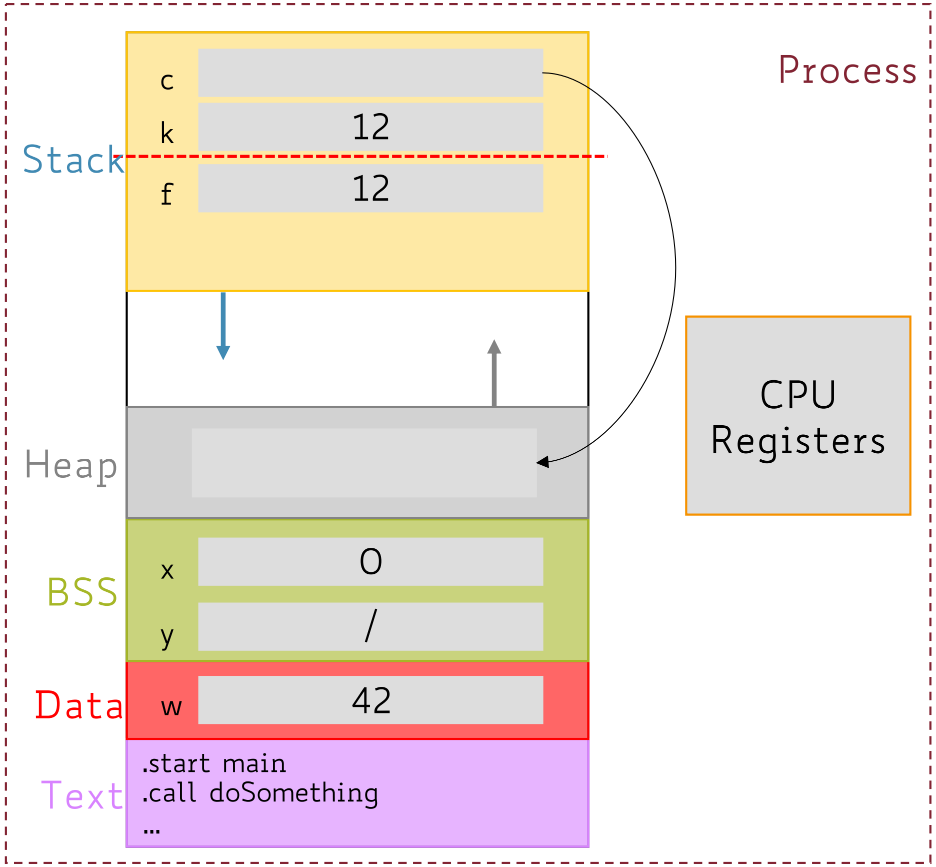
Program vs. Process: Example

Program

```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



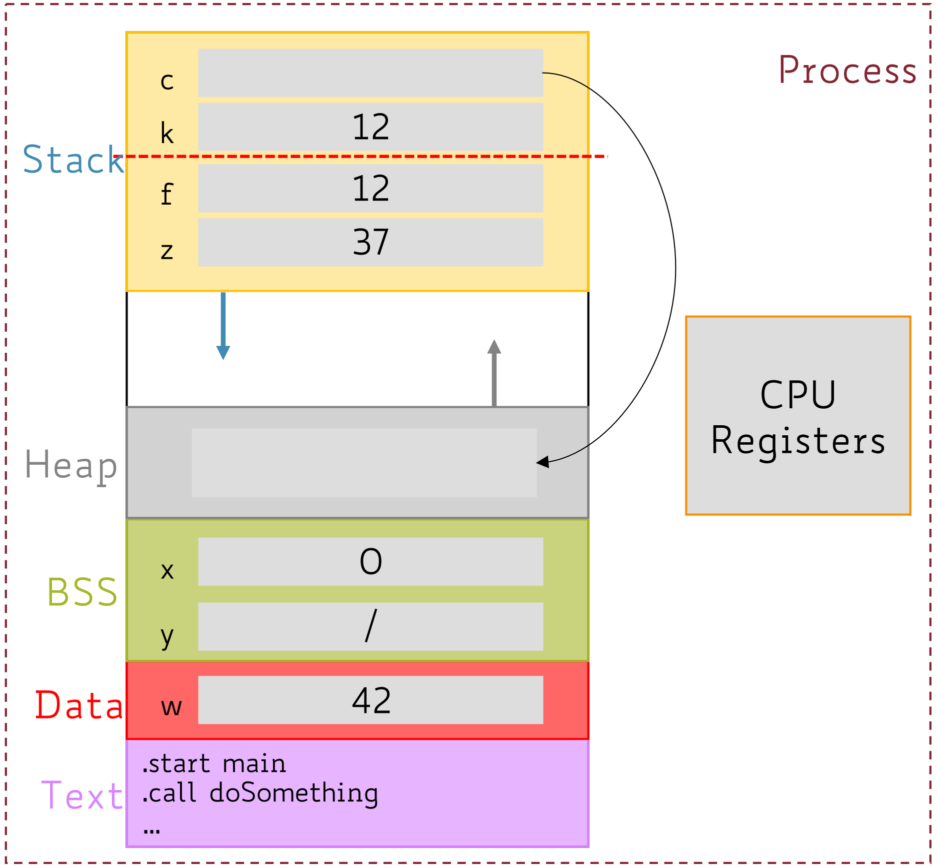
Program vs. Process: Example

Program

```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



Stack

- 2 **operations** are defined on a stack:
  - push → used to place items onto the stack
  - pop → user to remove items from the stack

Stack

- **2 operations** are defined on a stack:
  - push → used to place items onto the stack
  - pop → user to remove items from the stack
- A **dedicated register** (e.g., esp) whose content is the address in main memory of the top of the stack (%esp stands for its content)

Stack

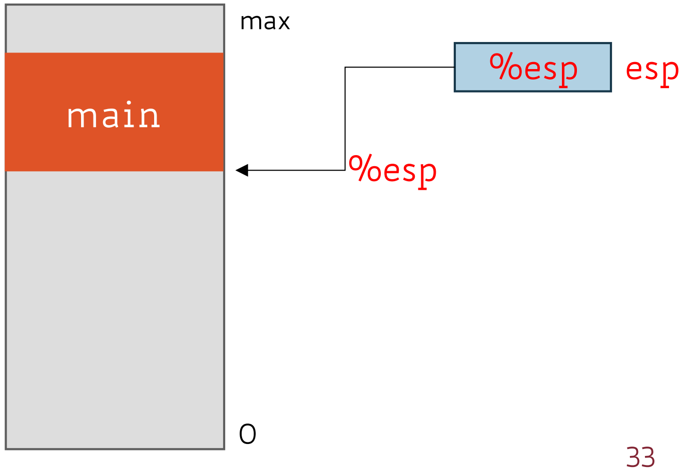
- **2 operations** are defined on a stack:
  - push → used to place items onto the stack
  - pop → user to remove items from the stack
- A **dedicated register** (e.g., esp) whose content is the address in main memory of the top of the stack (%esp stands for its content)
- Stack memory conventionally grows top-down, i.e., from higher to lower memory addresses

Function Call: Stack Frame

- Each function uses a portion of the stack, called **stack frame**
- At every point in time, multiple stack frames may simultaneously exist, due to several nested function calls, yet only one is **active**

```
int main() {
        …
    foo(x, y);

void foo(a,b) {
        …
}
        …
}
```

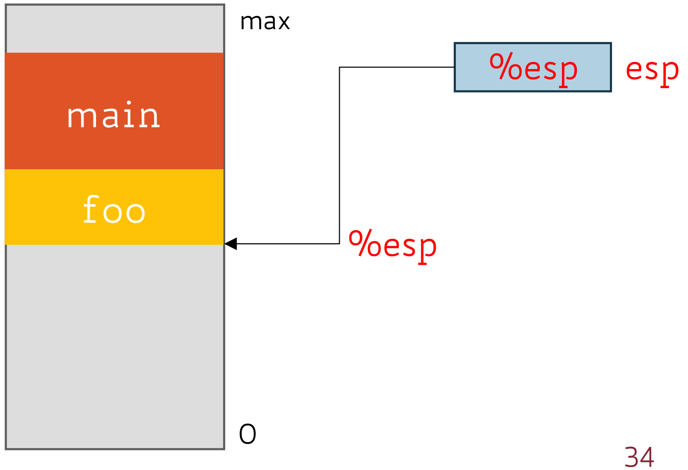


Function Call: Stack Frame

- Each function uses a portion of the stack, called **stack frame**
- At every point in time, multiple stack frames may simultaneously exist, due to several nested function calls, yet only one is **active**

```
int main() {
          …
    foo(x, y);

void foo(a,b) {
          …
}
          …
}
```



Function Call: Stack Frame

- The stack frame for each function is divided into **3 parts**:
  - function parameters + return address
  - back-pointer to the previous stack frame
  - local variables

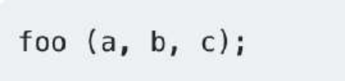
Function Call: Stack Frame

- The stack frame for each function is divided into 3 **parts**:
  - function parameters + return address
  - back-pointer to the previous stack frame
  - local variables
- The first one is set by the caller

Function Call: Stack Frame

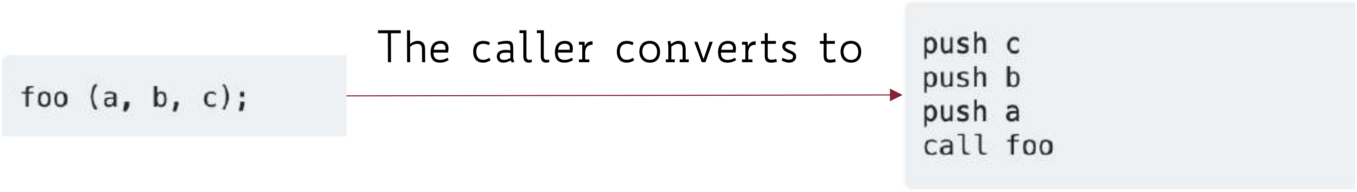
- The stack frame for each function is divided into 3 **parts**:
  - function parameters + return address
  - back-pointer to the previous stack frame
  - local variables
- The first one is set by the caller
- The second and the third ones are set by the callee

Stack Frame: Parameters + Return



```
foo (a, b, c);
```

Stack Frame: Parameters + Return

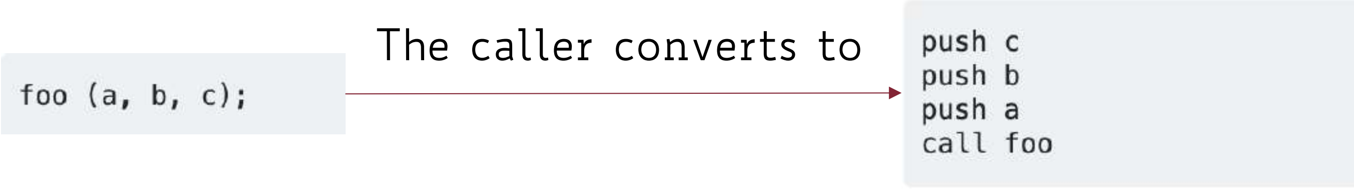


```
foo (a, b, c);
```

The caller converts to

```
push c
push b
push a
call foo
```

Stack Frame: Parameters + Return



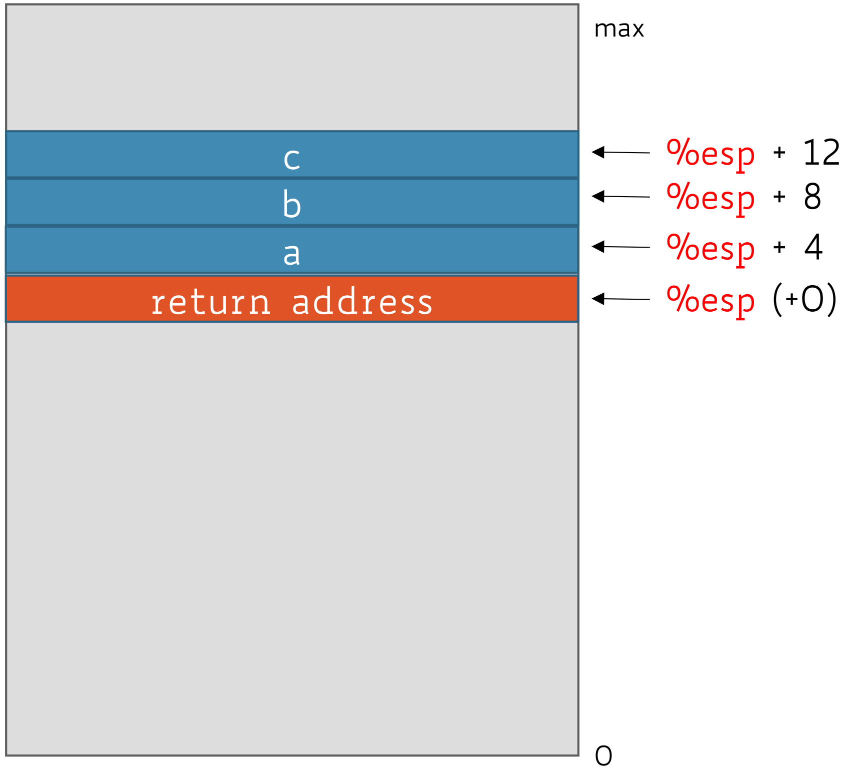
```
foo (a, b, c);
```

The caller converts to

```
push c
push b
push a
call foo
```

- Each item is pushed onto the stack, the stack grows down
- The value of esp register is decremented by, say, 4 bytes (i.e., in 32-bit machines), and the item is copied to the memory location pointed to by it
- The call instruction will implicitly push the return address on the stack

Stack Frame: Parameters + Return

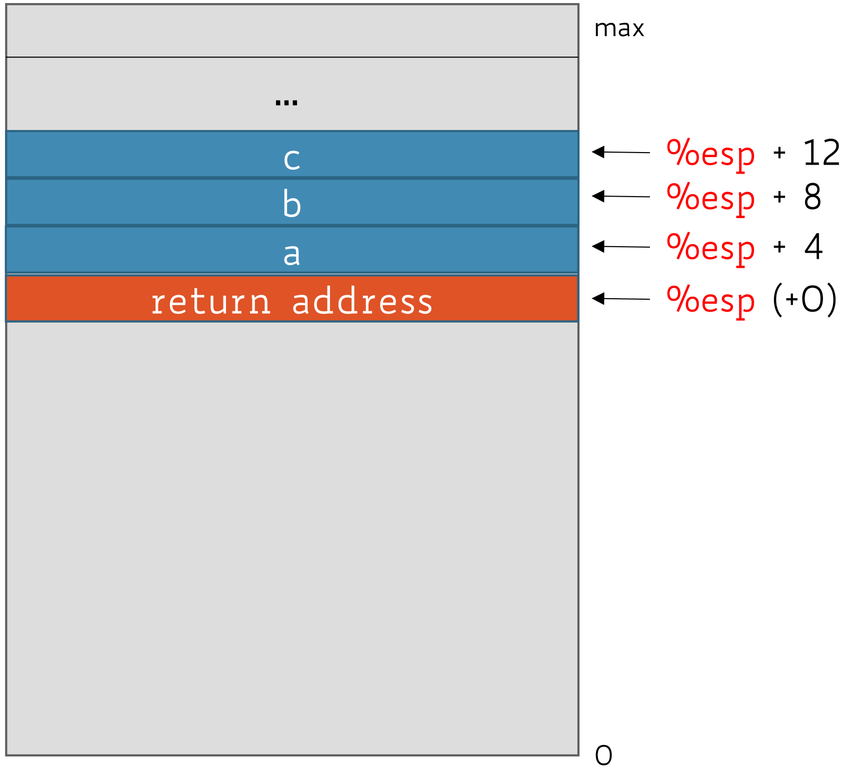


Stack Frame: Parameters + Return

Problem!

The esp pointer gets always updated as the stack grows

It is hard for the callee to access the actual parameters without a fixed reference on the stack



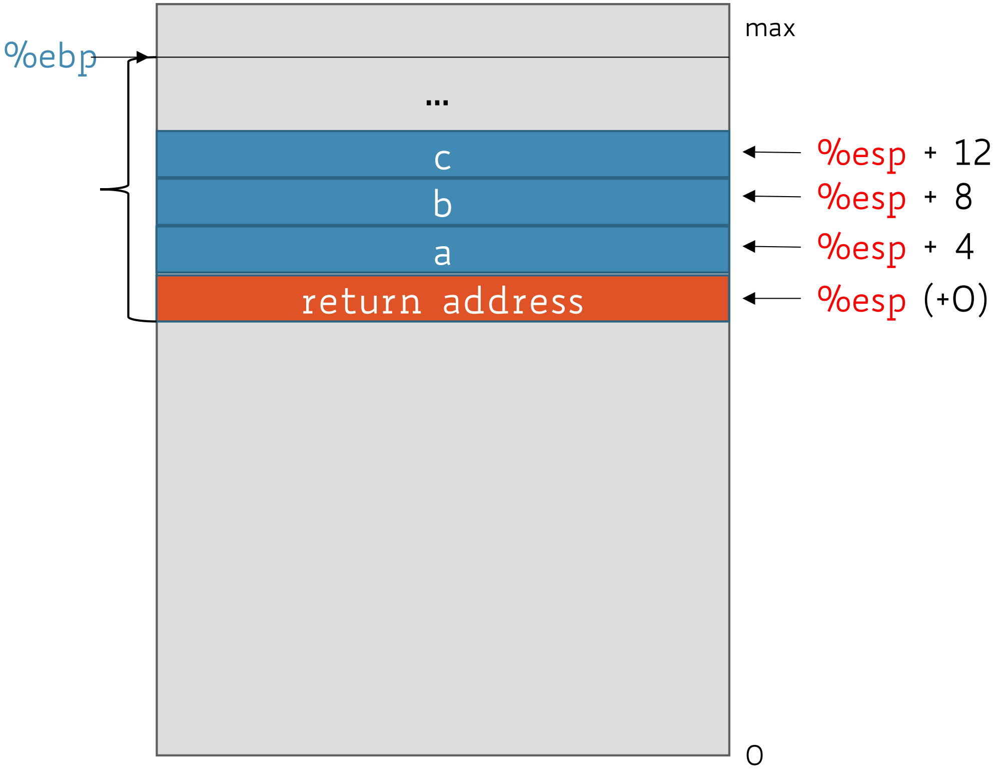
Stack Frame: Parameters + Return

Solution

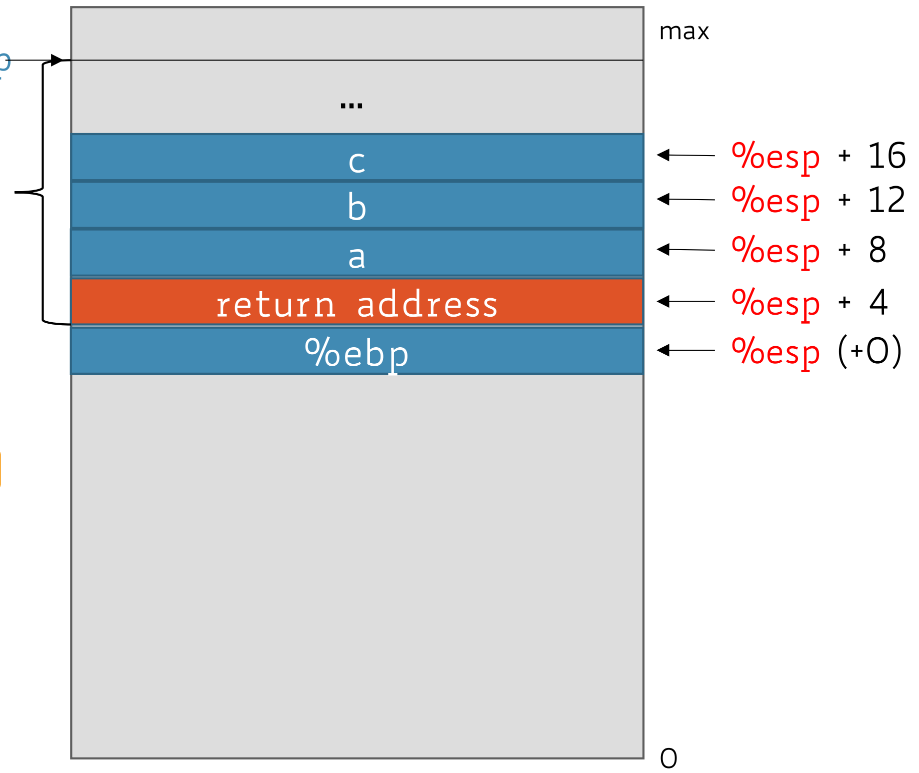
Instead of using a single pointer to the top of the stack (esp)

Use an additional pointer to the bottom (base) of the stack (ebp)

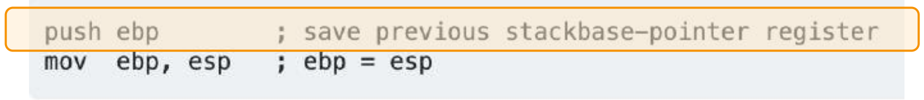
Let esp be free to change across different function calls, while keep ebp fixed within each stack frame



Stack Frame: Saving the Base Frame Pointer

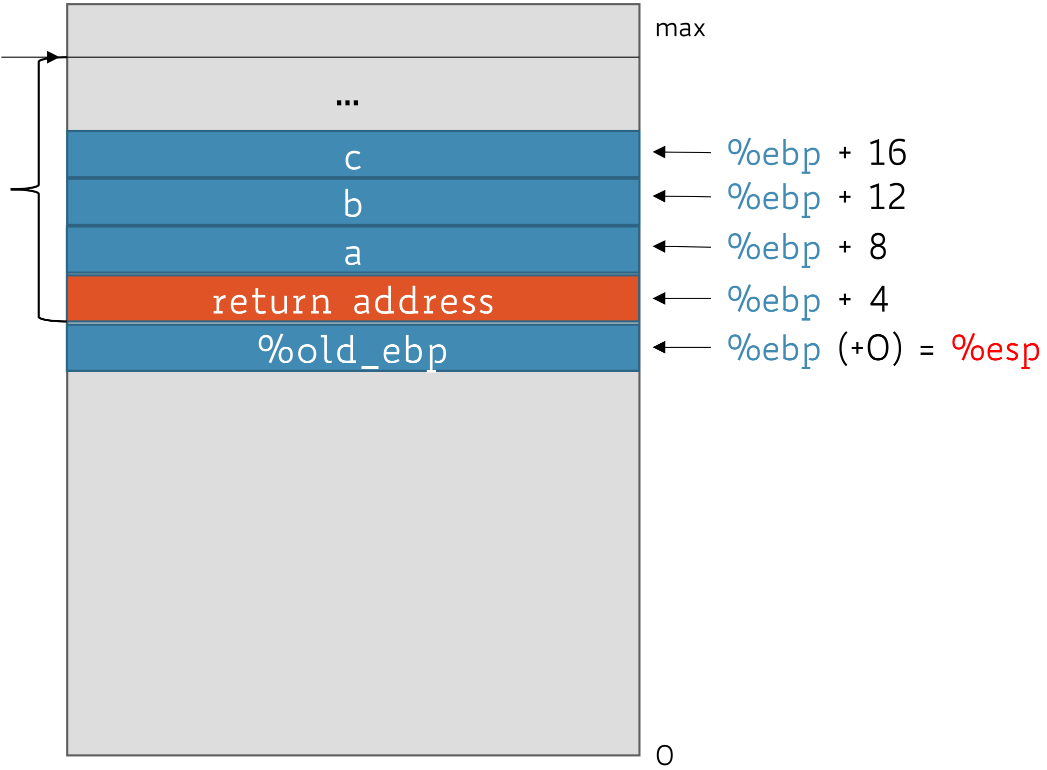


The callee first saves the current stack frame base pointer (%ebp) to the stack!

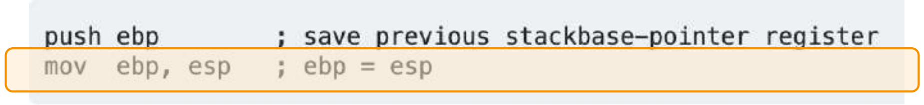


```
push ebp          ; save previous stackbase-pointer register
mov  ebp, esp     ; ebp = esp
```

Stack Frame: Saving the Base Frame Pointer

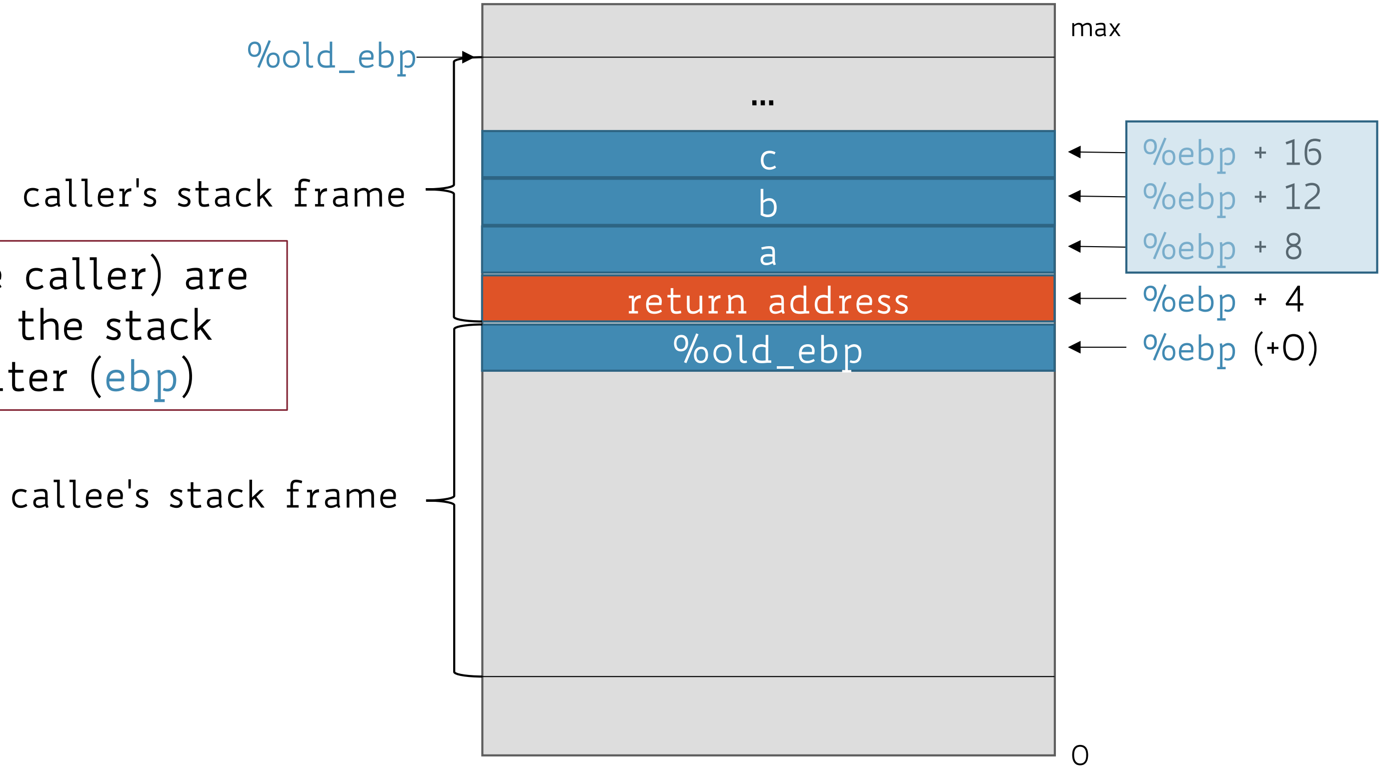


Then it sets the new base frame pointer to the current value of esp



```
push ebp        ; save previous stackbase-pointer register
mov  ebp, esp   ; ebp = esp
```

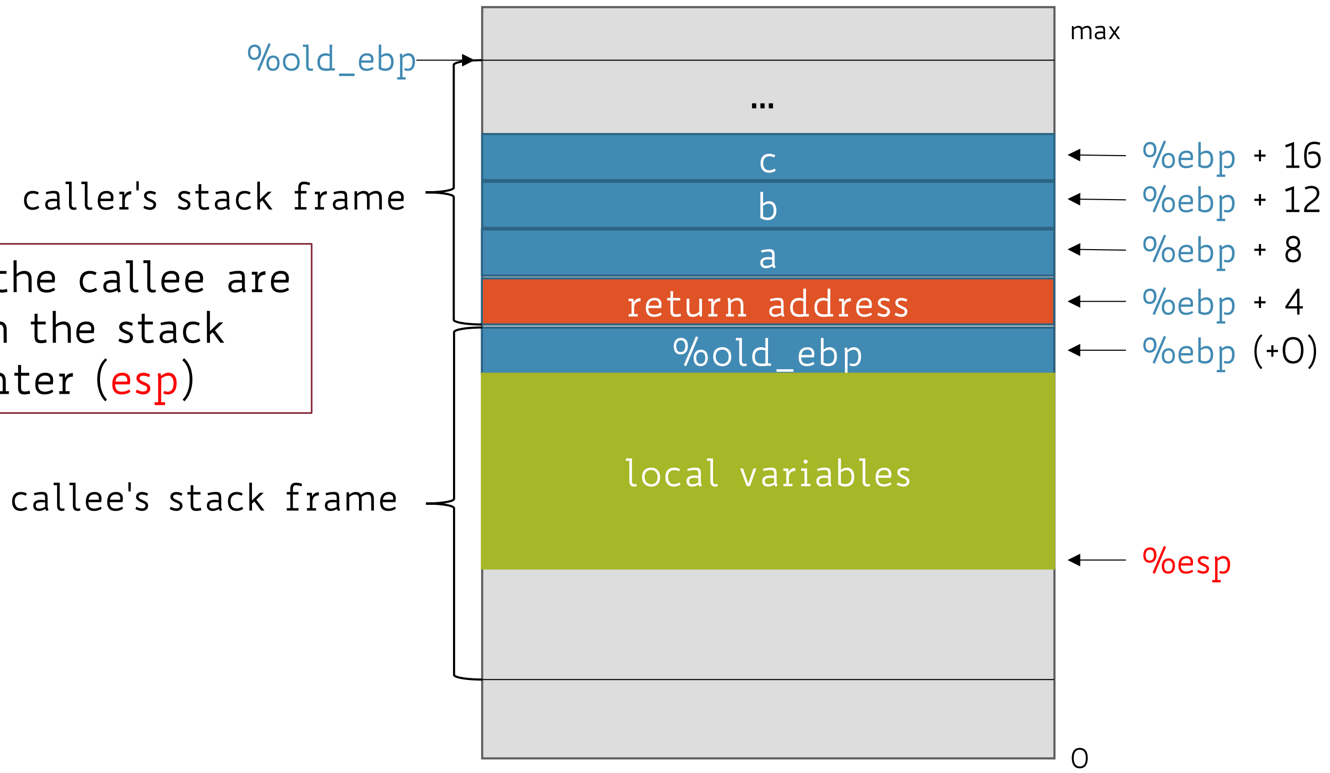
Parameters: Offset from the Base Frame Pointer



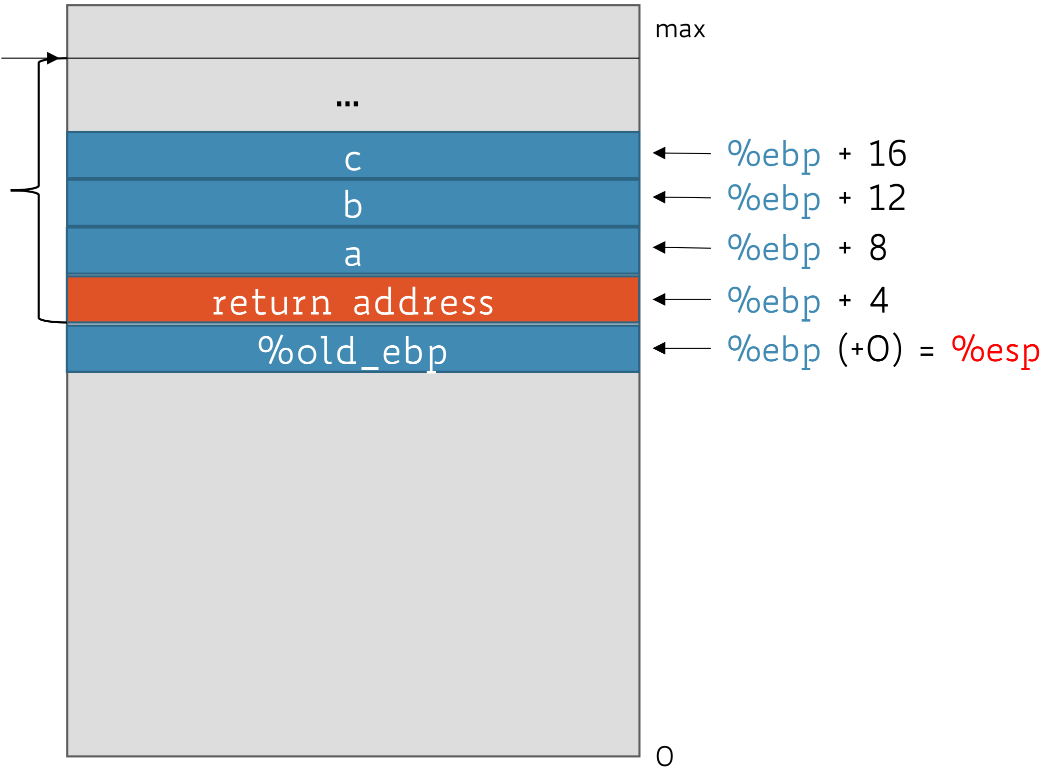
Parameters (of the caller) are accessed through the stack frame base pointer (ebp)

Local Variables: Offset from Stack Pointer

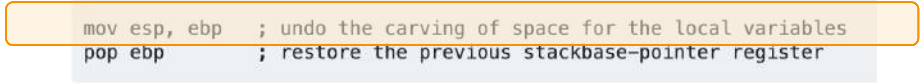
Local variables of the callee are accessed through the stack frame top pointer (esp)



Stack Frame: Cleanup and Return

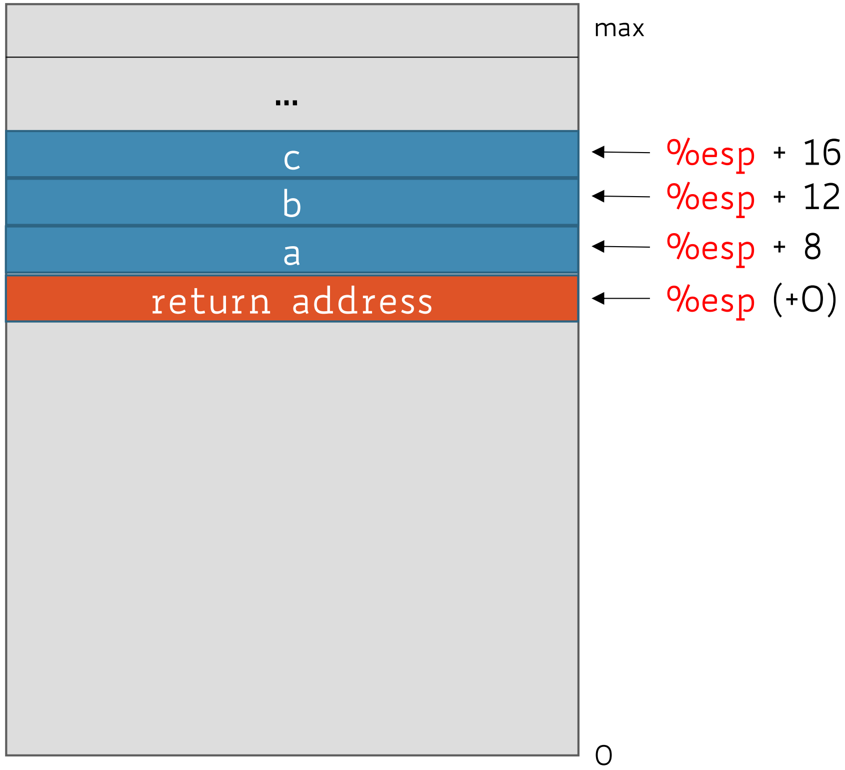


The old stack frame base pointer is restored

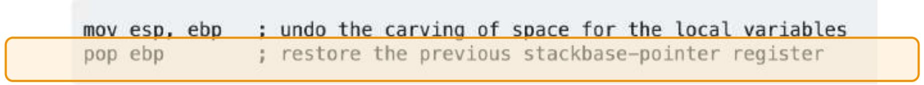


```
mov esp, ebp    ; undo the carving of space for the local variables
pop ebp         ; restore the previous stackbase-pointer register
```

Stack Frame: Cleanup and Return

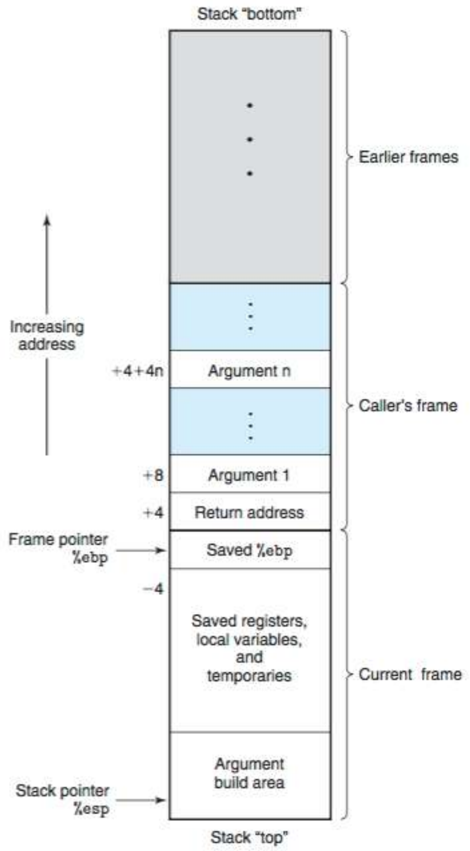


The old stack frame base pointer is restored



```
mov esp, ebp    ; undo the carving of space for the local variables
pop ebp         ; restore the previous stackbase-pointer register
```

Stack: Outline



Process Execution State

- At each time a process can be in one of the following **5 states**:

Process Execution State

- At each time a process can be in one of the following **5 states**:
  - New → The OS has set up the process state

Process Execution State

- At each time a process can be in one of the following **5 states**:
  - New → The OS has set up the process state
  - Ready → The process is ready to be executed yet waiting to be scheduled on to the CPU

Process Execution State

- At each time a process can be in one of the following **5 states**:
  - New → The OS has set up the process state
  - Ready → The process is ready to be executed yet waiting to be scheduled on to the CPU
  - **Running** → The process is actually executing instructions on the CPU

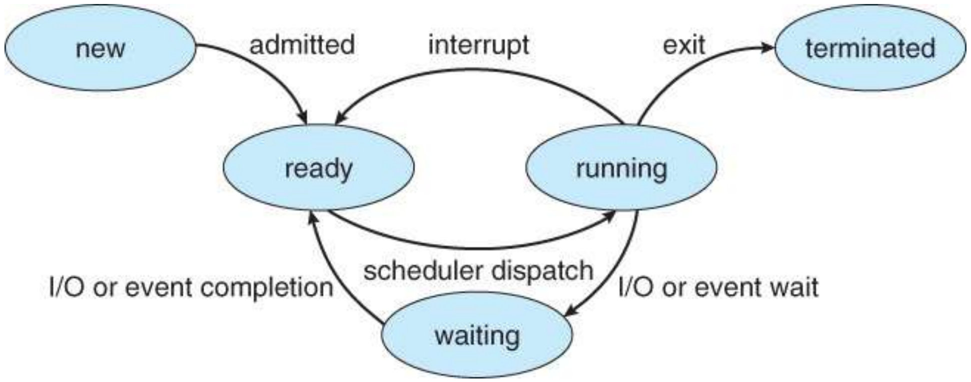
Process Execution State

- At each time a process can be in one of the following **5 states**:
  - New → The OS has set up the process state
  - Ready → The process is ready to be executed yet waiting to be scheduled on to the CPU
  - Running → The process is actually executing instructions on the CPU
  - Waiting → The process is suspended waiting for a resource to be available or an event to complete/occur (e.g., keyboard input, disk access, timer, etc.)

Process Execution State

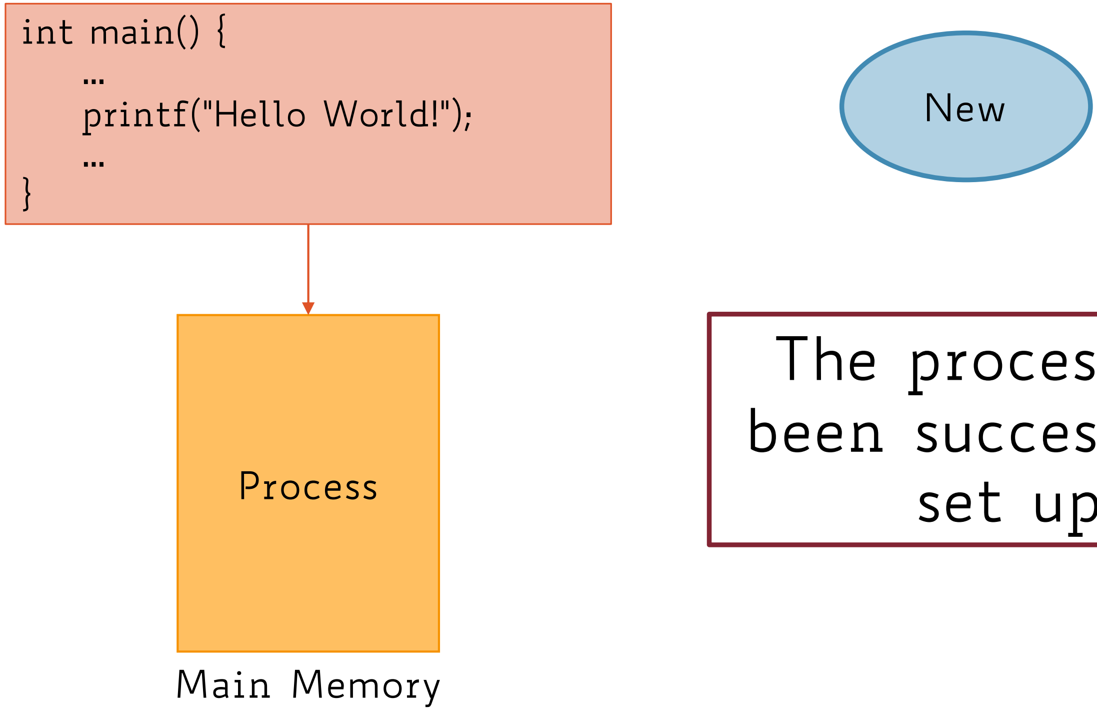
- At each time a process can be in one of the following **5 states**:
  - New → The OS has set up the process state
  - Ready → The process is ready to be executed yet waiting to be scheduled on to the CPU
  - Running → The process is actually executing instructions on the CPU
  - Waiting → The process is suspended waiting for a resource to be available or an event to complete/occur (e.g., keyboard input, disk access, timer, etc.)
  - **Terminated** → The process is finished and the OS can destroy it

Process Execution State Diagram



- As the process executes, it moves from state to state depending on:
    - program actions (e.g., system calls)
    - OS actions (e.g., scheduling)
    - external actions (e.g., interrupts)

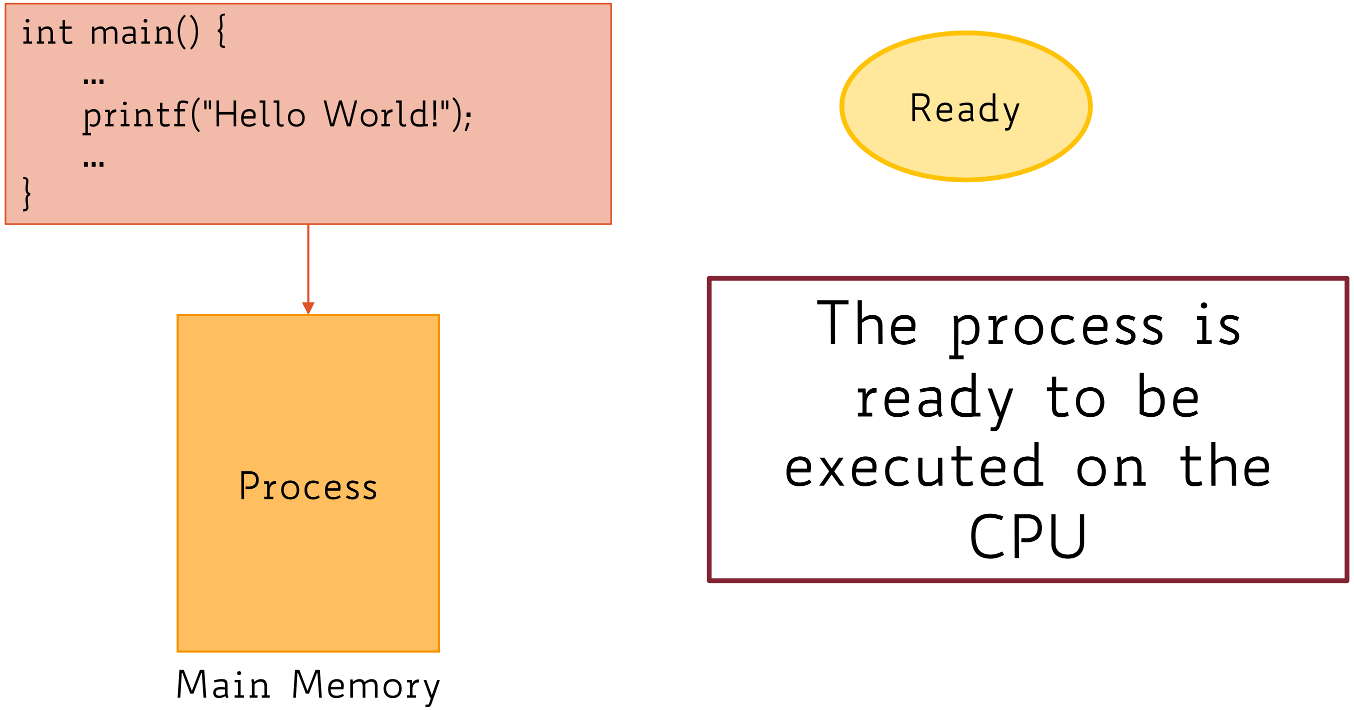
Process Execution State: Example



```
int main() {
    …
    printf("Hello World!");
    …
}
```

The process has been successfully set up

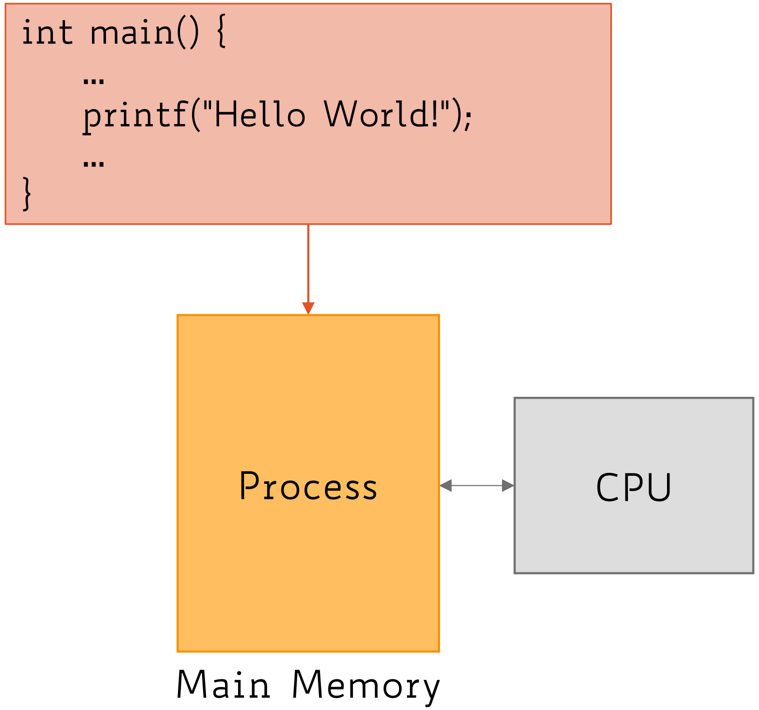
Process Execution State: Example



```
int main() {
    …
    printf("Hello World!");
    …
}
```

The process is ready to be executed on the CPU

Process Execution State: Example

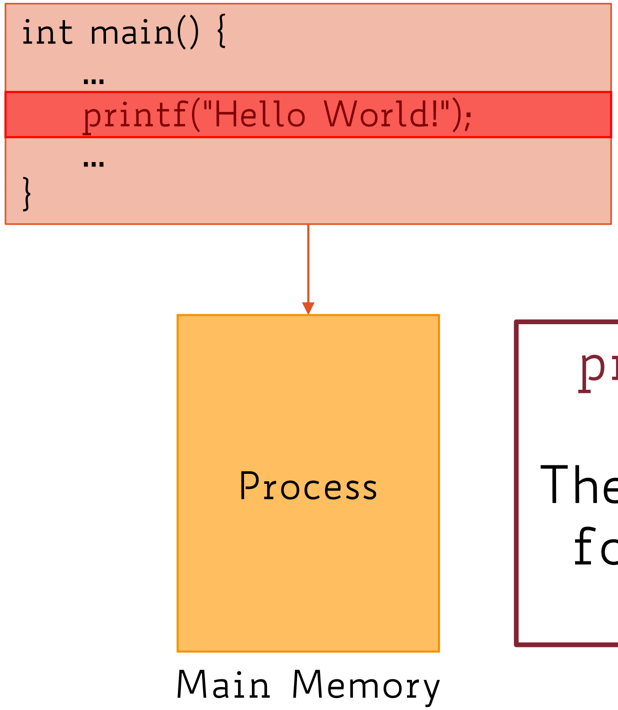


```
int main() {
    …
    printf("Hello World!");
    …
}
```

Running

The process is actually executing on the CPU

Process Execution State: Example

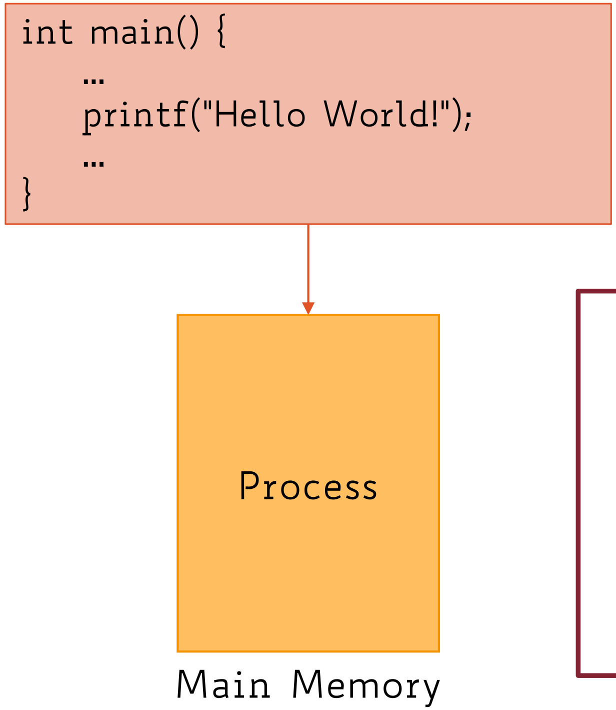


```
int main() {
    …
    printf("Hello World!");
    …
}
```

Waiting

printf delegates off to a blocking I/O system call:
The current process is suspended in order for the OS to schedule another process which is ready to run

Process Execution State: Example

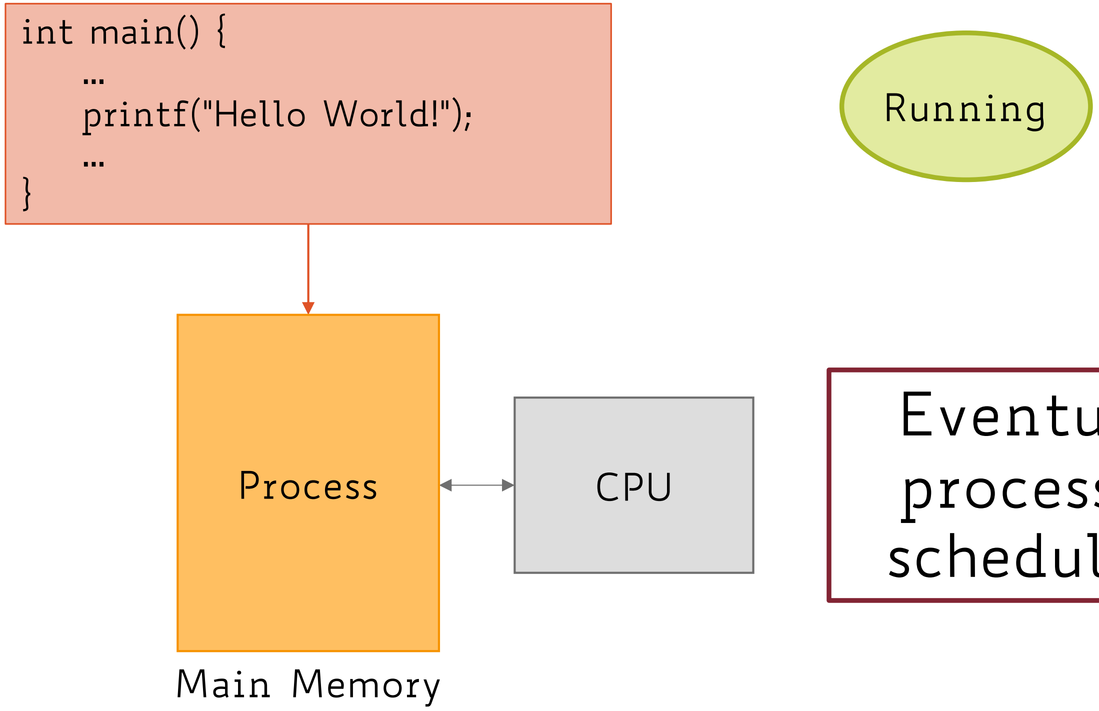


```
int main() {
    ...
    printf("Hello World!");
    ...
}
```

Ready

Once printf is done (e.g., HW interrupt) the process goes back to ready status yet it is not resumed directly!
(i.e., it is up to the OS to decide which process to schedule next)

Process Execution State: Example

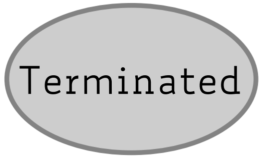


```
int main() {
    …
    printf("Hello World!");
    …
}
```

Eventually, the process will be scheduled again

Process Execution State: Example

```
int main() {
    …
    printf("Hello World!");
    …
}
```



Terminated

Finally, the process terminates

Blocking vs. Non-Blocking Calls (Again!)

- Most system calls (e.g., I/O ones) are blocking

Blocking vs. Non-Blocking Calls (Again!)

- Most system calls (e.g., I/O ones) are blocking
  - the caller process (user space) can't do anything until the system call returns

Blocking vs. Non-Blocking Calls (Again!)

- Most system calls (e.g., I/O ones) are blocking
    - the caller process (user space) can't do anything until the system call returns
    - the OS (kernel space):
        - sets the current process to a waiting state (i.e., waiting for the system call to return)
        - schedules a different ready process to avoid the CPU being idle

Blocking vs. Non-Blocking Calls (Again!)

- Most system calls (e.g., I/O ones) are blocking
    - the caller process (user space) can't do anything until the system call returns
    - the OS (kernel space):
        - sets the current process to a waiting state (i.e., waiting for the system call to return)
        - schedules a different ready process to avoid the CPU being idle
    - once the system call returns the previously blocked process is ready to be scheduled for execution again

Blocking vs. Non-Blocking Calls (Again!)

- Most system calls (e.g., I/O ones) are blocking
  - the caller process (user space) can't do anything until the system call returns
  - the OS (kernel space):
    - sets the current process to a waiting state (i.e., waiting for the system call to return)
    - schedules a different ready process to avoid the CPU being idle
  - once the system call returns the previously blocked process is ready to be scheduled for execution again

NOTE:
the whole system is **not** blocked, only the process which has requested the blocked call is!

Process State

- At least, process state consists of the following:
  - the code of the running program
  - the static data of the running program
  - the program counter (PC) indicating the next instruction to execute
  - CPU registers
  - the program's call chain (stack) along with frame and stack pointers
  - the space for dynamic memory allocation (heap) along with the heap pointer
  - the set of resources in use (e.g., open files)
  - the process execution state (ready, running, etc.)

Process Control Block (PCB)

- The main data structure used by the OS to keep track of any process
- The PCB keeps track of the execution state and location of a process
- The OS allocates a new PCB upon the creation of a process and places it into a state queue
- The OS deallocates a PCB as soon as the associated process terminates

Process Control Block (PCB)

- At least, the PCB contains the following:

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)
  - Program Counter (PC) + Stack Pointer (SP) + general purpose registers

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)
  - Program Counter (PC) + Stack Pointer (SP) + general purpose registers
  - CPU scheduling information → priority and pointers to state queues

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)
  - Program Counter (PC) + Stack Pointer (SP) + general purpose registers
  - CPU scheduling information → priority and pointers to state queues
  - Memory management information → page tables

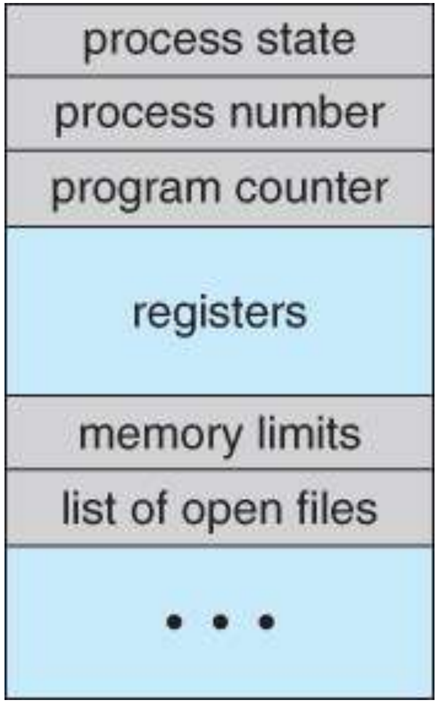
Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)
  - Program Counter (PC) + Stack Pointer (SP) + general purpose registers
  - CPU scheduling information → priority and pointers to state queues
  - Memory management information → page tables
  - Accounting information → user and kernel CPU time consumed, owner

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)
  - Program Counter (PC) + Stack Pointer (SP) + general purpose registers
  - CPU scheduling information → priority and pointers to state queues
  - Memory management information → page tables
  - Accounting information → user and kernel CPU time consumed, owner
  - I/O status → list of open files

Process Control Block (PCB)



process state

process number

program counter

registers

memory limits

list of open files

• • •

Summary

- Process is the **`unit of execution`** (running on a single CPU)

Summary

- Process is the **unit of execution** (running on a single CPU)
- OS gives every process the illusion of having a contiguous sequence of memory addresses that they can refer (**virtual address space**)

Summary

- Process is the **unit of execution** (running on a single CPU)
- OS gives every process the illusion of having a contiguous sequence of memory addresses that they can refer (**virtual address space**)
- OS keeps track of process-related information using an ad hoc data structure called **Process Control Block** (**PCB**)

Summary

- Process is the **unit of execution** (running on a single CPU)
- OS gives every process the illusion of having a contiguous sequence of memory addresses that they can refer (**virtual address space**)
- OS keeps track of process-related information using an ad hoc data structure called Process Control Block (PCB)
- Process can be in one of **5 possible states**: **new**, **ready**, **waiting**, **running**, or **terminated**