Process Anatomy

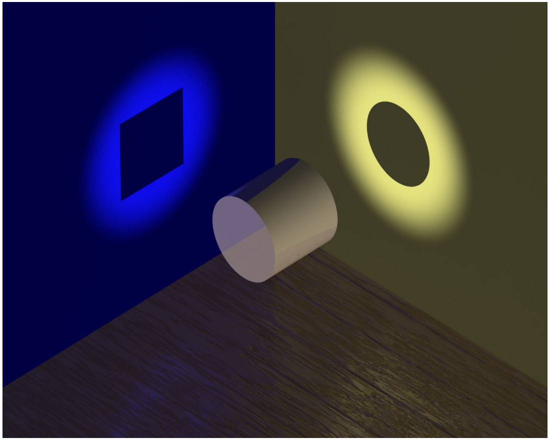
CSE 220: Systems Programming

Ethan Blanton & Carl Alphonce

Department of Computer Science and Engineering

University at Buffalo

Planning and Development I



Jean-Cristophe Benoist CC-BY-SA-3.0 (original)

Planning and Development II

When writing code, arrange different views of the problem:

- Documentation (your handout!)
- Diagrams
- Pseudocode
- Code

Each view will give you different insight.

Compare them against each other!

Processes

What is a process?
From the text:

[A] process is an instance of a program in execution.

If a program is a set of machine instructions, a process is:

- Those instructions
- The memory they use
- The system resources they access
- …

Programs

Source code is compiled into an executable.

The program that a process runs is loaded from an executable. Once loaded, the system provides an execution environment.

Unix Processes

A Unix process is protected from other processes:

- It has its own memory.
- It appears to execute on a dedicated CPU.
- The system services it uses are dedicated to it.

Hardware assistance is required to maintain this environment.

In particular, virtual memory provides the illusion of a private, contiguous memory space.

Basic Layout

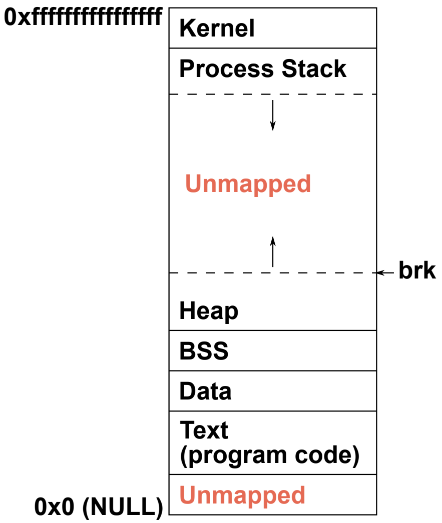
A process's memory is divided into sections.

These sections represent different types of information.

Some sections come from the executable.

Some sections are created at run time.

The lowest addresses are not used — specifically so that `NULL` remains invalid!



Static Data

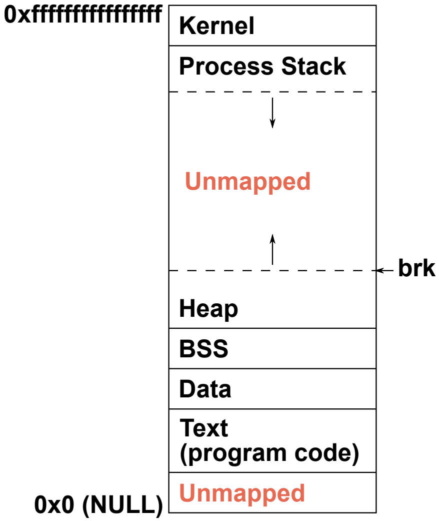
The lowest sections are known at compile time.

The text section is the executable code.

The data and BSS (“block started by symbol”) sections are global and static local variables.

Variables in the data section have initialized values in the source code.

Variables in the BSS do not.



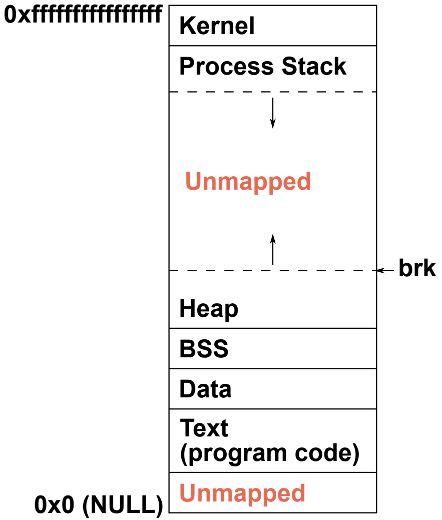
Dynamic Data

The middle sections are allocated dynamically as the program runs.

The heap is managed by the dynamic allocator (`malloc()`).

The stack contains local variables and information necessary for function calls.

These sections begin with zero size and grow as needed during execution.



Using Sections

Section locations can be valuable debugging information.

The precise locations of sections will vary.

You can assume their relative positions, however!

Recognizing whether a pointer is on the stack, on the heap, or in the data section can be very valuable.

Recognizing that very small pointers are invalid is even more valuable!

Example Layout

```
char *string = "hello";
int iSize;

char *f(void) {
    char *p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

Consider this function.

Example Layout

```
char *string = "hello";
int iSize;

char *f(void) {
    char *p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

These components are stored in the text section, created by the compiler at compile time.

Example Layout

```
char *string = "hello";
int iSize;

char *f(void) {
    char *p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

This variable is stored in the data section, initialized by the compiler at compile time.

Example Layout

```
char *string = "hello";
int iSize;

char *f(void) {
    char *p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

This variable is stored in the BSS, provisioned at compile time and set to zero at run time.

Example Layout

```
char *string = "hello";
int iSize;

char *f(void) {
    char *p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

This variable is stored on the stack, created by the compiled code at run time.

Example Layout

```
char *string = "hello";
int iSize;

char *f(void) {
    char *p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

This memory is stored on the heap, allocated at run time and its address assigned to a variable on the stack.

Static Allocations

The static allocations, the data and BSS sections, are created when the program starts.

The data section is copied from the executable into memory.

The BSS is set to its final size and cleared to binary zeroes.

The sizes of both of these sections are known at compile time.

This memory is released only when the program exits!

Dynamic Allocations

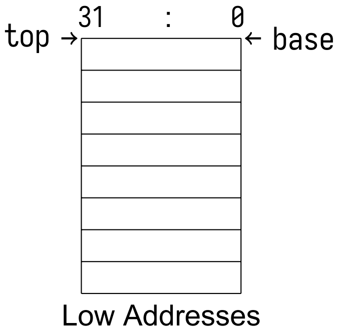
The dynamic allocations, the heap and stack sections, are resized as the program runs.

Stack frames are created and destroyed as functions are called.

Heap memory is allocated with `malloc()` *et al.* and freed with `free()`.

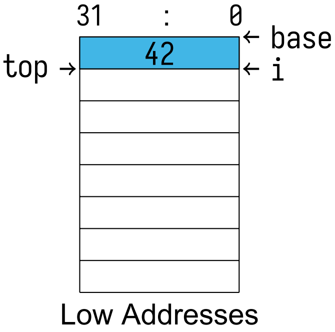
Un-freed memory is released by the OS when the program ends.

Stack Operations



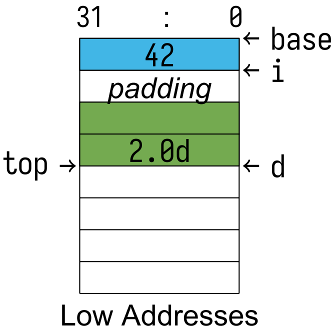
(An empty stack; each row is 32 bits.)

Stack Operations



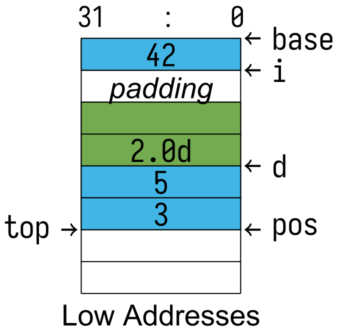
```
push int i = 42;
```

Stack Operations



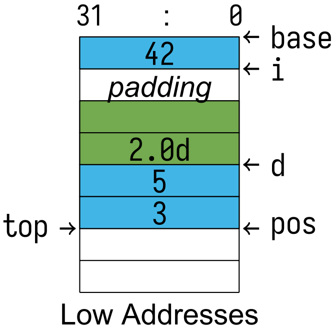
push `double d = 2.0;`
(Remember padding!)

Stack Operations



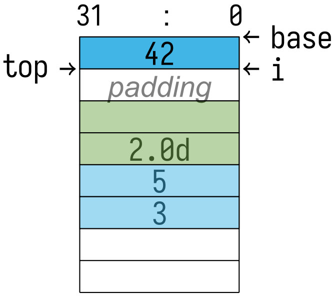
push `struct { int x; int y; } pos = { x = 3, y = 5 };`

Stack Operations



Stack items are typically referenced with respect to its top.
E.g., `d` is at `top + 8`

Stack Operations



Low Addresses

pop 20 bytes to remove `pos` and `d`
Note that the unused data remains present on the stack.

Variable Declarations

A variable declaration does two things:

- Asks the compiler to reserve space on the stack for data
- Names the location of that data

```
int array[32];
```

“Make space for 32 integers and call that space `array`.”

Variable Declarations

A variable declaration does two things:

- Asks the compiler to reserve space on the stack for data
- Names the location of that data

```
int array[32];
```

"Make space for 32 integers and call that space `array`."

Variable Declarations

A variable declaration does two things:

- Asks the compiler to reserve space on the stack for data
- Names the location of that data

```
int array[32];
```

"Make space for 32 integers and call that space `array`."

Variable Declarations

A variable declaration does two things:

- Asks the compiler to reserve space on the stack for data
- Names the location of that data

```
int array[32];
```

"Make space for 32 integers and call that space `array`."

Every non-`static` local variable is an automatic variable.

Automatic Variable Lifetime

Automatic variables are:

- Guaranteed to be allocated before they are first referenced
- Guaranteed to be valid until their enclosing block is done

In many cases they are created when the function is entered.

Placing automatic variables on the stack allows this.

Automatic Variable Placement

Automatic variables may be allocated anywhere.

The programmer cannot predict their order or location.

They may only be in registers!

Their structure will be preserved.

Automatic Variable Placement

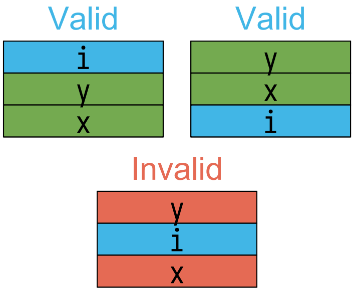
Automatic variables may be allocated anywhere.

The programmer cannot predict their order or location.

They may only be in registers!

Their structure will be preserved.

```
int i;
struct {
   int x; int y;
} pos;
```



Function Call Nesting

Note that:

- Function calls form a tree over the life of a program
- Function calls form a stack at any point in time

This is because:

- A function may call many functions consecutively
- A function can call only one function at a time

These properties directly affect the program stack.

Function Calls

At its simplest, a function call consists of:

- A jump to a new program location
- Execution of the function code
- A jump back to the calling location

However, many function calls are more complicated. They may:

- Allocate automatic variables
- Call other functions
- Temporarily save registers
- …

In these cases, functions require a stack frame.

Stack Frames

A stack frame[1] holds information for a single function invocation.

While the details vary by platform, it will include:

- Saved processor registers
- Local variables for the current function
- Arguments for any called function
- The return location for any called function

We will discuss all of these except saved processor registers.
(Maybe we'll get to those later.)

[1]You will sometimes see this called an activation record.

Local Variables

We have previously discussed automatic variables.

Often, all local variables for a function are allocated together.

When the function is entered, it will immediately move the top of the stack to make room for its local storage.

This portion of the stack frame is then of fixed size.

Its size is often not saved, but recorded in the program instructions by the compiler.

The location of individual variables are likewise recorded.

Function Arguments

The platform ABI will determine how arguments are passed.

Normally, it is a combination of registers and stack space.

On x86-64 Linux, the first six 64-bit values will be in registers.
Any additional arguments are pushed onto the stack.

Therefore, many functions have no arguments on the stack.

Function Arguments Layout

If function arguments are pushed onto the stack, they are normally pushed in reverse order.

That is, the first function argument is closest to the top.

Among other reasons, this allows for a variable number of arguments.

Consider `printf`: it takes 1 or more arguments.

The first format argument tells it how many.

The Program Counter

The other major item that must be tracked for the function call stack is the program counter.

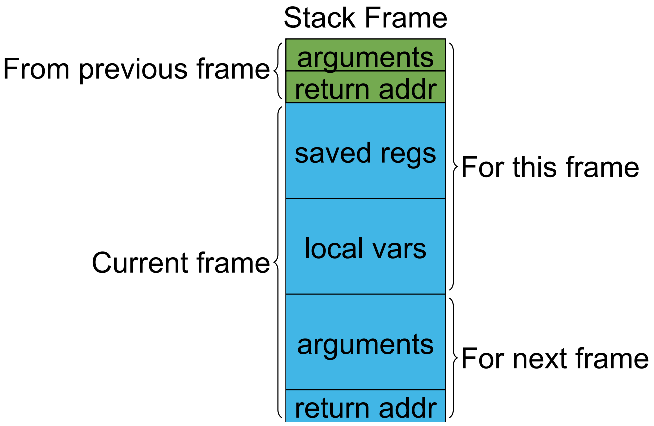
The program counter is the address of the machine instruction the processor is currently executing.

For a function call:

- the current program counter is pushed before jumping to the called function
- the called function pops the program counter in order to return

On some architectures there is a dedicated instruction for this.

A Stack Frame



(Exactly which elements are part of which frame is debatable.)

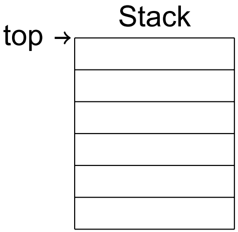
A Call Stack

```
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```



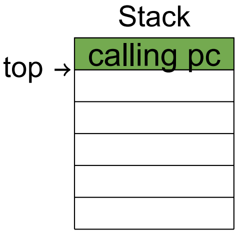
A Call Stack

```
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```



call foo()

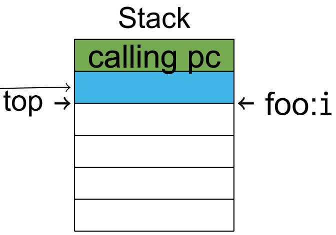
A Call Stack

```
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```



Reserve space for `foo()`'s locals

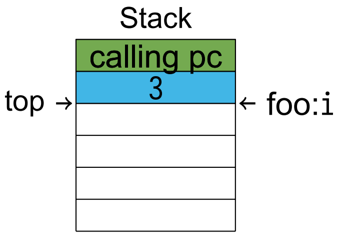
A Call Stack

```
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```



Execute foo()

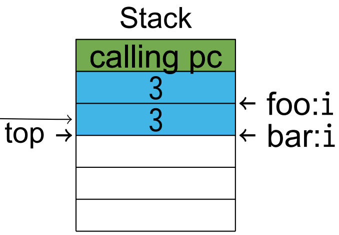
A Call Stack

```
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```



Execute `foo();`
prepare to call `bar()`

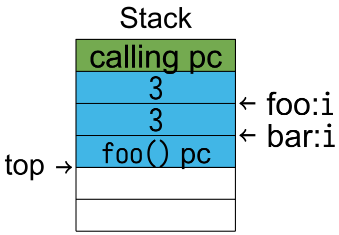
A Call Stack

```
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```



Push PC; call bar()

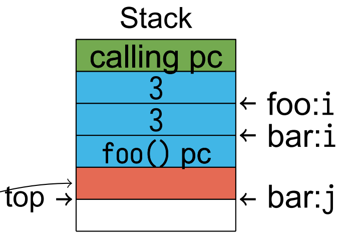
A Call Stack

```
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```



Reserve space for bar()'s locals

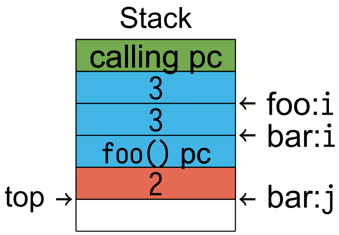
A Call Stack

```
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```



Execute bar()

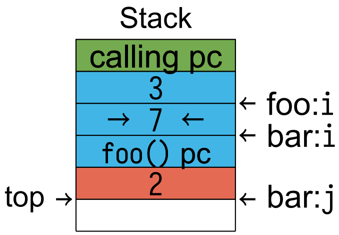
A Call Stack

```
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```



Execute bar()

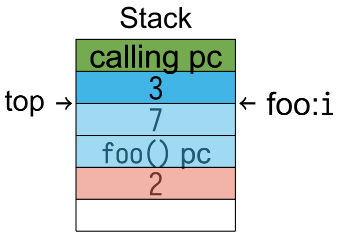
A Call Stack

```
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```



Return from bar();
Pop bar()'s stack frame;
Execute foo()

Summary

- POSIX programs are laid out in sections
- The stack grows downward
- Automatic variables are allocated on the stack
- Stack frames track function calls
- Items removed from the stack are not cleared
- Stack-allocated arguments are how C is call-by-value

References I

Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 3: 3.7 Intro, 3.7.1. Pearson, 2016.

[2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Chapter 4. Prentice Hall, 1988.

License