Process Layout

Karthik Dantu
Ethan Blanton
Computer Science and Engineering
University at Buffalo
kdantu@buffalo.edu

Karthik Dantu
• **C source code**
  C statements organized into functions
  Stored as a collection of files (.c and .h)

• **Executable module**
  Binary image generated by compiler
  Stored as a file (e.g., a.out)

• **Process**
  Instance of a program that is executing
  With its own address space in memory
  With its own id and execution state
  Managed by the operating system

---

Karthik Dantu
What is virtual memory?
Contiguous addressable memory space for a single process
May be swapped into physical memory from disk in pages
Let’s you pretend each process has its own contiguous memory
What to Store: Code and Constants

- Executable code and constant data
  Program binary, and any shared libraries it loads
  Necessary for OS to read the commands
- OS knows everything in advance
  Knows amount of space needed
  Knows the contents of the memory
- Known as the "text" segment
- Note: Some systems (e.g., hats) store some constants in "rodata" section

Karthik Dantu
What to Store: “Static” Data

- Variables that exist for the entire program
  Global variables, and “static” local variables
  Amount of space required is known in advance

- **Data**: initialized in the code
  Initial value specified by the programmer
  E.g., “int \( x = 97 \),”
  Memory is initialized with this value

- **BSS**: not initialized in the code
  Initial value not specified
  E.g., “int \( x \),”
  All memory initialized to 0 (on most OS’s)
  BSS stands for “Block Started by Symbol”

Karthik Dantu
What to Store: Dynamic Memory

- Memory allocated while program is running
  E.g., allocated using the `malloc()` function
  And deallocated using the `free()` function

- OS knows nothing in advance
  Doesn’t know the amount of space
  Doesn’t know the contents

- So, need to allow room to grow
  Known as the “heap”
  Detailed example in a few slides
  More in programming assignment #4

Karthik Dantu
What to Store: Temporary Variables

- Temporary memory during lifetime of a function or block
  Storage for function parameters and local variables

- Need to support nested function calls
  One function calls another, and so on
  Store the variables of calling function
  Know where to return when done

- So, must allow room to grow
  Known as the “stack”
  Push on the stack as new function is called
  Pop off the stack as the function ends

- Detailed example later on
Memory Layout: Summary

- **Text**: code, constant data
- **Data**: initialized global & static variables
- **BSS**: uninitialized global & static variables
- **Heap**: dynamic memory
- **Stack**: local variables
char* string = "hello";
int iSize;

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

Karthik Dantu
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
char* string = "hello";
int isize;

char* f(void)
{
    char* p;
    isize = 8;
    p = malloc(isize);
    return p;
}
char* string = "hello";
int iSize;

cchar* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
iSize = 8;
p = malloc(iSize);
return p;
}
Memory Allocation and De-allocation

- **How, and when, is memory allocated?**
  - Global and static variables: program startup
  - Local variables: function call
  - Dynamic memory: `malloc()`

- **How is memory deallocated?**
  - Global and static variables: program finish
  - Local variables: function return
  - Dynamic memory: `free()`

- **All memory deallocated when program ends**
  - It is good style to free allocated memory anyway

Karthik Dantu
Memory Allocation Example

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

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

Data: "hello" at startup
BSS: 0 at startup

Stack: at function call
Heap: 8 bytes at malloc
Memory Deallocation Example

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

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

- `char* string = "hello";`: Available till termination
- `int iSize;`: Available till termination
- `char* f(void){`: Deallocate on return from `f`
- `char* p;`: Deallocate on `free()`
Aside: Using Sections

• The exact addresses of sections will vary
• However, you can usually assume certain things
• We’ll look at some of those properties later
• Learning to recognize the location of a pointer is valuable
• For example: all pointers < 4096 (0x1000) are invalid!
Stack Operations

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

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

Karthik Dantu
push double d = 2.0;
(Remember padding!)
push `struct { int x; int y; } pos = { x = 3, y = 5 };`

Stack items are typically referenced with respect to its `top`. E.g., `d` is at `top + 8`
pop 20 bytes to remove pos and d
Note that the unused data remains present on the stack.

Karthik Dantu
Variable Declarations

• A variable does two things
  Ask compiler to reserve memory for data
  Name 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

Karthik Dantu
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

```c
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 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.)
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 are passed 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 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

From previous frame:
- arguments
- return addr

Current frame:
- saved regs
- local vars
- arguments
- return addr

For this frame:
- arguments
- return addr

For next frame:
- saved regs
- local vars
- arguments
- return addr
void foo() {
    int i = 3;

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

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

    i = 5 + j;
}
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
Stack Frame: Example

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Execute foo()
```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Stack Frame: Example

Stack

```
<table>
<thead>
<tr>
<th>calling pc</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
```

Execute `foo();` prepare to call `bar()`
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}

Stack Frame: Example
Stack Frame: Example

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Karthik Dantu
Stack Frame: Example

```c
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 is a section
• 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 why C is call-by-value

Karthik Dantu
Heap: Dynamic Memory

- `#include <stdlib.h>`
  - `void *malloc(size_t size);`
  - `void free(void *ptr);`

```c
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

Karthik Dantu
```c
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

Karthik Dantu
Heap: Dynamic Memory

- `#include <stdlib.h>
  void *malloc(size_t size);
  void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);`
Heap: Dynamic Memory

- `#include <stdlib.h>`
  - `void *malloc(size_t size);`
  - `void free(void *ptr);`

```c
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

Karthik Dantu
• #include <stdlib.h>
  void *malloc(size_t size);
  void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
Heap: Dynamic Memory

- `#include <stdlib.h>
  void *malloc(size_t size);
  void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);`
Heap: Dynamic Memory

- `#include <stdlib.h>`
  - `void *malloc(size_t size);`
  - `void free(void *ptr);`

  ```c
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
  ```
Heap: Dynamic Memory

- `#include <stdlib.h>
  void *malloc(size_t size);
  void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);`
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);