CSCB09 Software Tools and Systems Programming
I/O & Memory Model

Marcelo Ponce

Winter 2025

Department of Computer and Mathematical Sciences - UTSC

Today's lecture

File System

File IO

Arrays & Pointers

Memory Model

File System

File System, Inodes and Blocks

- Components of the OS in charge of managine *files*.

- Interact with lower-level IO subsystems

- All Unix filesystems use two basic components to organize
  and store data: **blocks** and **inodes**.

File System, Inodes and Blocks

- Components of the OS in charge of managine *files*.

- Interact with lower-level IO subsystems

- All Unix filesystems use two basic components to organize and store data: **blocks** and **inodes**.

Similarly as a physical disk is organized into *sectors*, data on a filesystem is abstracted into *blocks*.

File System, Inodes and Blocks

- Components of the OS in charge of managine *files*.

- Interact with lower-level IO subsystems

- All Unix filesystems use two basic components to organize and store data: **blocks** and **inodes**.

Similarly as a physical disk is organized into *sectors*, data on a filesystem is abstracted into *blocks*.

Blocks

- abstractions of data on the filesystem

- have a fixed size, determine at the time the filesystem is created

- the *block size* determines how many bytes are allocated to each block

- e.g. 32-bit: 1 KB, 2 KB, or 4 KB; 64-bit: 8 KB

Inodes

- data structure used to map blocks to physical disk locations

- every file, is assigned an inode

- analogous to pointers

- when a filesystem runs out of inodes, no new files can be created until existing files are deleted

Linux file system Architecture

SRC: https://developer.ibm.com/tutorials/l-linux-filesystem/
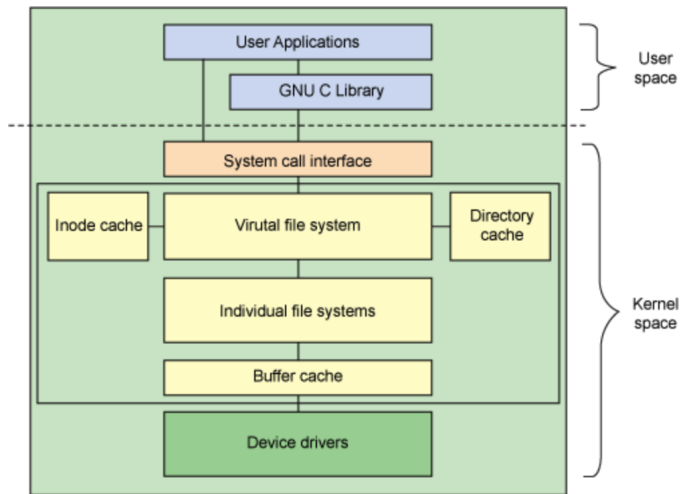
File IO

Basic Input/Output Operations – IOPS

What are I/O operations, or IOPS?

- Finding a file (ls)

  Check if that file exists, read metadata (file size, date stamp etc.)

- Opening a file:

  Check if that file exists, see if opening the file is allowed, possibly create it, find the block that has the (first part of) the file system.

- Reading a file:

  Position to the right spot, read a block, take out right part

- Writing to a file:

  Check where there is space, position to that spot, write the block. Repeated if the data read/written spans multiple blocks.

- Move the file pointer ("seek"):

  File system must check were on disk the data is.

- Close the file.

Basic File I/O functions (`stdio.h`)

```c
// Open
FILE *fopen(const char *path, const char *mode);

// Input:
int fscanf(FILE *stream, const char *format,...);
char *fgets(char *s, int size, FILE *stream);
char fgetc(FILE *stream);

// Output:
int fprintf(FILE *stream, const char *format,...);
int fputs(const char *str, FILE *stream);

// Position cursor:
int fseek(FILE *stream, long int offset, int whence);
void rewind(FILE *stream);

// Close
int fclose(FILE *stream);
```

Some reflections about basics of file I/O and command line arguments...

Questions...?

- Why do we need to open files before reading/writing on them?

- Why do we need to close files when done with them?

- Why did we not have to open STDOUT before we could use printf?

File interfaces in C/Unix

- Two main mechanisms for managing open files:

File interfaces in C/Unix

- Two main mechanisms for managing open files:
- File *descriptors* (low-level, managed by OS):
    - Each open file is identified by a small integer
    - STDIN is 0, STDOUT is 1
    - Use for pipes, sockets (will see later what those are ...)
    - (Remember how I said in first lecture that everything is a file ..?)

File interfaces in C/Unix

- Two main mechanisms for managing open files:
- File *descriptors* (low-level, managed by OS):
    - Each open file is identified by a small integer
    - STDIN is 0, STDOUT is 1
    - Use for pipes, sockets (will see later what those are ...)
    - (Remember how I said in first lecture that everything is a file ..?)
- File *pointers* (aka streams, file handles) for regular files:
    - A C language construct for easier working with file descriptors
    - You use a pointer to a file structure (FILE *) as handle to a file.
    - The file struct contains a file descriptor and a buffer.
    - Use for regular files

Standard File Descriptors

- The operating system (OS) maintains **for each process** a table with open file descriptors: **file descriptor table**

Standard File Descriptors

- The operating system (OS) maintains **for each process** a table with open file descriptors: **file descriptor table**
- The OS opens for each process automatically three standard file descriptors:

	Name	File Descriptor	Default
Standard Input	stdin	0	keyboard
Standard Output	stdout	1	screen
Standard Error	stderr	2	screen

Standard File Descriptors

- The operating system (OS) maintains **for each process** a table with open file descriptors: **file descriptor table**

- The OS opens for each process automatically three standard file descriptors:

	Name	File Descriptor	Default
Standard Input	stdin	0	keyboard
Standard Output	stdout	1	screen
Standard Error	stderr	2	screen

- All C programs automatically have three files stream open:

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```
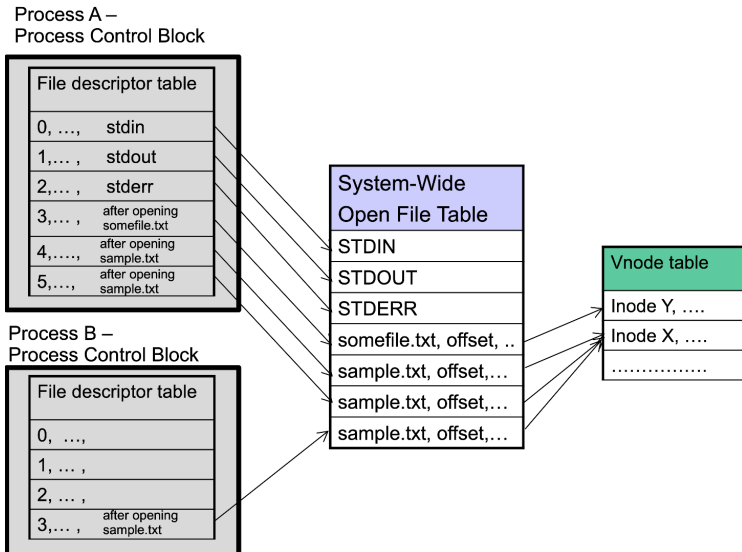
- Those are ready to use:

```
// using fprint and a file handler
fprintf(stdout, "Hello!\n");

// identical to use printf
printf("Hello!\n");
```

OS managing open files (in-memory data structures)

A new entry is added to
the file descriptor table
each time a file is opened

- Find *inode*
- Check permissions
- Initialize cursor to
  beginning of file

Process A –
Process Control Block

File descriptor table

0, ...,	stdin
1, ... ,	stdout
2, ...,	stderr
3, ...,	after opening somefile.txt
4, ...,	after opening sample.txt
5, ...,	after opening sample.txt

Process B –
Process Control Block

File descriptor table

0, ...,	
1, ... ,	
2, ... ,	
3, ...,	after opening sample.txt

System-Wide
Open File Table

STDIN
STDOUT
STDERR
somefile.txt, offset, ..
sample.txt, offset,...
sample.txt, offset,...
sample.txt, offset,...

Vnode table

Inode Y, ...
Inode X, ...
....

Arrays & Pointers

Arrays & Pointers

```c
int x[5];
int *y = NULL;

for (int i = 0; i < 5; i++){
  x[i] = 1;
}
```

y	NULL	0x88681140
x[0]	1	0x88681144
x[1]	1	0x88681148
x[2]	1	0x8868114c
x[3]	1	0x88681150
x[4]	1	0x88681154
	?	0x88681158

```c
int x[5];
int *y = NULL;

for (int i = 0; i < 5; i++){
  x[i] = 1;
}
```

If an array of ints contains 10 ints, then the array size is 40 bytes. There is nothing extra.

- In particular, size is not stored.
- No memory is reserved to hold the address where array starts.

For a pointer variable, space to hold size of an address (4 or 8 bytes) is reserved.

Arrays & Pointers

```
int x[5];
int *y = NULL;
y = x;

for (int i = 0; i < 5; i++)
    {
  y[i] = 1;
}
```

Arrays & Pointers

y		0x88681140
x[0]		0x88681144
x[1]		0x88681148
x[2]		0x8868114c
x[3]		0x88681150
x[4]		0x88681154
		0x88681158

```c
int x[5];
int *y = NULL;
y = x;

for (int i = 0; i < 5; i++)
    {
  y[i] = 1;
}
```

Arrays & Pointers

y		0x88681140	y	0x88681144	0x88681140
x[0]		0x88681144	x[0]	1	0x88681144
x[1]		0x88681148	x[1]	1	0x88681148
x[2]		0x8868114c	x[2]	1	0x8868114c
x[3]		0x88681150	x[3]	1	0x88681150
x[4]		0x88681154	x[4]	1	0x88681154
		0x88681158		?	0x88681158

```
int x[5];
int *y = NULL;
y = x;

for (int i = 0; i < 5; i++)
    {
  y[i] = 1;
}
```

An array name in expression context decays into the array's starting address (address of zero'th element).

However this would **not** be ok: x = y; $\Rightarrow$ No space reserved for x to hold an address.

Any pointer can be used with the array access operator [ ]. $\Rightarrow$ y in the example above can be used like x.

Pointer Arithmetics

- The array access operator [ ] is really only a shorthand for **pointer arithmetic + dereference**
- These are equivalent in C: x[i] == *(x + i)

Pointer Arithmetics

- The array access operator [ ] is really only a shorthand for **pointer arithmetic + dereference**

- These are equivalent in C: `x[i] == *(x + i)`

- The compiler resolves the name of an array to the starting address of the array and adds to it.

- So the program will happily try to access contents at address `*(x+999999)`, even if array size is much smaller than `999999`.

- Behaviour of exceeding array bounds is "undefined"
  - program might appear to work
  - program might crash (segmentation fault)
  - program might do something apparently random

Pointers Types

- Why do pointers have a type, e.g.
  `int*` or `char*`?

Pointers Types

- Why do pointers have a type, e.g.
  int* or char*?

  - Pointer arithmetic needs to
    know the size of object that
    pointer points to so it knows by
    how much to increment to get
    to a[i].

y	NULL	0x88681140
x[0]	1	0x88681144
x[1]	1	0x88681148
x[2]	1	0x8868114c
x[3]	1	0x88681150
x[4]	1	0x88681154
	?	0x88681158

Differences between pointers and arrays

```
/////////////////////////////
int x[5];
int *y = NULL;
y = x;

// what is sizeof(x)
// what is sizeof(y)

func(x);
func(y);
/////////////////////////////
```

```
/////////////////////////////
void func(int x[]) {
  // what is sizeof(x)
}
/////////////////////////////
```

Differences between pointers and arrays

```
/////////////////////////////

int x[5];
int *y = NULL;
y = x;

// what is sizeof(x)
// what is sizeof(y)

func(x);
func(y);

/////////////////////////////
```

```
/////////////////////////////

void func(int x[]) {
  // what is sizeof(x)
}

/////////////////////////////
```

- Inside `main`:

$$\underbrace{\texttt{sizeof(y)}}_{\substack{\text{size of an address} \\ \text{(4 or 8 bytes)}}} \neq \underbrace{\texttt{sizeof(x)}}_{\substack{\text{5 * size of an int} \\ \text{(typically 20 bytes)}}}$$

Differences between pointers and arrays

```
/////////////////////////////
int x[5];
int *y = NULL;
y = x;

// what is sizeof(x)
// what is sizeof(y)

func(x);
func(y);
/////////////////////////////
```

```
/////////////////////////////
void func(int x[]) {
  // what is sizeof(x)
}

/////////////////////////////
```

- Inside `main`:

$$\underbrace{\texttt{sizeof(y)}}_{\substack{\text{size of an address} \\ \text{(4 or 8 bytes)}}} \neq \underbrace{\texttt{sizeof(x)}}_{\substack{\text{5 * size of an int} \\ \text{(typically 20 bytes)}}}$$

- Inside `void func(int x[])`:

  `sizeof(x) == size of an address`

Memory Model

Memory Model

- The memory for a process (a running program) is called its **address space**.
- Memory is just a sequence of bytes.
- A memory location (a byte) is identified by an *address*.

Memory Model

Logical address

0

- The memory for a process (a running program) is called its **address space**.
- Memory is just a sequence of bytes.
- A memory location (a byte) is identified by an *address*.
- In A48, you learned about the "memory model"... that was an "incomplete" picture...
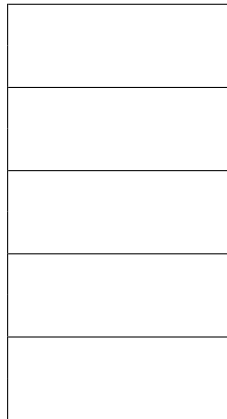
$2^{32} - 1$

The Address Space

- **Static data** Space for the *evil* **global** variables and variables declared as **static**
- **Dynamic data (Heap)** Space for dynamically allocated data structures (`malloc`, `calloc`).
- **Stack** Space for variables created in function calls: a function's parameters and a function's local variables

Logical address

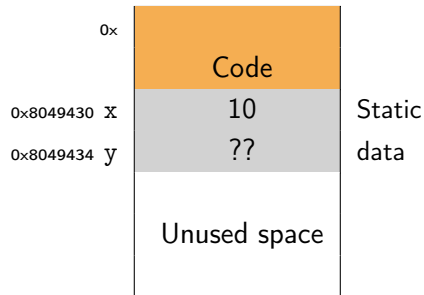| |
|---|
| Code |
| Static data |
| Dynamic data (Heap) ↓ |
| Unused space |
| ↑ Stack |

0

$2^{32} - 1$

Examples

```c
int x = 10;
int y;

int f(int p, int q) {
    int j = 5;
    return p * q + j;
}

int main() {

    int i = x;
    y = f(i, i);

    return 0;
}
```

Logical address

0x
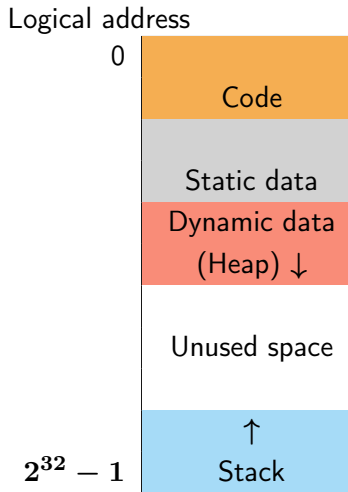
Code

Examples

```c
1  int x = 10;
2  int y;
3
4  int f(int p, int q) {
5      int j = 5;
6      return p * q + j;
7  }
8
9  int main() {
10
11     int i = x;
12     y = f(i, i);
13
14     return 0;
15 }
```
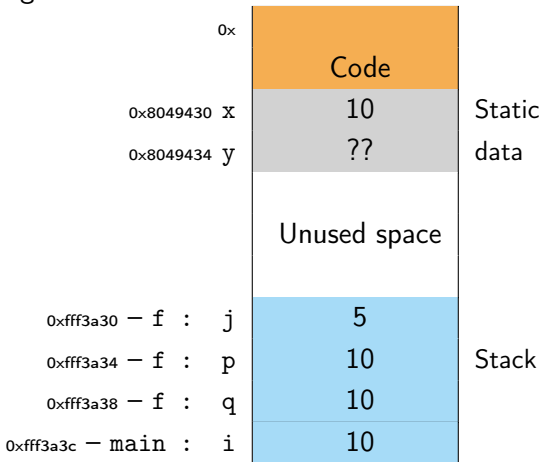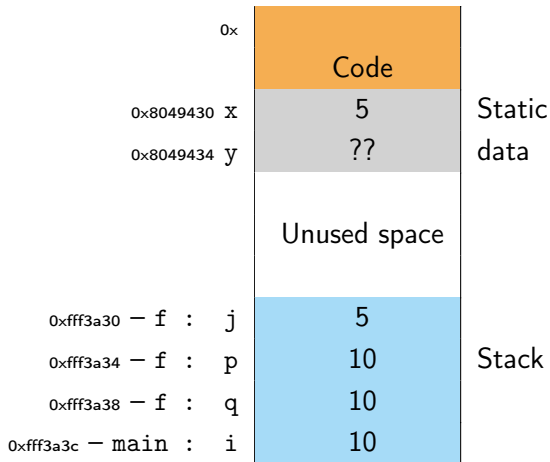
Logical address

0×	Code
0×8049430 X	10
0×8049434 y	??
	Unused space

Static data

Examples

```c
int x = 10;
int y;

int f(int p, int q) {
    int j = 5;
    return p * q + j;
}

int main() {

    int i = x;
    y = f(i, i);

    return 0;
}
```

Logical address

0x	Code	
0x8049430 x	10	Static
0x8049434 y	??	data
	Unused space	
0xfff3a30 — f : j	5	
0xfff3a34 — f : p	10	Stack
0xfff3a38 — f : q	10	
0xfff3a3c — main : i	10	

Examples – Global Variables

```c
int x = 10;
int y;

int f(int p, int q) {
    int j = 5;
    x = 5;
    return p * q + j;
}

int main() {

    int i = x;
    y = f(i, i);

  printf("x=%d, y=%d\n", x, y);
    return 0;
}
```

If `f()` were to modify `x` or `y`, would this change be permanent?

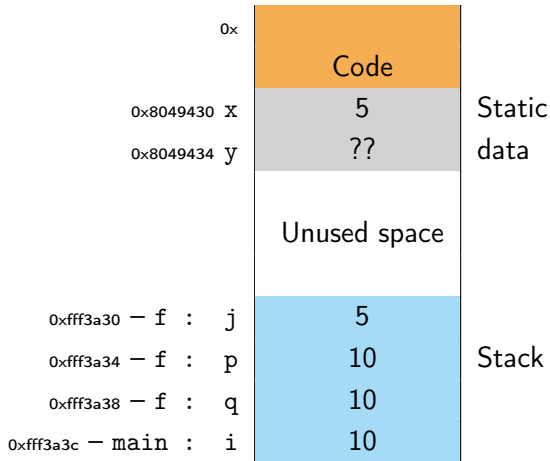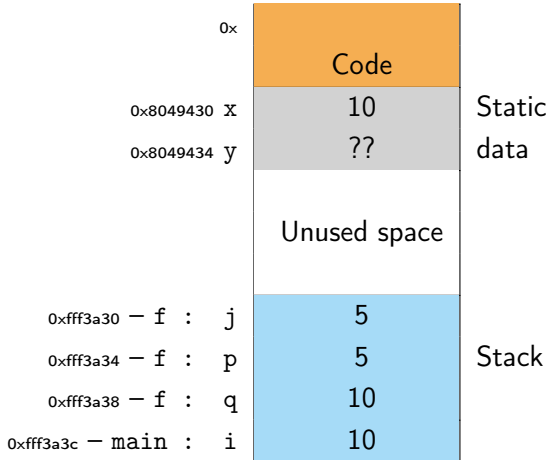|		
0x	Code	
0x8049430 x	5	Static
0x8049434 y	??	data
	Unused space	
0xfff3a30 — f : j	5	
0xfff3a34 — f : p	10	Stack
0xfff3a38 — f : q	10	
0xfff3a3c — main : i	10	

Examples – Global Variables

```c
int x = 10;
int y;

int f(int p, int q) {
    int j = 5;
    x = 5;
    return p * q + j;
}

int main() {

    int i = x;
    y = f(i, i);

  printf("x=%d, y=%d\n",x,y);
    return 0;
}
```

0×		Code
0x8049430 x	5	Static
0x8049434 y	??	data
	Unused space	
0xfff3a30 — f : j	5	
0xfff3a34 — f : p	10	Stack
0xfff3a38 — f : q	10	
0xfff3a3c — main : i	10	

If `f()` were to modify `x` or `y`, would this change be permanent?
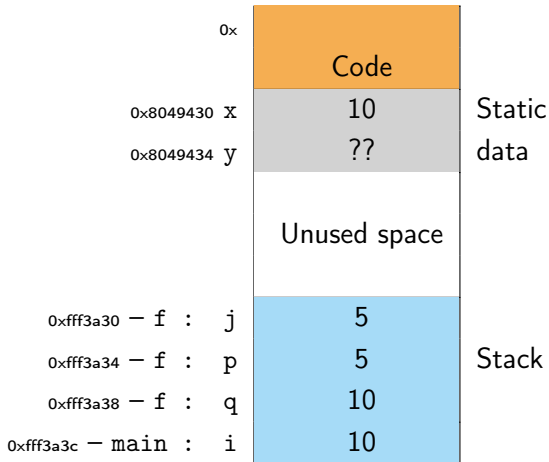
Evil global variables!
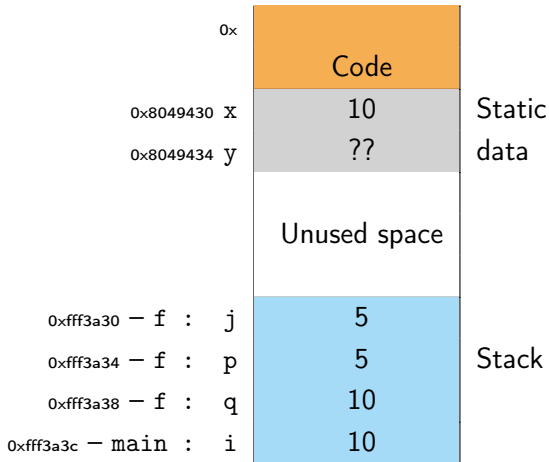
Examples – Passing Function Arguments

```
1  int x = 10;
2  int y;
3
4  int f(int p, int q) {
5      int j = 5;
6      p = 5;
7      return p * q + j;
8  }
9
10 int main() {
11
12     int i = x;
13     y = f(i, i);
14
15     return 0;
16 }
```

If f() were to modify p or q, will that change the value of main's int i?

0x		
	Code	
0x8049430 x	10	Static
0x8049434 y	??	data
	Unused space	
0xfff3a30 — f : j	5	
0xfff3a34 — f : p	5	Stack
0xfff3a38 — f : q	10	
0xfff3a3c — main : i	10	

Examples – Passing Function Arguments

```c
int x = 10;
int y;

int f(int p, int q) {
    int j = 5;
    p = 5;
    return p * q + j;
}

int main() {

    int i = x;
    y = f(i, i);

    return 0;
}
```

If f() were to modify p or q, will that change the value of main's int i?

C passes basic data structures **by value**

0x	
	Code
0x8049430 x	10
0x8049434 y	??
	Unused space
0xfff3a30 − f : j	5
0xfff3a34 − f : p	5
0xfff3a38 − f : q	10
0xfff3a3c − main : i	10

Static data · Stack

Examples – Passing Function Arguments

```
1  int x = 10;
2  int y;
3
4  int f(int p, int q) {
5      int j = 5;
6      p = 5;
7      return p * q + j;
8  }
9
10 int main() {
11
12     int i = x;
13     y = f(i, i);
14
15     return 0;
16 }
```

0x	Code	
0x8049430 x	10	Static
0x8049434 y	??	data
	Unused space	
0xfff3a30 – f : j	5	
0xfff3a34 – f : p	5	Stack
0xfff3a38 – f : q	10	
0xfff3a3c – main : i	10	

If `f()` were to modify `p` or `q`, will that change the value of `main`'s `int i`?

C passes basic data structures **by value** $\Rightarrow$ What change is needed for `f()` to change one of `main`'s variables?

Examples – Passing Function Arguments

```c
int x = 10;
int y;

void f(int *p, int q) {
    *p = 5;
}

int main() {
    int i = x;
    f(&i, i);
    return 0;
}
```
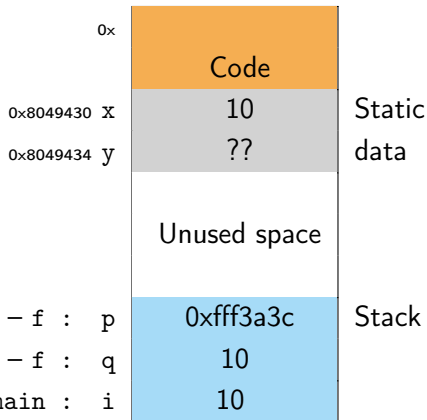
Changes

f(&i,i);

void f(int *p, q)

*p = 5;

Code

0x8049430 X   10   Static

0x8049434 y   ??   data

Unused space

0xfff3a3c — main : i   10

Examples – Passing Function Arguments

```
1  int x = 10;
2  int y;
3
4  void f(int *p, int q) {
5      *p = 5;
6  }
7
8  int main() {
9
10     int i = x;
11     f(&i, i);
12     return 0;
13 }
```
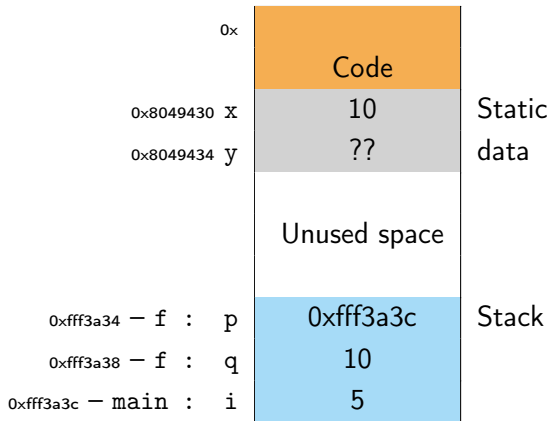
When calling `f()`

Changes

f(&i,i);

void f(int *p, q)

*p = 5;

Examples – Passing Function Arguments

```c
int x = 10;
int y;

void f(int *p, int q) {
    *p = 5;
}

int main() {

    int i = x;
    f(&i, i);
    return 0;
}
```

After `f()` returns

Changes

`f(&i,i);`

`void f(int *p, q)`

`*p = 5;`

The Address Space, continued...

- **Static data**
  .data = global or static variables with predefined values that can be modified
  BSS segment = global/static without predefined values
- **Dynamic data (Heap)**
- **Stack**
- **Code/Text**

Logical address

0
Code
Static data
Dynamic data (Heap) ↓
Unused space
↑
$2^{32} - 1$ Stack

Dynamic Memory Management

Let's write our own `concat` function:

```
1
2  char *concat(const char *s1,
       const char *s2) {
3     char result[70];
4
5     strcpy(result, s1);
6     strcat(result, s2);
7
8     return result;
9  }
```

Any problems with this implementation?

Dynamic Memory Management

Let's write our own `concat` function:

```c
1
2 char *concat(const char *s1,
      const char *s2) {
3     char result[70];
4
5     strcpy(result, s1);
6     strcat(result, s2);
7
8     return result;
9 }
```

Any problems with this implementation?

- `strncpy` & `strncat` would be safer..
- But what else?

Dynamic Memory Management

Let's write our own `concat` function:

```c
char *concat(const char *s1,
    const char *s2) {
    char result[70];

    strcpy(result, s1);
    strcat(result, s2);

    return result;
}
```

```c
char *concat(const char *s1, const char *s2)
{
    // temp local variable to store result
    char *result;

    // allocating enough memory
    result = malloc(strlen(s1) + strlen(s2)
        + 1);
    if (result == NULL) {
        printf ( Error: malloc failed\ n
            );
        exit(1);
    }
    //

    strcpy (result, s1);
    strcat (result, s2);

    return result;
}
```

The Address Space, continued...

- **Static data**
- **Dynamic data (Heap)**
  Space for dynamically allocated data structures (`malloc`, `calloc`).
  - Memory allocated here will never be released/freed automatically by the system.
  - Completely under programmers control.
  - Memory here can be allocated at any time during the run of a program
- **Stack**
- **Code/Text**

Logical address

Differences between `char` pointers and `char` arrays

```
// char array
char a[] = "array";

// char pointer
// AKA "string literal"
char *p = "pointer";
```

```
p[0] = 'z';    // Illegal!
a[0] = 'z';    // OK!
```

"pointer" is stored in **read-only memory**.

No other space is reserved for p, except to store a memory address.

The *Complete* Memory Model

- **Static data**
  Space for the *evil* **global** variables and variables declared as **static**

- **Dynamic data (Heap)**
  Space for dynamically allocated data structures (`malloc`, `calloc`).

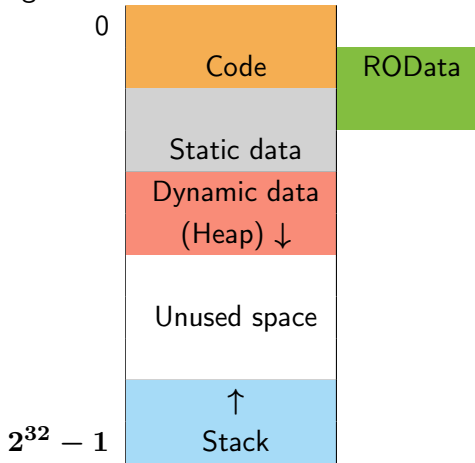- **Stack**
  Space for variables created in function calls: a function's parameters and a function's local variables

- **Code/Text**
  string literals: RODatas (read-only data)
  code/text, or static data depending on platform

Logical address

0

Code	ROData
Static data	
Dynamic data (Heap) ↓	
Unused space	

$2^{32} - 1$

Stack ↑

Passing Arrays as Parameters

Suppose we want to write a function to sum an array of ints.

```c
int main()
{
  int i[3] = {10, 9, 8};

  printf("sum is %d\n", sum(i)); /*??*/

  return 0;
}

int sum( /* What goes here? */ ) {

}
```

What is being passed to the function is the name of the array which *decays* to a pointer to the first element – a pointer of type int.

Passing Arrays as Parameters

Suppose we want to write a function to sum an array of ints.

```c
int main()
{
   int i[3] = {10, 9, 8};

   printf("sum is %d\n", sum(i)); /*??*/

   return 0;
}

int sum( /* What goes here? */ ) {

}
```

```c
int sum( int *a ) {
   int i, s = 0;

   for(i = 0; i < ??; i++)
      s += a[i];   /* this is
          legal */

   return s;
}
```

Passing Arrays as Parameters

Suppose we want to write a function to sum an array of ints.

```
int main()
{
  int i[3] = {10, 9, 8};

  printf("sum is %d\n", sum(i)); /*??*/

  return 0;
}

int sum( /* What goes here? */ ) {

}
```

```
int sum( int *a ) {
  int i, s = 0;

  for(i = 0; i < ??; i++)
    s += a[i];   /* this is
        legal */

  return s;
}
```

- How do we know how big the array is?
- Pass in the *size* of the array as another parameter.

Array Parameters

- `int sum(int *a, int size)`
- Also valid C code is:
  `int sum(int a[], int size)`

Array Parameters

- `int sum(int *a, int size)`

- Also valid C code is:

  `int sum(int a[], int size)`

- However, many reasons against using this form:
  - We are really passing a pointer-to-int **not** an array.
  - We still don't know how big the array is.
  - Outside of a *formal parameter declaration* `int a[];` is **illegal**

- `int a;` and `int a[10];` are completely different things

- Arrays are **not** pointers
- "Equivalence" of pointers and arrays
  Decay: An *lvalue* of type *array-of-T* which appears in an expression **decays** (with three exceptions) into a pointer to its first element; the type of the resultant pointer is pointer-to-T.
  (The exceptions are when the array is the operand of a sizeof or & operator, or is a *literal string* initializer for a character array.)
- Array and pointer declarations are interchangeable as function formal parameters
- Arrays automatically allocate space, but can't be relocated or resized.
  Pointers must be explicitly assigned to point to allocated space (perhaps using malloc), but can be reassigned (i.e. pointed at different objects) at will, and have many other uses besides serving as the base of blocks of memory.

- Due to the so-called equivalence of arrays and pointers, arrays and pointers often seem interchangeable, and in particular a pointer to a block of memory assigned by `malloc` is frequently treated (and can be referenced using [] exactly) as if it were a true array.