CSC 6580
Spring 2020

Instructor: Stacy Prowell

Homework:

Structuring 1

Starting with the provided solution to the basic block homework, or with your own solution, apply the constructive proof of the structure theorem. We will do this in pieces. For next time:

1. Find the entry point and see if it appears the C runtime is in use. If so, figure out where `main` is and add it to the addresses to extract.
2. Create a Python class for a Node. There should be three types of Node, possibly by subclass.
   a. A function node that holds a basic block that has a single next address or no next address ("unknown")
   b. A predicate node that holds a basic block that has two next addresses, one for true and one for false
   c. A label assignment node that holds an address to assign to the label
3. Package all basic blocks into Node instances. Don't worry about the label assignments yet.
4. Don't worry about output yet. Turn your program in by Tuesday.

Please name your program `structure.py`.

Structuring 1

Full discussion of this in lecture from 2/13: ELF, the entry point, and main.

Starting the C runtime

At entry (from the loader):

- RDX contains the address of the destructor function call handler for the dynamic linker, `_dl_fini`.
- The stack contains `argc`, `argv`, and `envp`, with `argc` on top.

When we call `main`, we at least want:

- EDI to contain `argc`
- ESI to contain the pointer to `argv`
- EDX to contain the pointer to `envp`

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Is the C runtime being used?

We can use a *heuristic* to figure this out.

Go to the entry point. Extract the basic block there. If there is a single basic block, and if that basic block ends in a call to an address that is out of scope, and if `EDI` is set to an address that is in scope, then we assume that the address in EDI is `main`.

I created the classes first, then ran the program, building the data structures during the second pass (instead of printing them). I then performed the check above and (potentially) re-started with the computed `main` address.

Memory in Assembly

Aside:
Memory Management

Direct Access

For some simple processors, or for the X86 in real mode, there is *no memory management*. You just **directly access** memory as you wish. There is no notion of allocation or deallocation, unless you, yourself, create it.

Typically you would build a *memory map* for your application. This would identify blocks of memory, single addresses, or even bits and what each was used for.

If you wanted *dynamic* allocation, you probably needed to keep track of the top of the heap, and increment that when needed. So long as this was below the stack (and the stack stayed above it) life was good.

Memory Area: pacman_map

Address Range	Length	Function	Description
0x0000-0x3FFF	16384	Mirror, ROM	0x8000,
0x4000-0x43FF	1024	Mirror, RAM Write, Shared	0xa000, pacman_videoram_w, videoram
0x4400-0x47FF	1024	Mirror, RAM Write, Shared	0xa000, pacman_colorram_w, colorram
0x4800-0x4BFF	1024	Mirror, Read, Write NOP	0xa000, pacman_read_nop,
0x4C00-0x4FEF	1008	Mirror, RAM	0xa000,
0x4FF0-0x4FFF	16	Mirror, RAM, Shared	0xa000, , spriteram
0x5000	1	Mirror, Write	0xaf38, irq_mask_w
0x5000	1	Mirror, Read Port	0xaf3f, IN0 0x0001 Joystick Up Active Low 0x0002 Joystick Left Active Low 0x0004 Joystick Right Active Low 0x0008 Joystick Down Active Low 0x0010 Off Active High 0x0000 On Active High 0x0020 Coin 1 Active Low 0x0040 Coin 2 Active Low 0x0080 Service 1 Active Low
0x5001	1	Mirror, Device Write	0xaf38, namco, namco_device, pacman_sound_enable_w
0x5002	1	Mirror, Write NOP	0xaf38,
0x5003	1	Mirror, Write	0xaf38, pacman_flipscreen_w
0x5004-0x5005	2	Mirror, Write NOP	0xaf38, (// AM_WRITE(pacman_leds_w))
0x5006	1	Mirror, Write NOP	0xaf38, (// AM_WRITE(pacman_coin_lockout_global_w))
0x5007	1	Mirror, Write	0xaf38, pacman_coin_counter_w
0x5040	1	Mirror, Read Port	0xaf3f, IN1 0x0001 Joystick Up Active Low 0x0002 Joystick Left Active Low 0x0004 Joystick Right Active Low 0x0008 Joystick Down Active Low 0x0020 Start 1 Active Low 0x0040 Start 2 Active Low 0x0080 Upright Active High 0x0000 Cocktail Active High
0x5040-0x505F	32	Mirror, Device Write	0xaf00, namco, namco_device, pacman_sound_w
0x5060-0x506F	16	Mirror, Write Only, Shared	0xaf00, , spriteram2
0x5070-0x507F	16	Mirror, Write NOP	0xaf00,
0x5080	1	Mirror, Write NOP	0xaf3f,
0x5080	1	Mirror, Read Port	0xaf3f, DSW1 0x0003 Coinage Active High 0x0003 2C_1C Active High 0x0001 1C_1C Active High 0x0002 1C_2C Active High 0x0000 Free_Play Active High 0x000c Lives Active High 0x0000 1 Active High 0x0004 2 Active High 0x0008 3 Active High 0x000c 5 Active High 0x0030 Bonus_Life Active High 0x0000 10000 Active High 0x0010 15000 Active High 0x0020 20000 Active High 0x0030 None Active High 0x0040 Difficulty Active High 0x0040 Normal Active High 0x0000 Hard Active High 0x0080 Normal Active High 0x0000 Alternate Active High
0x50C0	1	Mirror, Device Write	0xaf3f, watchdog, watchdog_timer_device, reset_w
0x50C0	1	Mirror, Read Port	0xaf3f, DSW2 0x00ff Unused Active High

Memory Area: writeport

Address Range	Length	Function	Description
0x0000	1	Write	pacman_interrupt_vector_w (/* Pac-Man only */)

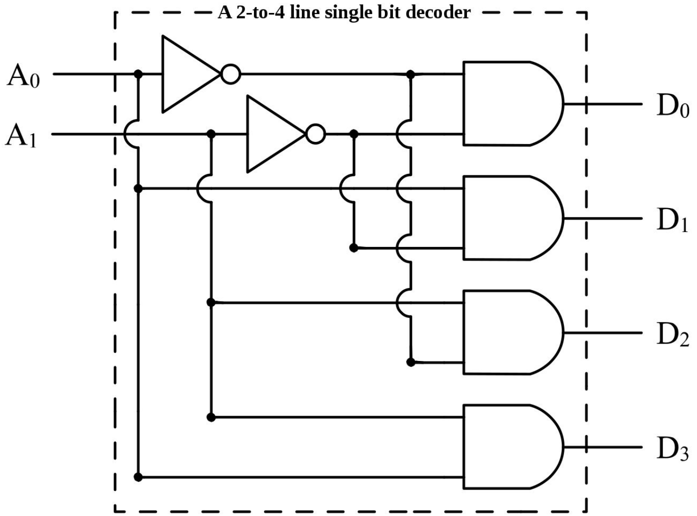
Direct Access

This direct access model works well if you have some discipline, plan ahead, and are (typically) only running one process.

If you only have a 16-bit address space (like so many processors did and some still do), how do you deal with more than 64KiB?

- **Bank switching**! Pick a special address and write to it. This address is hardware-mapped, and latches a value into a special register that feeds that value to a decoder (reads a binary value, and uses it to set one of *n* lines high or low). The decoder feeds to the chip select inputs on the different memory banks *et voilà!* You have selected a bank of (up to) 64KiB. This approach is still used!

Further Aside: Binary Decoders



Truth Table

A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

source: Wikipedia

Given an *n* bit input, a binary decoder translates this into a 2^n bit output, where only one bit is on.

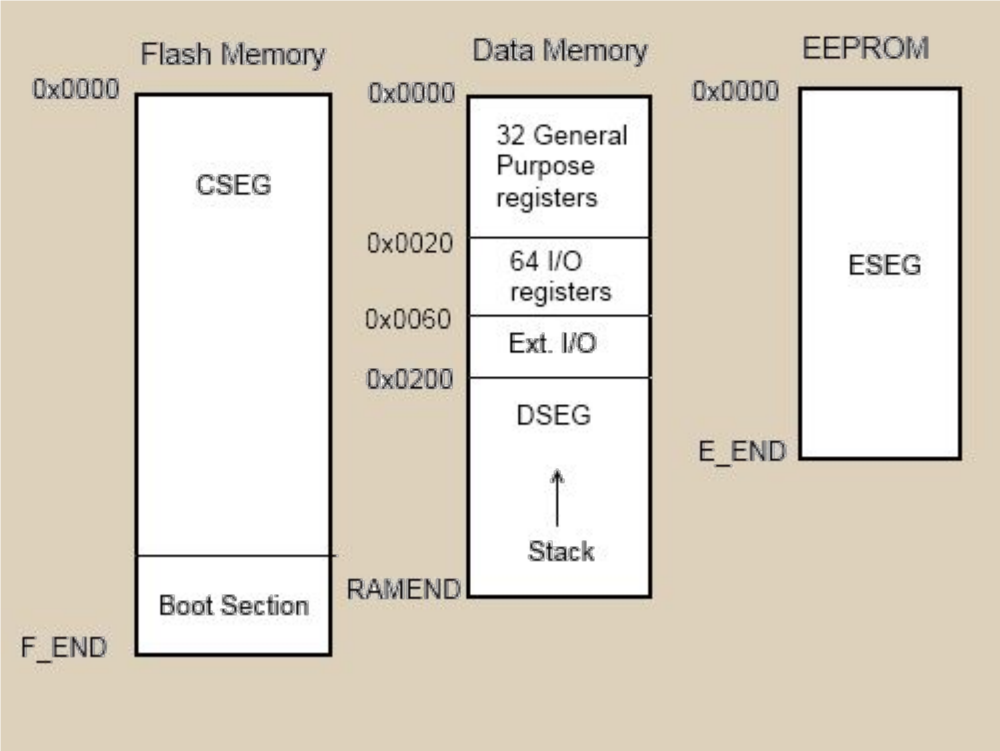
That is, the input bits "select" a single bit in a larger space.

This can be fed to an enable line on other hardware to select that hardware.

Direct Access / Harvard Model

Recall: The **Harvard architecture** has separate storage and instruction pathways for *data* and *program*. A **modified Harvard architecture** allows program memory to be accessed as data.

The Arduino (using the AVR) has a modified Harvard architecture, with the register file mapped into memory.



MMU

A **memory management unit** (MMU) translates addresses from the CPU into physical addresses. There are big advantages here.

- Memory protection. Processes can isolate their memory space from other processes.
- Memory fragmentation. Helps solve the fragmentation problem because of same-size pages.
- Address translation. Different processes can use the same (virtual) address space.
- Caching. Often provides for caching of memory, providing faster access.
- Large memory. Access memory larger than the processor address space.
- Error detection. You can catch some bugs early thanks to memory protection.

Divide memory into *pages* and keep track of those pages. Uses a *page table* (to keep track of permissions and allocation) and a *translation lookaside buffer* (to keep track of address translation).

MMU

External MMU chips exist to add memory management to processors without it.

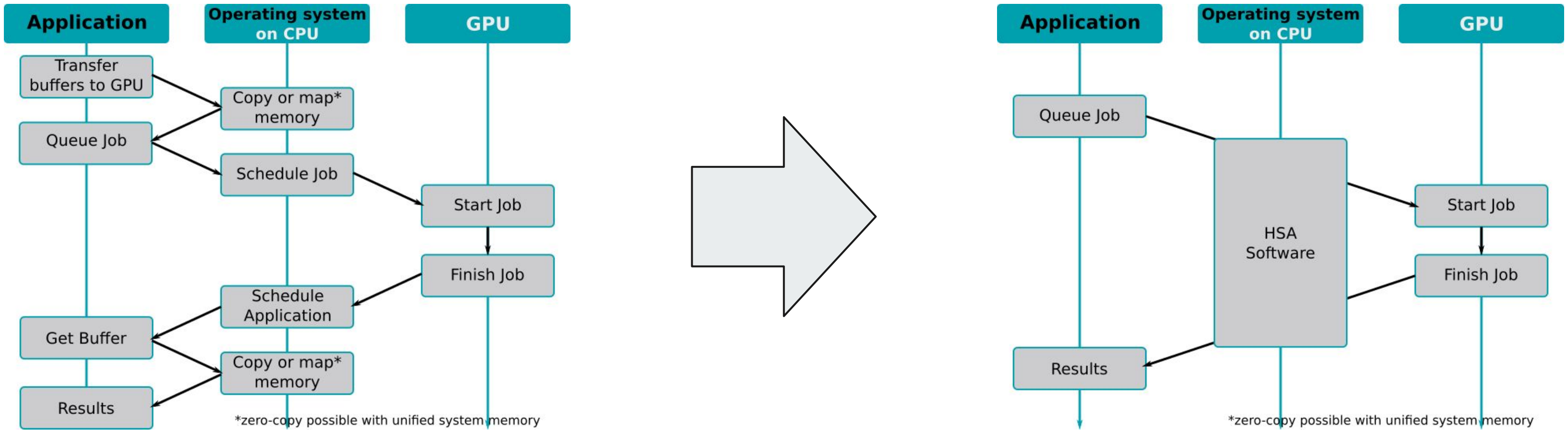
Nearly all modern processors have an MMU built in.

The IBM System/360 had an MMU in August 1965. The X86 MMU has many operating modes and is quite complex; in fact, it is Turing complete! https://github.com/jbangert/trapcc

Watch as an X86 plays Conway's Game of Life without executing any instructions!
https://www.youtube.com/watch?v=eSRcvrVs5ug

Heterogeneous System Architecture (HSA)

Integrate the CPU and GPU on the same bus with shared memory and tasks.



http://www.hsafoundation.com/

Memory in Assembly

You need to understand all this if you are going to write your own operating system. Otherwise, you can just use the operating system.

Linux Memory Management

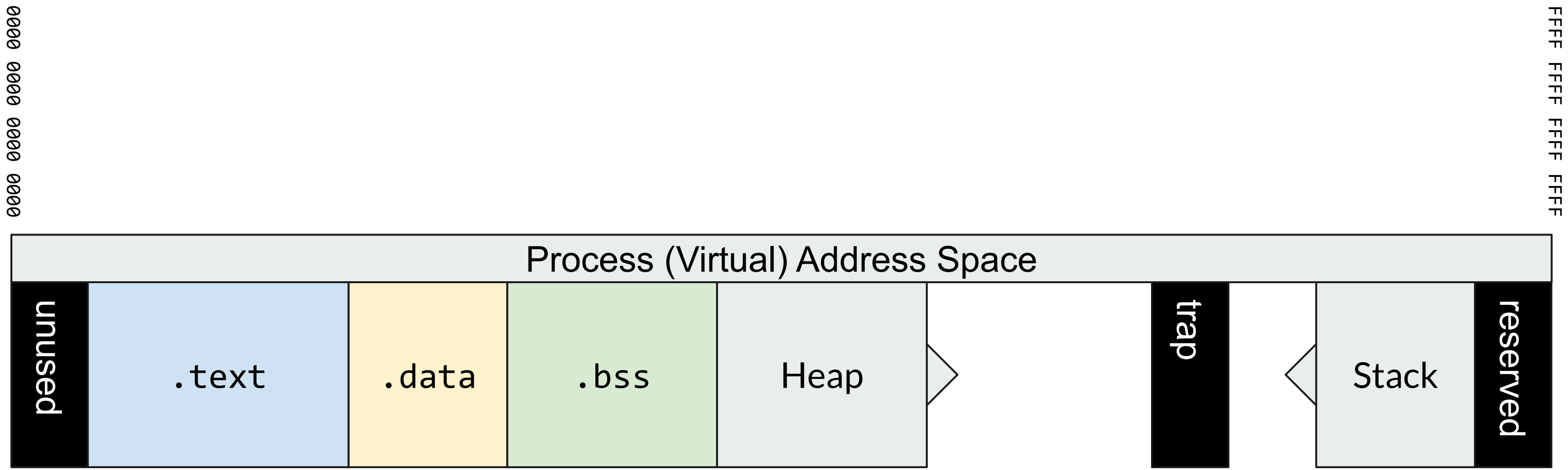
Linux Memory Management

Linux memory management looks like memory management in most operating systems (except some like VxWorks).

Each process has a (virtual) memory map.

- Executable code (.text) *near* the bottom of memory (address 0)

Simplified Linux Process Memory



What's *really* in my process address space?

```
$ cat /proc/10715/maps
```

Process number

You can find out a lot about your process by looking in the `/proc` file system. There will be an entry for each process by process id, and `maps` will give you the process memory map.

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95                /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95                /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95                /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                  [heap]
```

Start address

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95              /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95              /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95              /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                [heap]
```

End address

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95              /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95              /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95              /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                [heap]
```

Start End

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95             /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95             /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95             /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0               [heap]
```

Permissions

```
  r = read
  w = write
  x = execute
p/s = private/shared
```

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95             /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95             /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95             /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0               [heap]
```

File offset (if relevant)

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95              /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95              /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95              /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                [heap]
```

Major:Minor device number (if from a file)

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95                /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95                /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95                /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                  [heap]
```

File number

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95             /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95             /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95             /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0               [heap]
```

Path or content

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95                     /bin/less     .text (r-xp)
56208ab53000-56208ab54000 r--p 00025000 103:05 95                     /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95                     /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                       [heap]
```

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95             /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95             /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95             /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0               [heap]
```

.data (r--p)

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95                /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95                /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95                /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                  [heap]
```

.bss (rw-p)

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95                 /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95                 /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95                 /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                   [heap]
```

Heap

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95             /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95             /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95             /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0               [heap]
```

The program break

The program break is the highest address available for use by the program. It is the end of the heap.

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95             /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95             /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95             /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0               [heap]
```

The program break

It is managed by the operating system, and can be changed by the `sys_brk` system call.

sys_brk

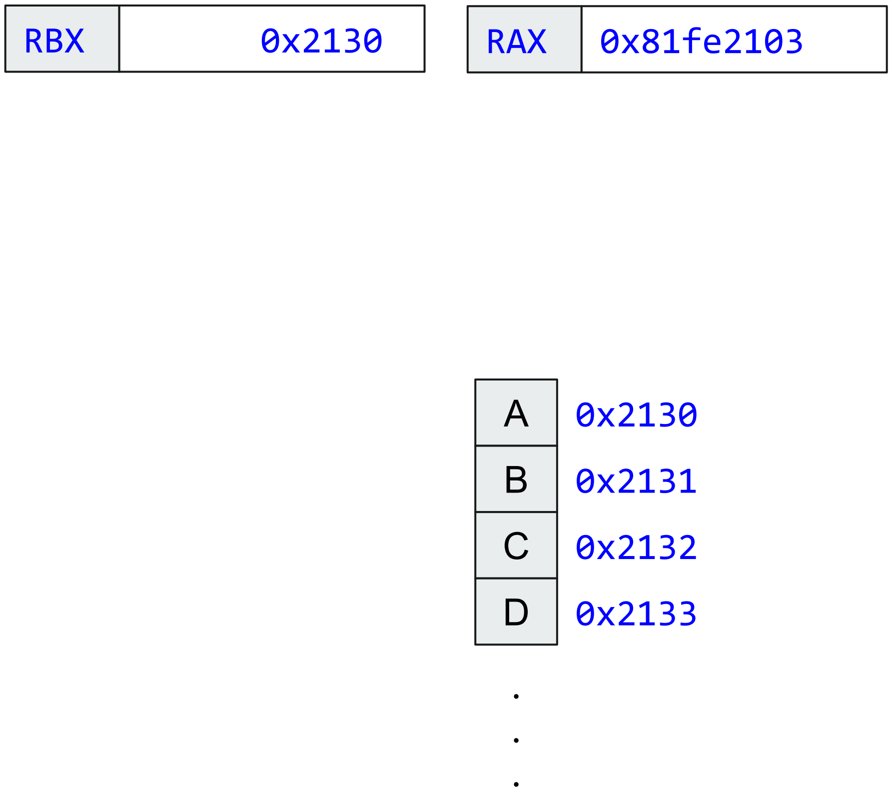
%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
12	sys_brk	unsigned long brk					

New break address

https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

Assembly:
Simple lookup tables

xlat / xlatb



(They're the same; just use `xlatb` and expect to see `xlat` in disassemblies.)

This instruction looks up a byte in a table and loads it into `AL`.

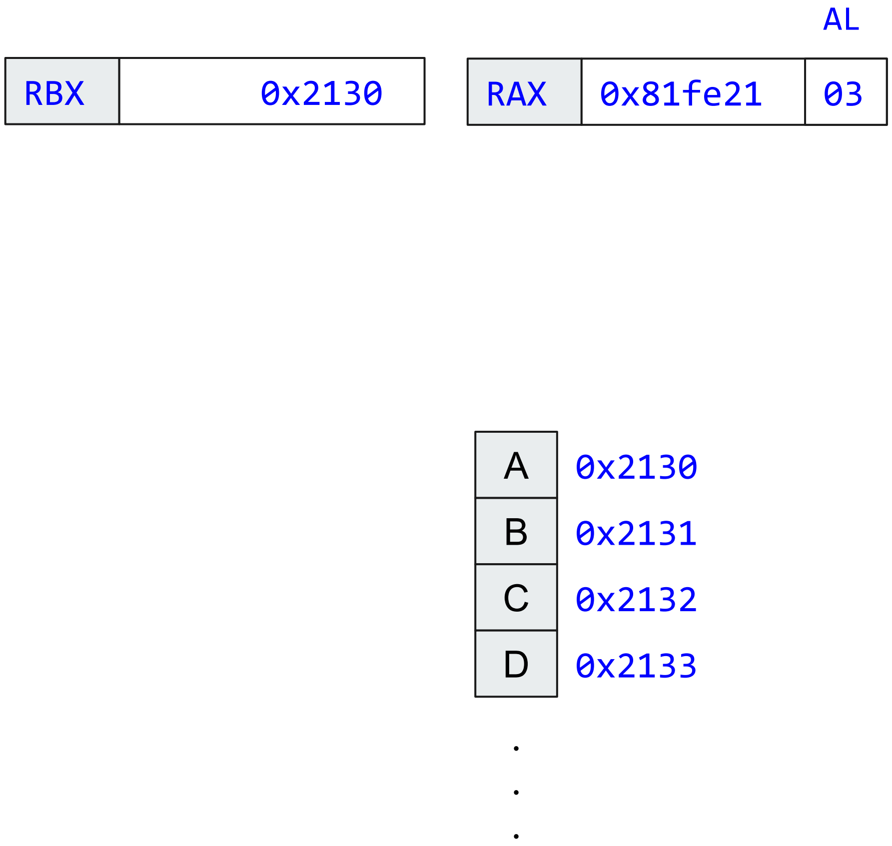
It is essentially: `mov ah, [rbx+ah]` (which isn't an actual instruction because the register sizes are different), without setting flags.

xlat / xlatb

(They're the same; just use `xlatb` and expect to see `xlat` in disassemblies.)

This instruction looks up a byte in a table and loads it into `AL`.

It is essentially: `mov ah, [rbx+ah]` (which isn't an actual instruction because the register sizes are different), without setting flags.

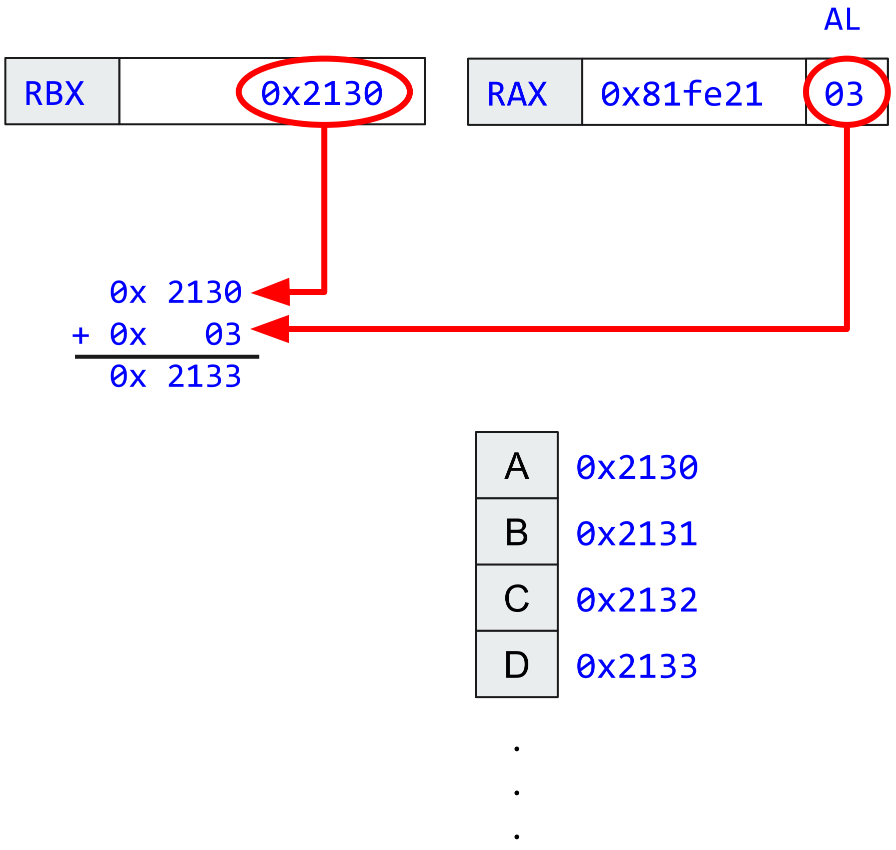


xlat / xlatb

(They're the same; just use `xlatb` and expect to see `xlat` in disassemblies.)

This instruction looks up a byte in a table and loads it into `AL`.

It is essentially: `mov ah, [rbx+ah]` (which isn't an actual instruction because the register sizes are different), without setting flags.

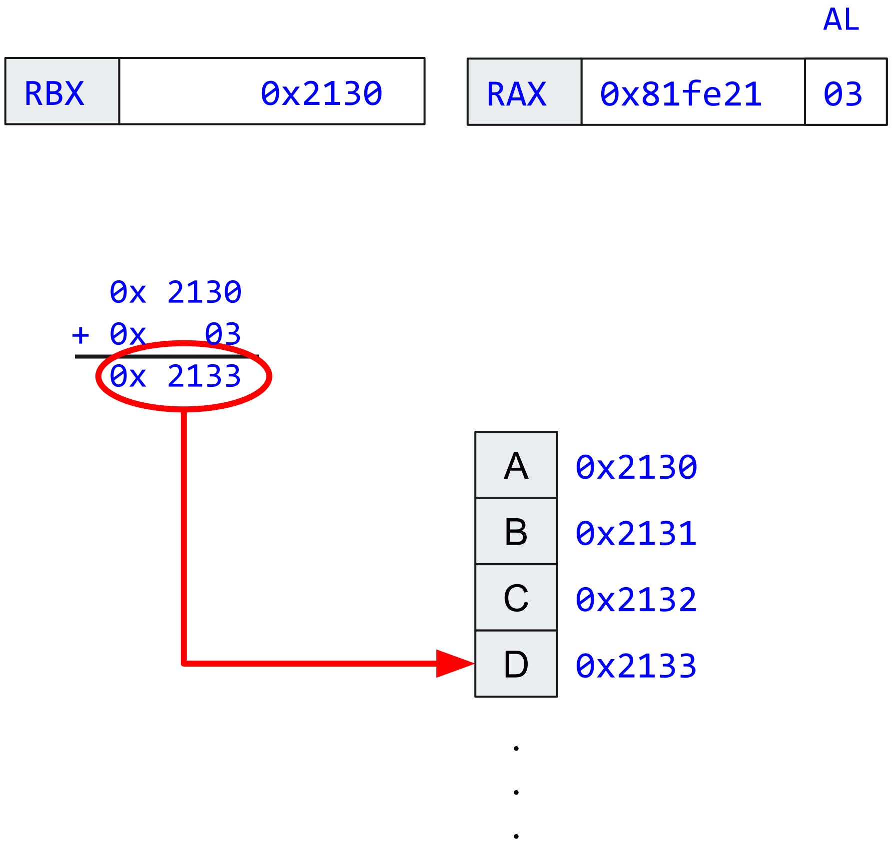


xlat / xlatb

(They're the same; just use `xlatb` and expect to see `xlat` in disassemblies.)

This instruction looks up a byte in a table and loads it into `AL`.

It is essentially: `mov ah, [rbx+ah]` (which isn't an actual instruction because the register sizes are different), without setting flags.

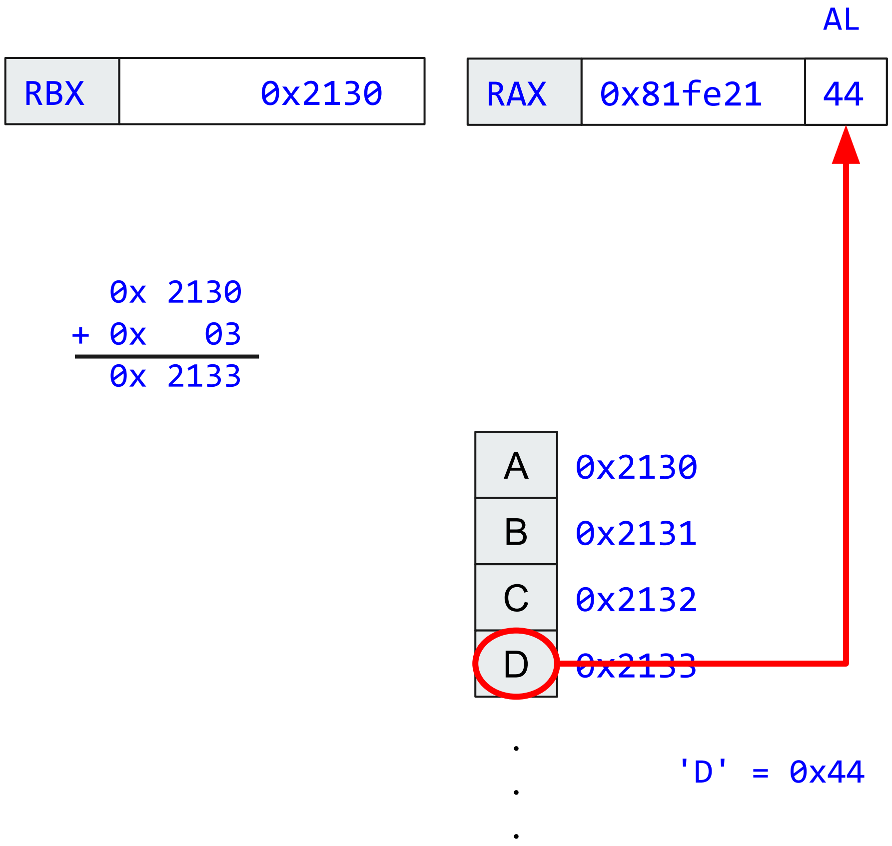


xlat / xlatb

(They're the same; just use `xlatb` and expect to see `xlat` in disassemblies.)

This instruction looks up a byte in a table and loads it into `AL`.

It is essentially: `mov ah, [rbx+ah]` (which isn't an actual instruction because the register sizes are different), without setting flags.



Example:
Manipulating brk

Back To the Memory Map

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95              /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95              /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95              /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                [heap]
7f951c427000-7f951c997000 r--p 00000000 103:05 86              /usr/lib/locale/locale-archive
```

Localized strings

Try: `strings /usr/lib/locale/locale-archive`

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95                    /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95                    /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95                    /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                      [heap]
7f951c427000-7f951c997000 r--p 00000000 103:05 86                    /usr/lib/locale/locale-archive
7f951c997000-7f951c99a000 rw-p 00000000 00:00 0
7f951c99a000-7f951c9bf000 r--p 00000000 103:05 4349                  /lib/x86_64-linux-gnu/libc-2.30.so
. . .
7f951cc06000-7f951cc07000 rw-p 0002c000 103:05 4331                  /lib/x86_64-linux-gnu/ld-2.30.so
```

Loaded shared libraries

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95          /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95          /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95          /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0            [heap]
7f951c427000-7f951c997000 r--p 00000000 103:05 86          /usr/lib/locale/locale-archive
7f951c997000-7f951c99a000 rw-p 00000000 00:00 0
7f951c99a000-7f951c9bf000 r--p 00000000 103:05 4349        /lib/x86_64-linux-gnu/libc-2.30.so
. . .
7f951cc06000-7f951cc07000 rw-p 0002c000 103:05 4331        /lib/x86_64-linux-gnu/ld-2.30.so
7f951cc07000-7f951cc08000 rw-p 00000000 00:00 0
7fff1d961000-7fff1d982000 rw-p 00000000 00:00 0            [stack]  The stack
```

Note that between the heap and the stack is stuff. If either grows into this area, it's an exception.

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95             /bin/less
```

Fast system calls. This maps some systemcalls into user space along with some data, so the calls happen fast.

`vsyscall` is an older method that is very limited and not secure.

The virtual dynamic shared object (vDSO) is a library exported by the kernel containing system calls that do not necessarily have to run in kernel space. The data needed is exported to `vvar`.

```
7f951cc07000-7f951cc08000 rw-p 00000000 00:00 0
7fff1d961000-7fff1d982000 rw-p 00000000 00:00 0                [stack]
7fff1d98a000-7fff1d98d000 r--p 00000000 00:00 0                [vvar]
7fff1d98d000-7fff1d98e000 r-xp 00000000 00:00 0                [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0        [vsyscall]
```

What's in my process address space?

```
$ cat /proc/10715/maps
56208a92e000-56208a954000 r-xp 00000000 103:05 95                 /bin/less
56208ab53000-56208ab54000 r--p 00025000 103:05 95                 /bin/less
56208ab54000-56208ab58000 rw-p 00026000 103:05 95                 /bin/less
56208ab58000-56208ab5c000 rw-p 00000000 00:00 0
56208c62d000-56208c64e000 rw-p 00000000 00:00 0                   [heap]
7f951c427000-7f951c997000 r--p 00000000 103:05 86                 /usr/lib/locale/locale-archive
7f951c997000-7f951c99a000 rw-p 00000000 00:00 0
7f951c99a000-7f951c9bf000 r--p 00000000 103:05 4349               /lib/x86_64-linux-gnu/libc-2.30.so
. . .
7f951cc06000-7f951cc07000 rw-p 0002c000 103:05 4331               /lib/x86_64-linux-gnu/ld-2.30.so
7f951cc07000-7f951cc08000 rw-p 00000000 00:00 0
7fff1d961000-7fff1d982000 rw-p 00000000 00:00 0                   [stack]
7fff1d98a000-7fff1d98d000 r--p 00000000 00:00 0                   [vvar]
7fff1d98d000-7fff1d98e000 r-xp 00000000 00:00 0                   [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0           [vsyscall]
```

Memory Management Using the Standard Library

malloc / calloc / realloc / free

These are library routines declared in `stdlib.h` and available in `libc.so`.

You call them like any other function.

```
extern malloc

mov edi, 64*1024
call malloc wrt ..plt
; Pointer to newly-allocated memory is in RAX
```

Don't like the standard library or have special needs for a constant time, fast, or fixed memory malloc? There are a lot of alternatives you can use!

Homework
Due: Tuesday, April 21

Structuring

Now that you have the program and some basic data structures, create two new data structures:

- An if-then-else data structure that holds a basic block ending in true / false branching
- A sequence data structure that holds a series of (sequential) basic blocks.

Now for each basic block you found, create a prime (if-then-else or sequence). Let the exit be label 0. For now treat any block that ends with an unknown destination as an exit. Keep track of the number of times a label is referenced and, for any label referenced only once (except the entry point and the exit) substitute the label setting block with the corresponding prime. Be careful to avoid an infinite recursion!

Structuring

```
if
  lea rcx, [rax]
  jz 0x2132
then
  mov edi, 1
  jmp 0x214f
  L = 0x214f
else
  call 0x215a
  L = 0x214f
fi
```

```
mov edi, 1
jmp 0x214f
L = 0x214f
```

Next Time: SSA