SLS Lecture 14 : Assembly using the OS

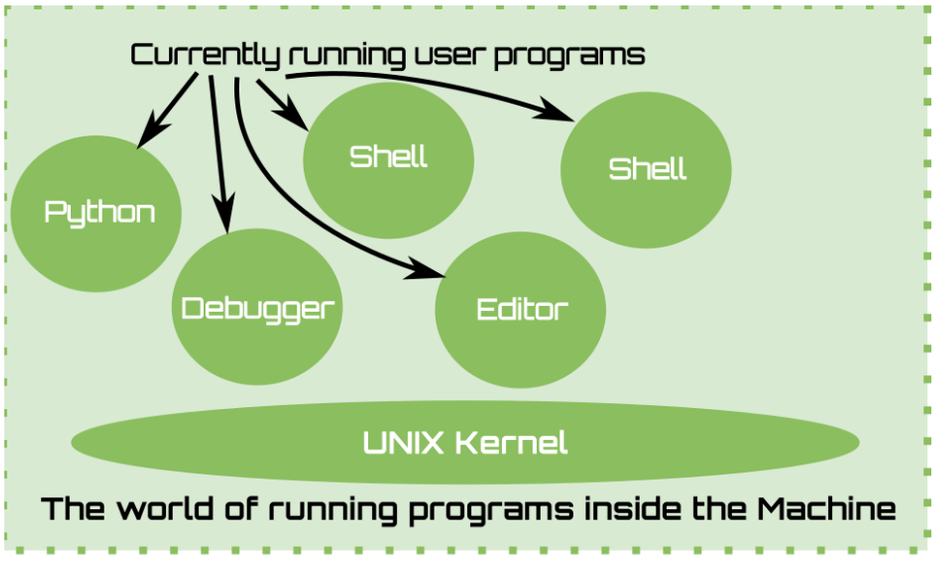
Contents

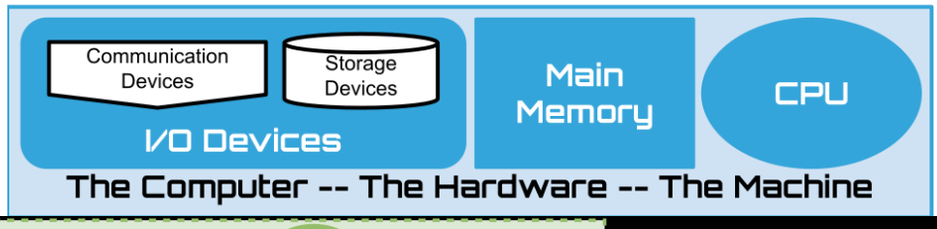
- Review of Processes
- Review of Systems Calls
- I/O - Assembly "Hello World" and read
- Process Address Spaces
  - Dynamic Memory : Adding and Growing our heap

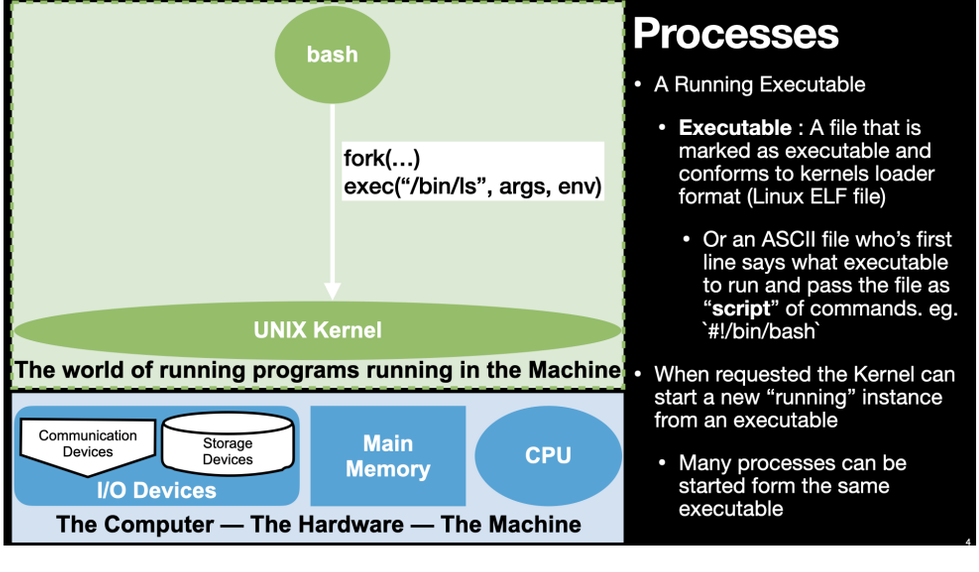
  - create a directory `mkdir syscalls; cd syscalls`
  - copy examples
  - add a `Makefile` to automate assembling and linking
    - we are going run the commands by hand this time to highlight the details
  - add our `setup.gdb` to make working in gdb easier
  - normally you would want to track everything in git

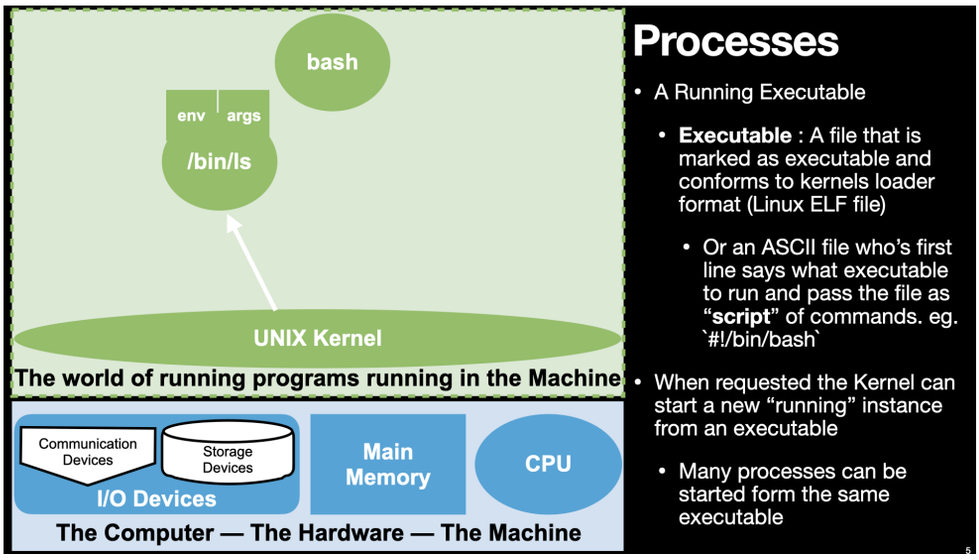
14.1. Review of Processes

14.1.1. Processes and the Kernel









14.2. Review of System Calls - From Lecture 08

ker·nel

/ˈkərnl/

See definitions in:

All | Botany | Computing | Linguistics

noun

noun: **kernel**; plural noun: **kernels**

a softer, usually edible part of a nut, seed, or fruit stone contained within its hard shell.

Similar: seed | grain | heart | core | stone | nut | meat

- the seed and hard husk of a cereal, especially wheat.
- the central or most important part of something.
  "this is the kernel of the argument"

  Similar: essence | core | heart | essential part | essentials | quintessence
- the most basic level or core of an operating system of a computer, responsible for resource allocation, file management, and security.
- LINGUISTICS
  denoting a basic unmarked linguistic string.
  modifier noun: **kernel**

Origin

ENGLISH corn → OLD ENGLISH cyrnel → kernel

Old English *cyrnel*, diminutive of corn[1].

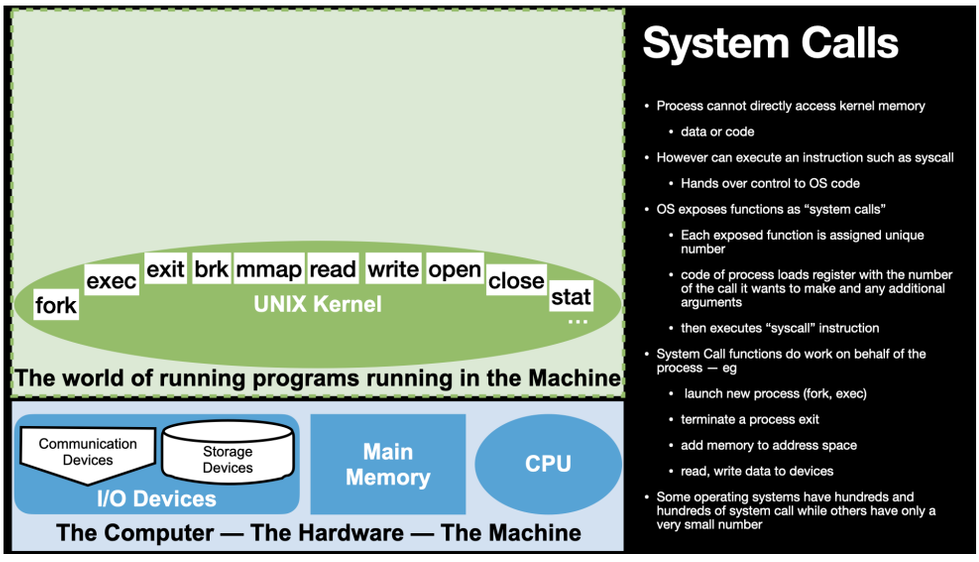
Remember this...

The "Kernel" – Unique to Every OS

1. Bootstraps the HW and has direct access to all of it
2. Bottom layer that enables other programs to run
3. **A unique collection of functions that programs can invoke**

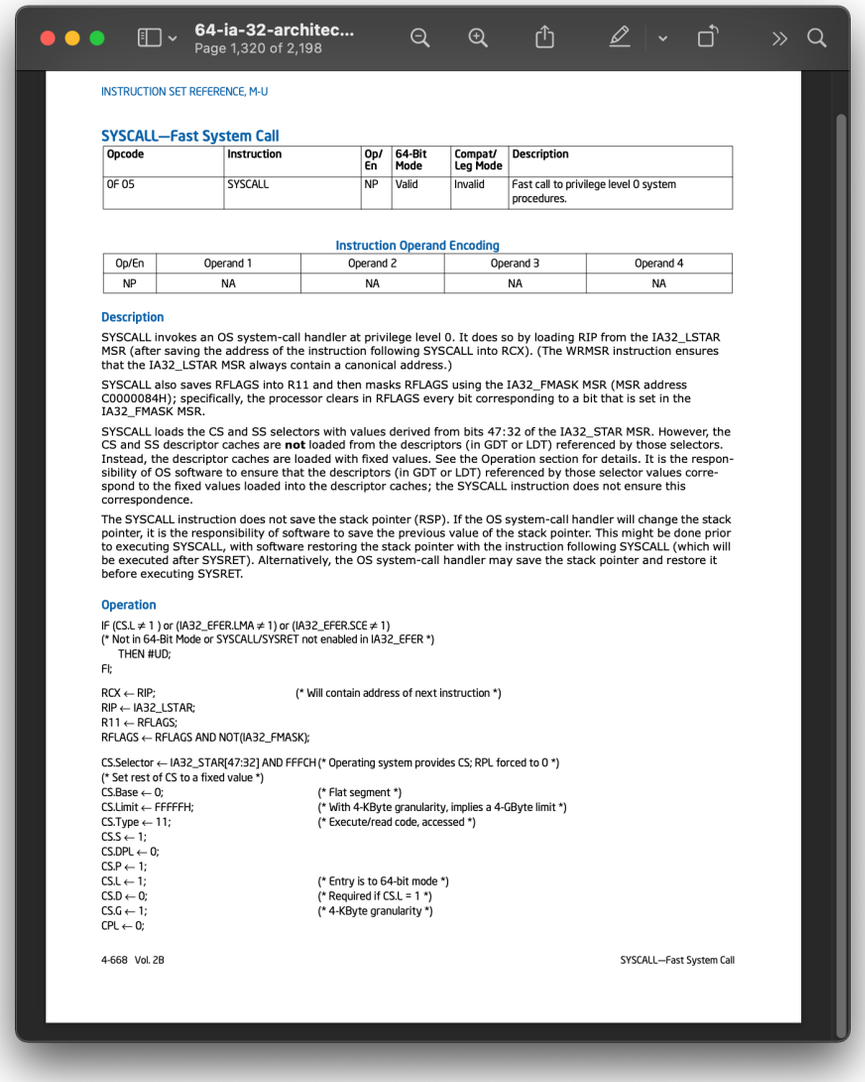
Not useful on its own only useful and accessed by running other programs.

use kernel to access



14.2.1. Intel `syscall`

On Intel the instruction is `syscall` – **NOTE "CLOBBERS" RCX and R11**



14.2.2. The OS System Calls

Each OS Kernel provides a set of calls that a process can invoke using the `syscall` instruction on an Intel based computer

The Linux Kernel supports a very large number of system calls each is identified by a unique number that must be placed in RAX prior to executing the `syscall` instruction. Additional arguments are passed in by setting other registers.

With each version of the Kernel the table of calls changes. Here is one site that provides a list

14.2.3. LINUX SYSTEM CALLS

- reading some man pages `man syscalls` and `man syscall` we find that
  - to exit we must place `60` in `rax`

exit status

- and that the value we want to set as our exit status code in `rdi`

- each system call will accept arguments in various registers
- NOTE: **On INTEL 64bit processors `rcx` will be overwritten by the syscall instruction**
- On INTEL 64bit processors Linux system calls will return a value back in `rax` and possibly `rdx`
  - `rax` and possibly `rdx` will be overwritten by a system call

```
        .intel_syntax noprefix

        .text
        .global _start
_start:
        mov     rax, 60    # Linux exit system call number is 60
        mov     rdi, 0     # rdi is return value 0 success

        syscall
```

14.3. I/O

Filippo.io

Searchable Linux Syscall Table for x86 and x86_64

There are some tables like this around, but they are usually cool auto-generated hacks and that has the downfall of not distinguishing what of the different implementations is the correct one, etc.

So, here is a lovingly hand-crafted Linux Syscall table for the x86[-64] architecture, with arguments, calling convention and links to the code included. Also, fuzzy search!

64-bit

32-bit

(Coming soon)

Instruction: `syscall`

Return value found in: `%rax`

Syscalls are implemented in functions named as in the *Entry point* column, or with the `DEFINE_SYSCALLx(%name%` macro.

Relevant man pages: `syscall(2)`, `syscalls(2)`

Double click on a row to reveal the arguments list. Search using the fuzzy filter box.

Filter:

%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c
3	close	sys_close	fs/open.c
4	stat	sys_newstat	fs/stat.c
5	fstat	sys_newfstat	fs/stat.c
6	lstat	sys_newlstat	fs/stat.c
7	poll	sys_poll	fs/select.c
8	lseek	sys_lseek	fs/read_write.c
9	mmap	sys_mmap	arch/x86/kernel/sys_x86_64.c
10	mprotect	sys_mprotect	mm/mprotect.c
11	munmap	sys_munmap	mm/mmap.c
12	brk	sys_brk	mm/mmap.c
13	rt_sigaction	sys_rt_sigaction	kernel/signal.c
14	rt_sigprocmask	sys_rt_sigprocmask	kernel/signal.c
15	rt_sigreturn	stub_rt_sigreturn	arch/x86/kernel/signal.c
16	ioctl	sys_ioctl	fs/ioctl.c
17	pread64	sys_pread64	fs/read_write.c
18	pwrite64	sys_pwrite64	fs/read_write.c
19	readv	sys_readv	fs/read_write.c
20	writev	sys_writev	fs/read_write.c
21	access	sys_access	fs/open.c
22	pipe	sys_pipe	fs/pipe.c
23	select	sys_select	fs/select.c
24	sched_yield	sys_sched_yield	kernel/sched/core.c
25	mremap	sys_mremap	mm/mmap.c
26	msync	sys_msync	mm/msync.c
27	mincore	sys_mincore	mm/mincore.c
28	madvise	sys_madvise	mm/madvise.c
29	shmget	sys_shmget	ipc/shm.c

THE STORY SO FAR

... the inner world of a program – Main Memory

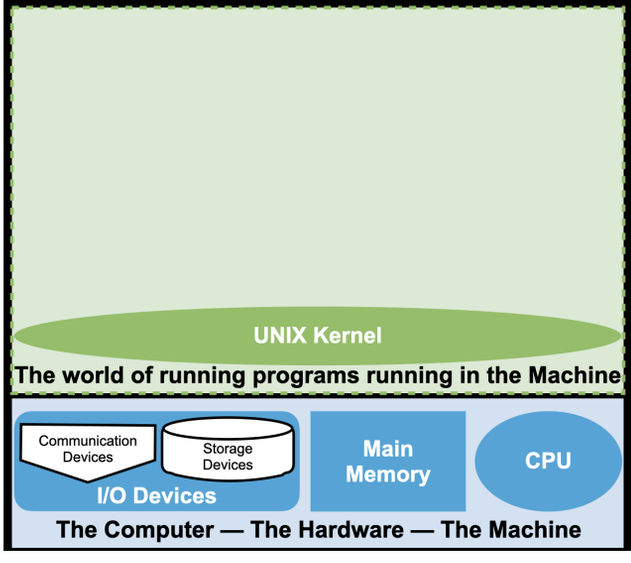
- Programs run and operate in the address space of a process and CPU registers
- Assembler and linker let us write binaries that directly access and use the main memory of the Process
- The OS lets us load and run our binaries
- But how do we get data and put data to the outside world????
- Remember stdin, stdout, and stderr?

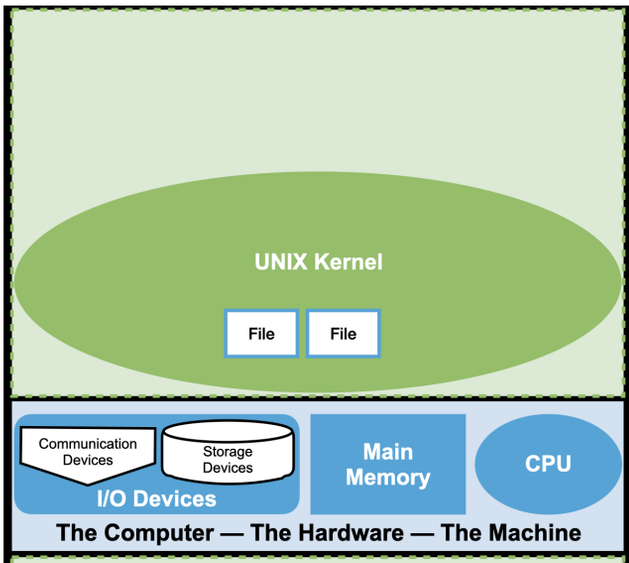
Without IO our programs are trapped in a box of main memory

14.3.1. Review from Lecture 3

Files, Streams and File Descriptors/Handles

Now we can talk about I/O that we have a working model of Processes

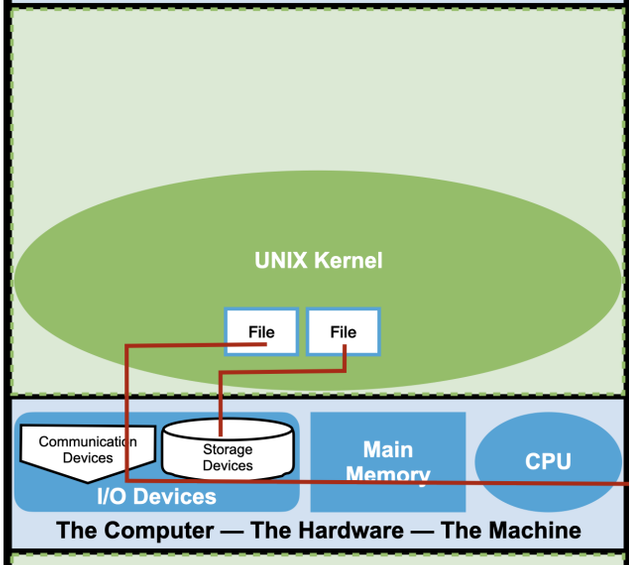




Closer look at Files

- Files as kernel objects
  - data structures in the kernel with operations
    - read file data
    - write file data

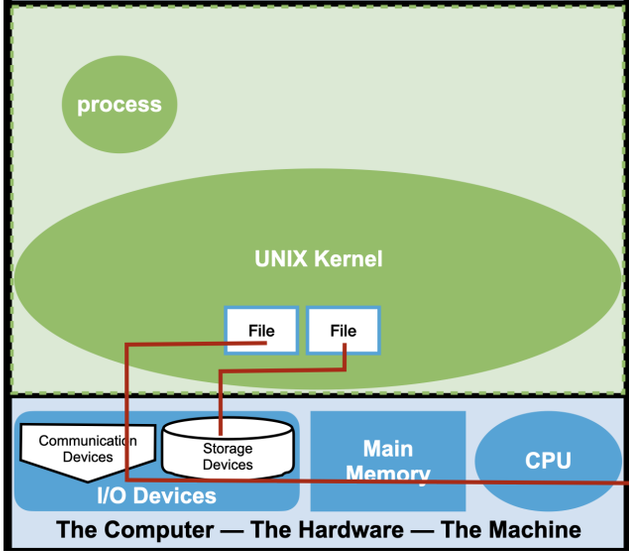
8



Closer look at Files

- Files as kernel objects
  - data structures in the kernel with operations
    - read file data
    - write file data
  - UNIX “Everything is a file”
    - data of file maybe “backed” by a device or logical device
      - eg. terminal, disk file, network connection,…

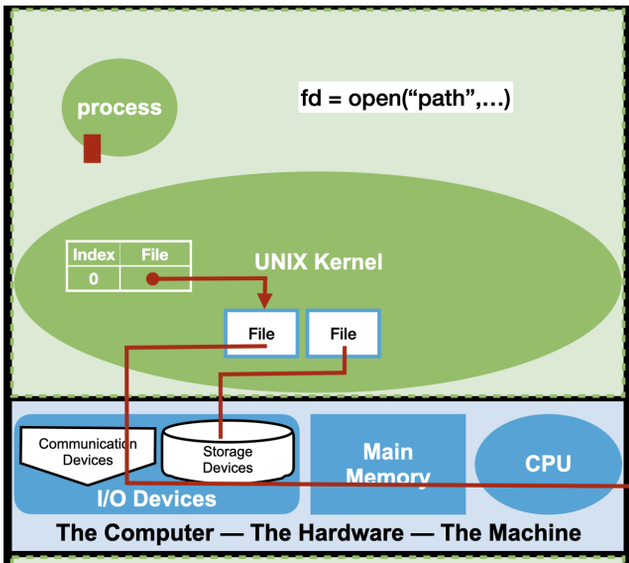
9



Processes and Streams

- UNIX allows Process to attach a “stream” to a kernel File object

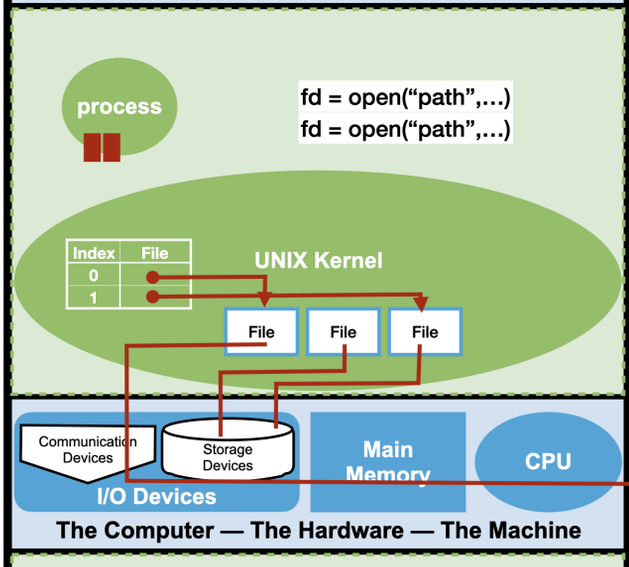
10



Processes and Streams

- UNIX allows Process to attach a "stream" to a kernel File object
  - Kernel maintains a File Descriptor Table for each process
  - A stream is an index into the table
  - Open allows process to point an index to a particular file object (and direction - open for read, write or both)

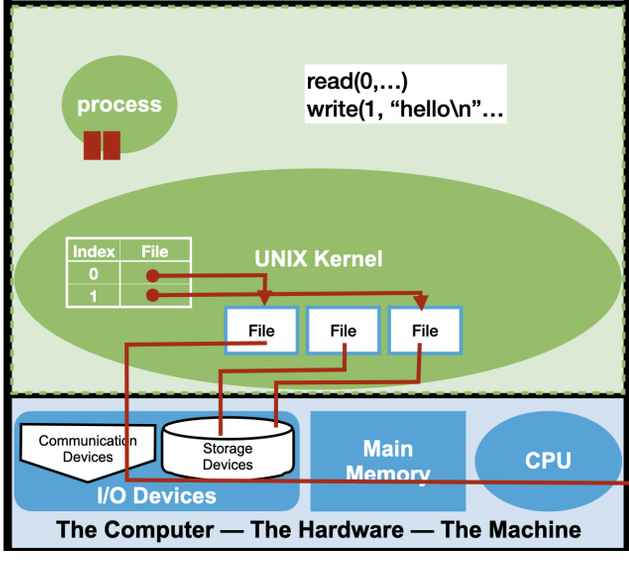
11



Processes and Streams

- UNIX allows Process to attach a "stream" to a kernel File object
  - Kernel maintains a File Descriptor Table for each process
  - A stream is an index into the table
  - Open allows process to point an index to a particular file object (and direction - open for read, write or both)
    - Open parameters can be used to also create new file objects

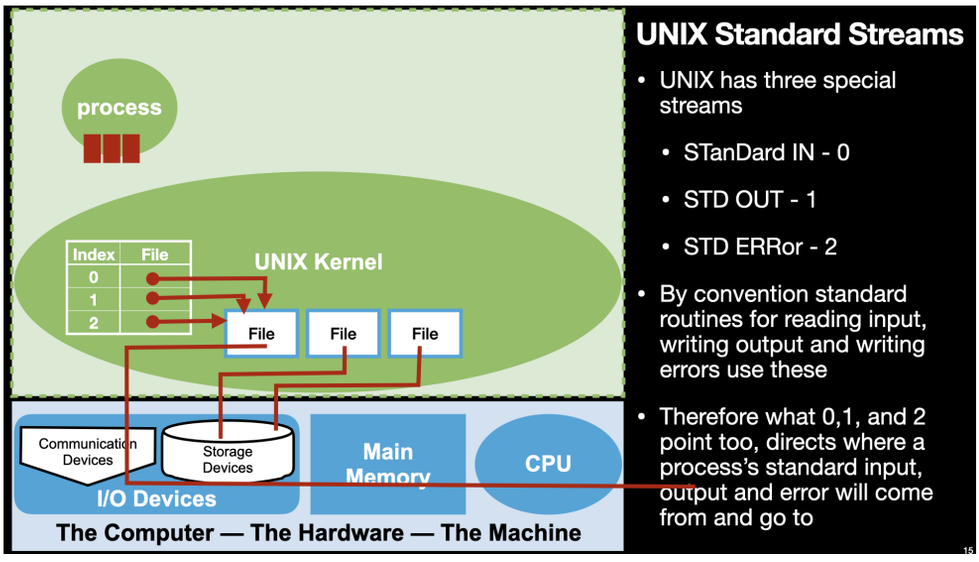
12



Processes and Streams

- Process can read and write bytes to and from a stream
- close can be used to disconnect a file from a process's file descriptor table

13



14.3.2. Finally "Hello World in Assembly"

CODE: asm - hello.s

```
        .intel_syntax noprefix
        .section .rodata        # readonly data
str:
        .string "Hello World\n"  # the string

        .section .text
        .global _start
_start:
        mov rax, 1          # write syscall number 1
        mov rdi, 1          # fd = rdi = stdout = 1
        mov rsi, OFFSET str # address of data to send
        mov rdx, 12         # length of data
        syscall

        mov rax, 60         # exit syscall number 60
        mov rdi, 3          # rdi = exit status code = 3
        syscall
```

Read
Std input

rax: 1 → write
rdi: 1
rsi: str
rdx: 12

→ end

14.3.2.1. At long last we have a program that we don't need the debugger for!

14.3.3. How about input – remember standard input

```
display(Markdown(FileCodeBox(
    file="../src/read.s",
    lang="gas",
    title="<b>CODE: asm - read.s",
    h="100%",
    w="107em"
)))
```

CODE: asm - read.s

Std input Read

```
        .intel_syntax noprefix

        .section .data
buffer:
        .byte 0x0           # space to read one byte
                            # from standard input
        .section .text
        .global _start
_start:
        mov rax, 0          # read syscall number 0
        mov rdi, 0          # rdi = fd = stdin = 0
        # rsi address of memory to place data read
        mov rsi, OFFSET buffer
        mov rdx, 1          # maximum bytes to read
        syscall
        # will "block" until data is received on
        # standard input -- this means we will
        # hang here until some presses enter

        # now exit
        mov rax, 60
        mov rdi, 0
        syscall
```

-a= read.0.1st (while creating object file create list)

14.3.4. Blocking: System calls can "block" your program

By default the read system call will not return until some data is read or an error occurs. So when we run the read example it will wait for the read to return before it goes on to exit. This waiting on a system call is called "blocking"

14.3.5. There are many more calls for doing I/O

But we can now

1. **transfer bytes from our address space to a "file" - write system call**
2. **transfer bytes from a "file" to our address space - read system call**
3. **connect (and possibly create) files to our address space for reading and writing - open system call**
4. **disconnect a file from our address space - close system call**

Filippo.io

Searchable Linux Syscall Table for x86 and x86_64

There are some tables like this around, but they are usually cool auto-generated hacks and that has the downfall of not distinguishing what of the different implementations is the correct one, etc.

So, here is a lovingly hand-crafted Linux Syscall table for the x86[-64] architecture, with arguments, calling convention and links to the code included. Also, fuzzy search!

64-bit

32-bit

(Coming soon)

Instruction: `syscall`

Return value found in: `%rax`

Syscalls are implemented in functions named as in the *Entry point* column, or with the `DEFINE_SYSCALLx(%name%` macro.

Relevant man pages: `syscall(2)`, `syscalls(2)`

Double click on a row to reveal the arguments list. Search using the fuzzy filter box.

Filter:

%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c
3	close	sys_close	fs/open.c
4	stat	sys_newstat	fs/stat.c
5	fstat	sys_newfstat	fs/stat.c
6	lstat	sys_newlstat	fs/stat.c
7	poll	sys_poll	fs/select.c
8	lseek	sys_lseek	fs/read_write.c
9	mmap	sys_mmap	arch/x86/kernel/sys_x86_64.c
10	mprotect	sys_mprotect	mm/mprotect.c
11	munmap	sys_munmap	mm/mmap.c
12	brk	sys_brk	mm/mmap.c
13	rt_sigaction	sys_rt_sigaction	kernel/signal.c
14	rt_sigprocmask	sys_rt_sigprocmask	kernel/signal.c
15	rt_sigreturn	stub_rt_sigreturn	arch/x86/kernel/signal.c
16	ioctl	sys_ioctl	fs/ioctl.c
17	pread64	sys_pread64	fs/read_write.c
18	pwrite64	sys_pwrite64	fs/read_write.c
19	readv	sys_readv	fs/read_write.c
20	writev	sys_writev	fs/read_write.c
21	access	sys_access	fs/open.c
22	pipe	sys_pipe	fs/pipe.c
23	select	sys_select	fs/select.c
24	sched_yield	sys_sched_yield	kernel/sched/core.c
25	mremap	sys_mremap	mm/mmap.c
26	msync	sys_msync	mm/msync.c
27	mincore	sys_mincore	mm/mincore.c
28	madvise	sys_madvise	mm/madvise.c
29	shmget	sys_shmget	ipc/shm.c
30	shmat	sys_shmat	ipc/shm.c

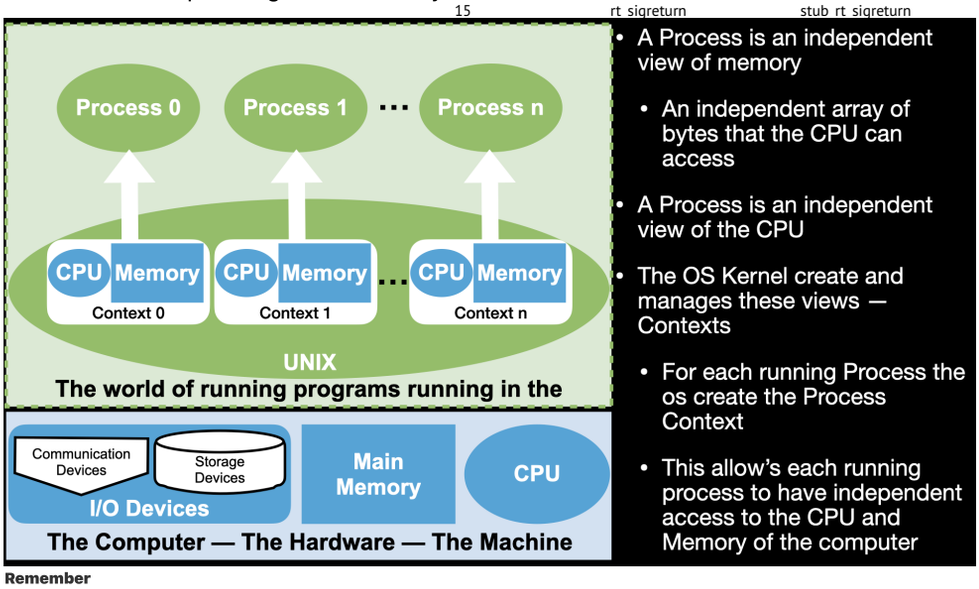
14.3.6. Exercises

Change the `usesum` program to:

- **exit properly**
- **allocate memory for the data**
- **read the data in from standard in**
- **write the binary result to standard out**
  - **use a shell command like `od -l -A10` to convert the output to ascii**
- **open the binary file of numbers and read them in**
- **can you figure out how to get the command line arguments?**
- **Write a new program that uses the random instruction to create a data file**

14.4. Process Address Spaces

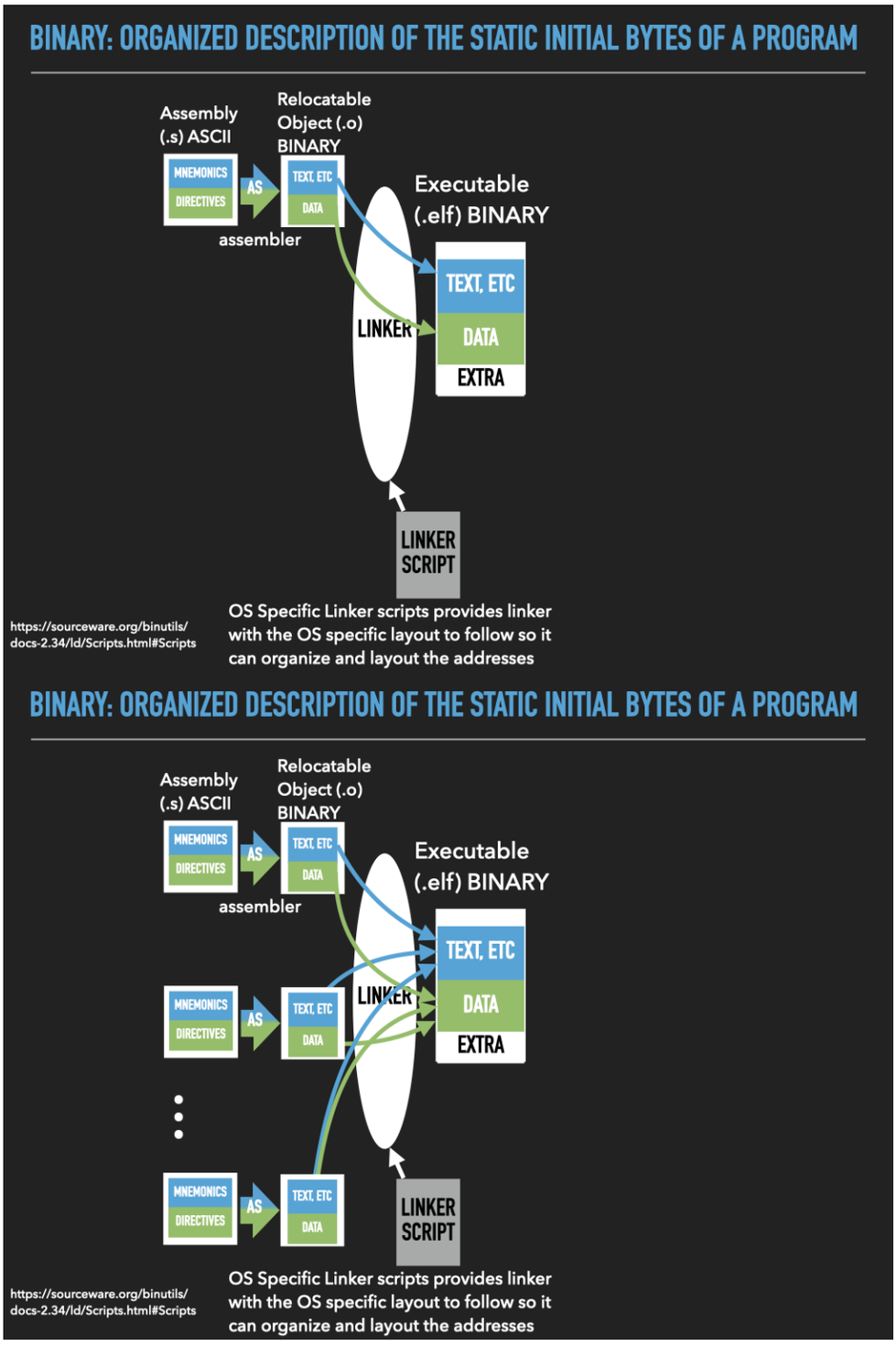
14.4.1. Address Space organizes Memory of a Process

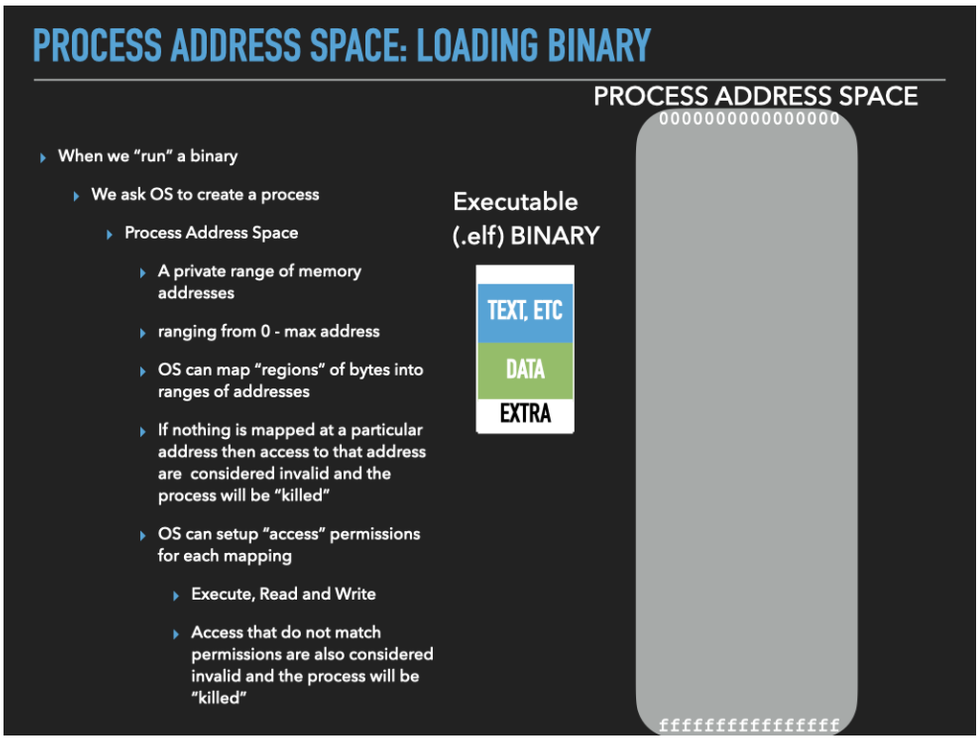


Remember

1. **The binaries we create using the assembler and linker are Executables**
2. **When we "run" our executables via the shell it creates a new process context and our binary is loaded as the initial memory "image".**
3. **The memory and register values of the process are unique to each process and change as the instructions of our binary "execute"**

14.4.2. Assembly Programming as Process Address Space Image creation



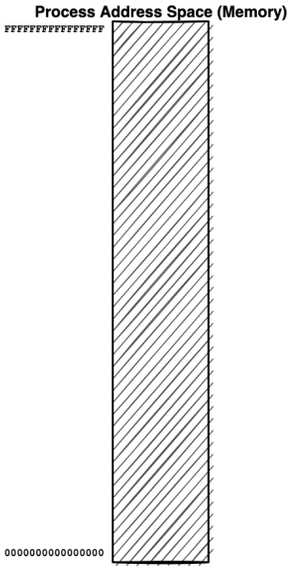


14.4.3. The Details

14.4.4. Process Address Space

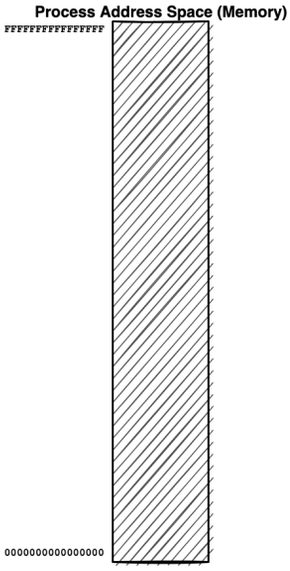
The Memory of a Process is organized as an (Virtual) Address Space

1. **Each possible location of a byte is identified by a unique address (number)**
2. **To associate a Region (range of continues addresses) with actual memory requires a "mapping"**
   - **mappings associate a Region with Memory and a possible source of values**
   - **mappings can restrict what kind of access can be made to region**
     1. **fetch for execute: x**
     2. **read: r**
     3. **write: w**



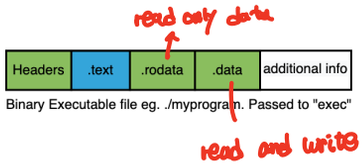
Executing a binary executable file and the Process Address Space

A Process's Address Space is like an empty container.
Regions of it must be "mapped" too memory by the OS for portions to be valid.
Accessing (fetching, reading or writing) an address with no mapping is invalid and the process will be killed. A mapping can restrict what access are allowed within a region. Eg a region maybe mapped to only allow subset of fetching, reading, or writing.

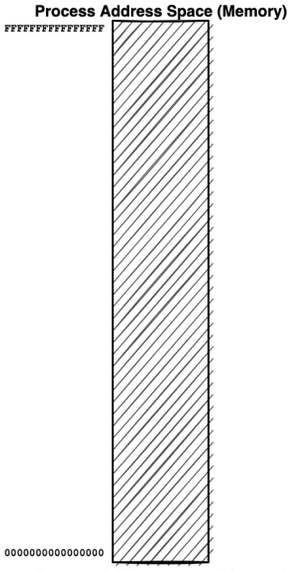


Executing a binary executable file and the Process Address Space

A Process's Address Space is like an empty container.
Regions of it must be "mapped" too memory by the OS for portions to be valid.
Accessing (fetching, reading or writing) an address with no mapping is invalid and the process will be killed. A mapping can restrict what access are allowed within a region. Eg a region maybe mapped to only allow subset of fetching, reading, or writing.



Binary Executable file eg. ./myprogram. Passed to "exec"

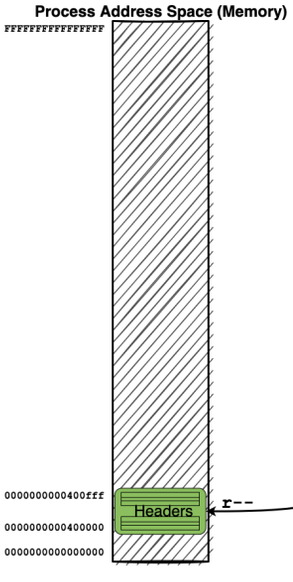


Executing a binary executable file and the Process Address Space

A Process's Address Space is like an empty container.
Regions of it must be "mapped" too memory by the OS for portions to be valid.
Accessing (fetching, reading or writing) an address with no mapping is invalid and the process will be killed. A mapping can restrict what access are allowed within a region. Eg a region maybe mapped to only allow subset of fetching, reading, or writing.

OS *exec* call uses info in binary file to setup some initial mappings for Process Address Space

Binary Executable file eg. ./myprogram. Passed to "exec"

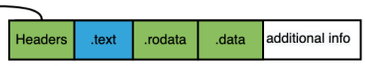


Executing a binary executable file and the Process Address Space

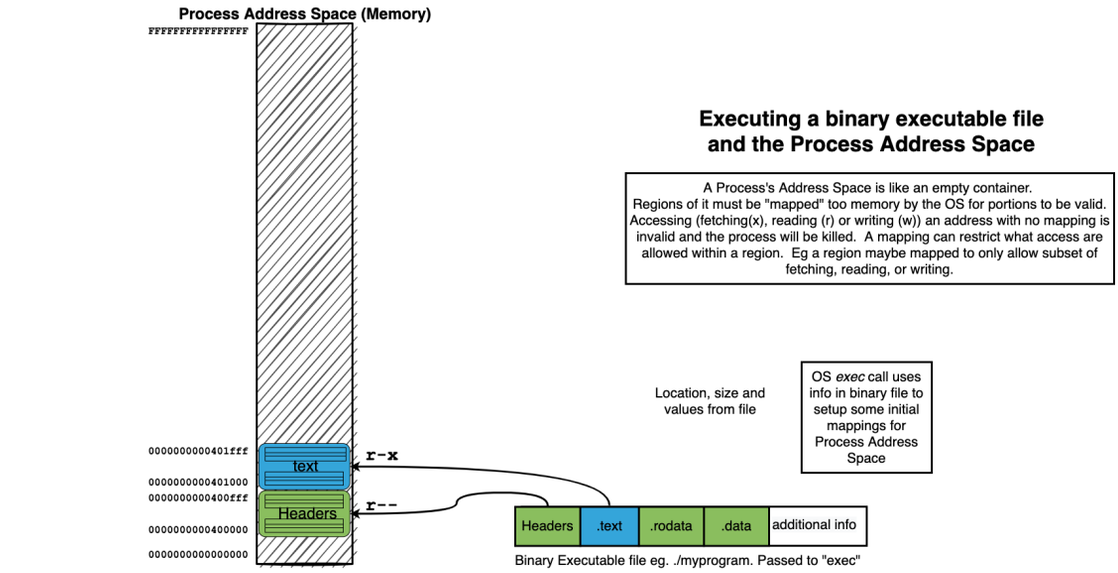
A Process's Address Space is like an empty container.
Regions of it must be "mapped" too memory by the OS for portions to be valid.
Accessing (fetching(x), reading (r) or writing (w)) an address with no mapping is invalid and the process will be killed. A mapping can restrict what access are allowed within a region. Eg a region maybe mapped to only allow subset of fetching, reading, or writing.

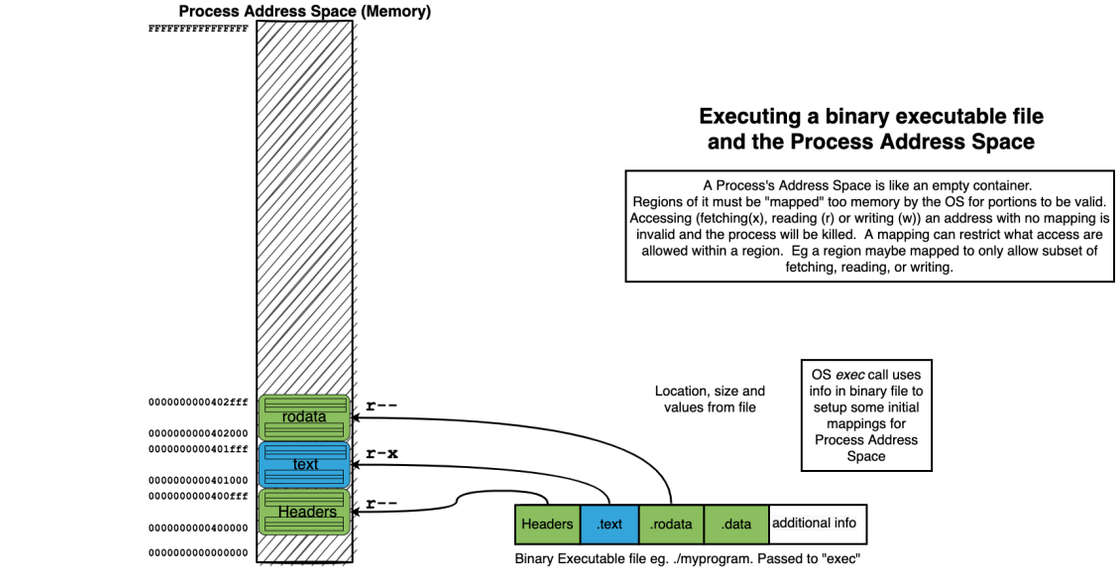
Location, size and values from file

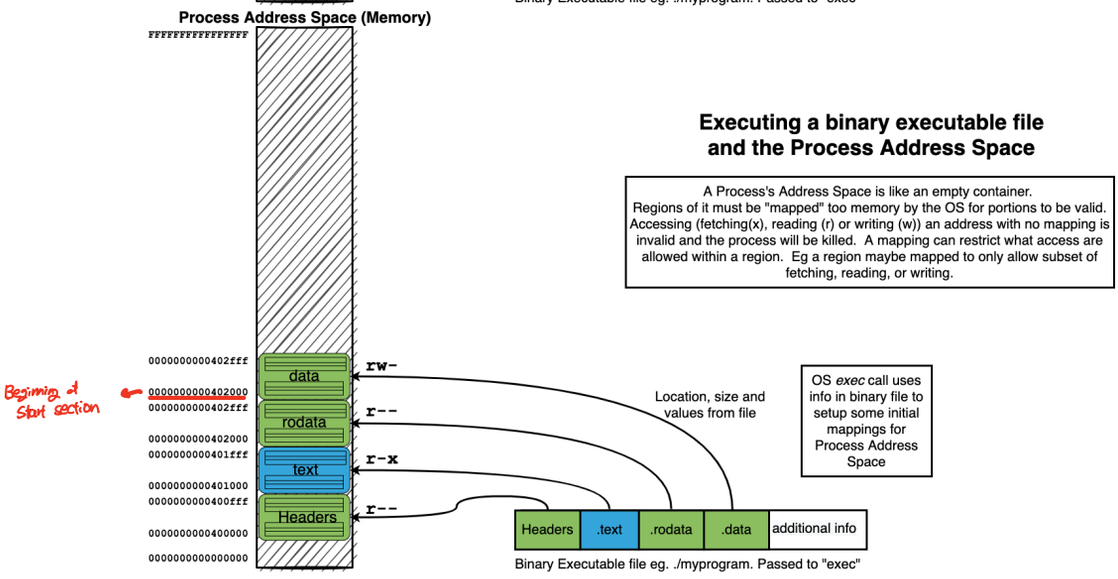
OS *exec* call uses info in binary file to setup some initial mappings for Process Address Space



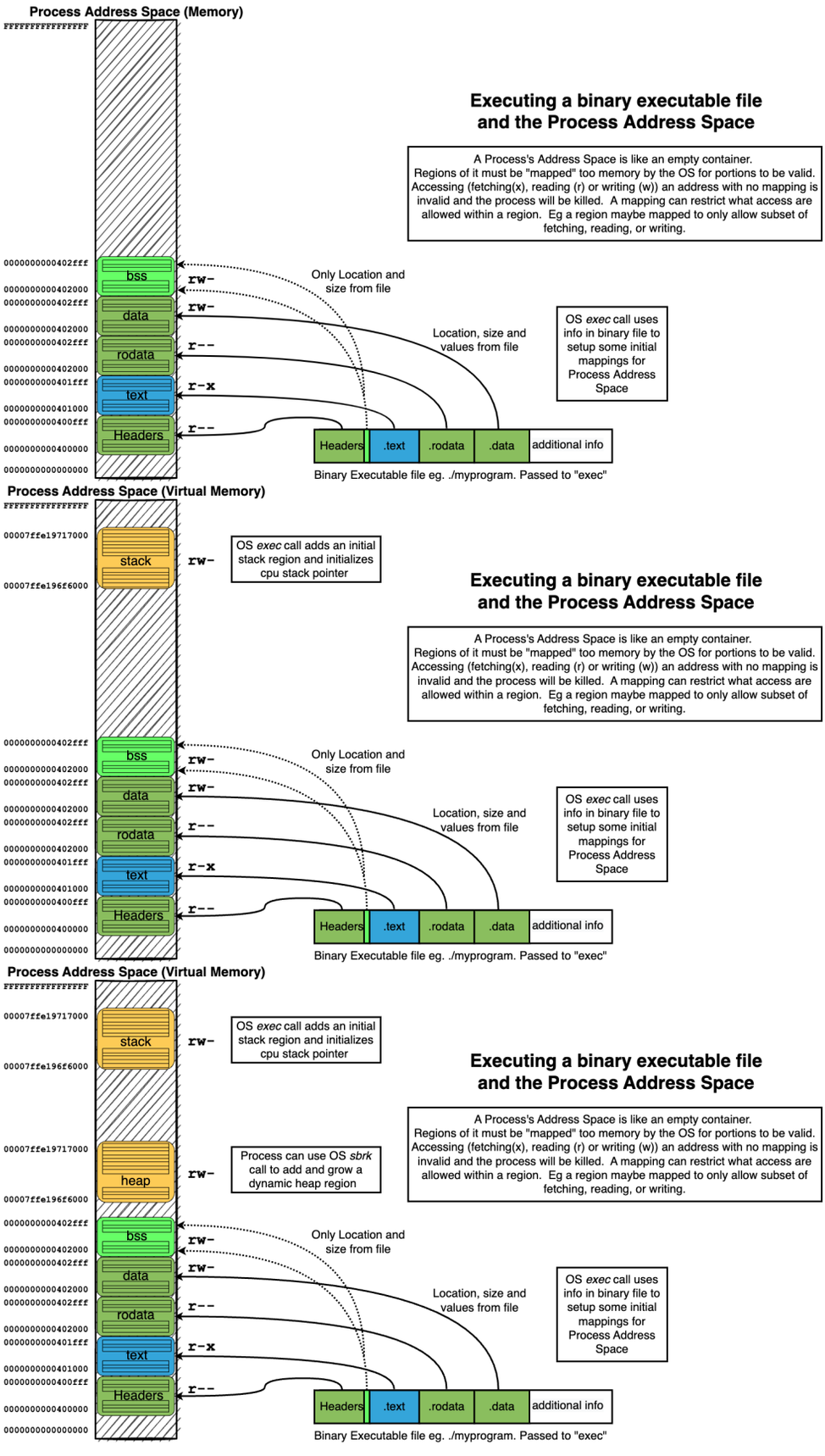
Binary Executable file eg. ./myprogram. Passed to "exec"

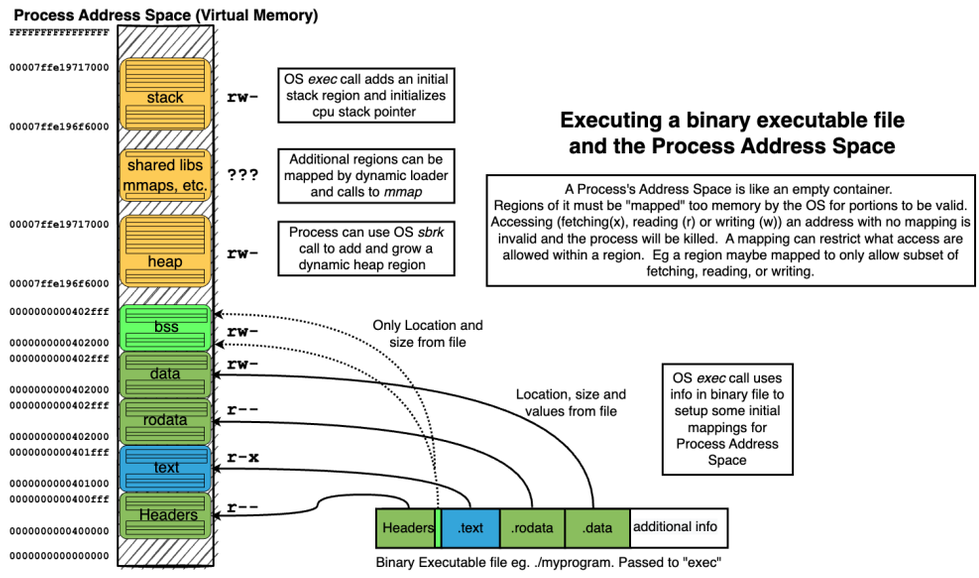




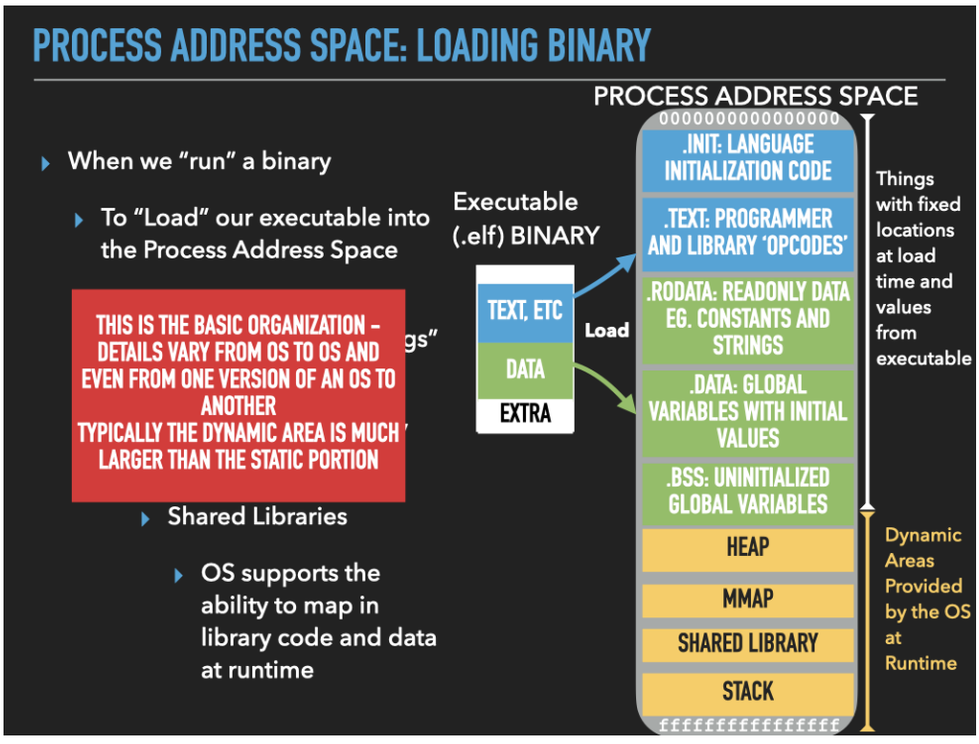


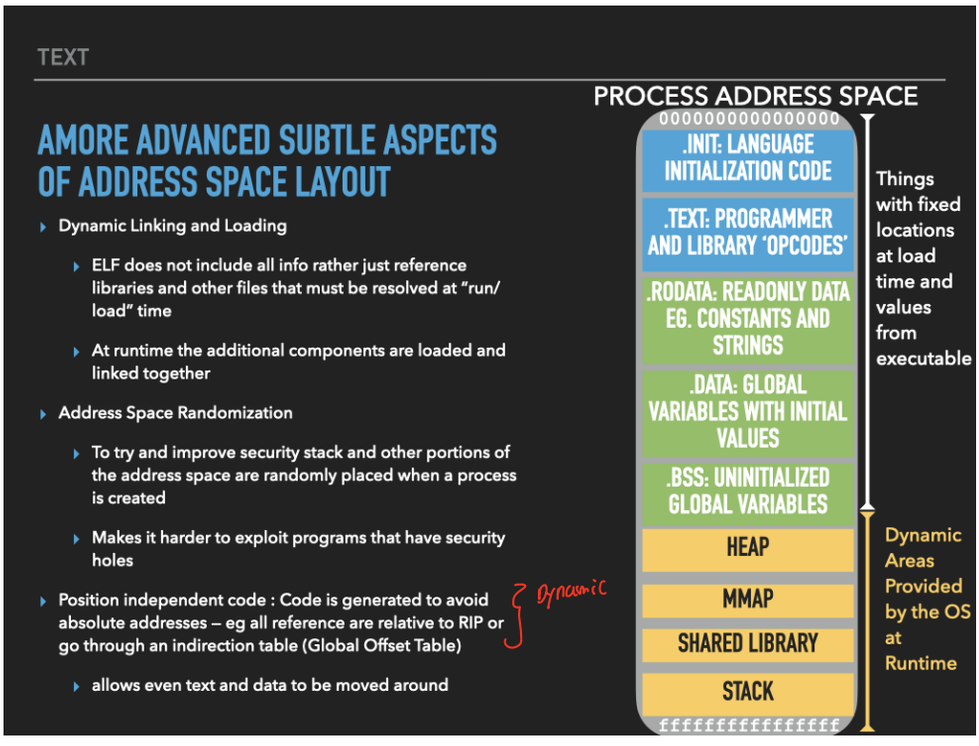
Beginning of Start section


Process Address Space (Memory)
FFFFFFFFFFFFFFFF
0000000000402fff
bss
0000000000402000
0000000000402fff
data
0000000000402000
0000000000402fff
rodata
0000000000402000
0000000000401fff
text
0000000000401000
0000000000400fff
Headers
0000000000400000
0000000000000000
rw-
rw-
r--
r-x
r--
Only Location and size from file
Location, size and values from file
OS exec call uses info in binary file to setup some initial mappings for Process Address Space
Headers
.text
.rodata
.data
additional info
Binary Executable file eg. ./myprogram. Passed to "exec"
Executing a binary executable file and the Process Address Space
A Process's Address Space is like an empty container.
Regions of it must be "mapped" too memory by the OS for portions to be valid.
Accessing (fetching(x), reading (r) or writing (w)) an address with no mapping is invalid and the process will be killed. A mapping can restrict what access are allowed within a region. Eg a region maybe mapped to only allow subset of fetching, reading, or writing.
Process Address Space (Virtual Memory)
FFFFFFFFFFFFFFFF
00007ffe19717000
stack
00007ffe196f6000
rw-
OS exec call adds an initial stack region and initializes cpu stack pointer
Executing a binary executable file and the Process Address Space
A Process's Address Space is like an empty container.
Regions of it must be "mapped" too memory by the OS for portions to be valid.
Accessing (fetching(x), reading (r) or writing (w)) an address with no mapping is invalid and the process will be killed. A mapping can restrict what access are allowed within a region. Eg a region maybe mapped to only allow subset of fetching, reading, or writing.
0000000000402fff
bss
0000000000402000
0000000000402fff
data
0000000000402000
0000000000402fff
rodata
0000000000402000
0000000000401fff
text
0000000000401000
0000000000400fff
Headers
0000000000400000
0000000000000000
rw-
rw-
r--
r-x
r--
Only Location and size from file
Location, size and values from file
OS exec call uses info in binary file to setup some initial mappings for Process Address Space
Headers
.text
.rodata
.data
additional info
Binary Executable file eg. ./myprogram. Passed to "exec"
Process Address Space (Virtual Memory)
FFFFFFFFFFFFFFFF
00007ffe19717000
stack
00007ffe196f6000
rw-
OS exec call adds an initial stack region and initializes cpu stack pointer
Executing a binary executable file and the Process Address Space
A Process's Address Space is like an empty container.
Regions of it must be "mapped" too memory by the OS for portions to be valid.
Accessing (fetching(x), reading (r) or writing (w)) an address with no mapping is invalid and the process will be killed. A mapping can restrict what access are allowed within a region. Eg a region maybe mapped to only allow subset of fetching, reading, or writing.
00007ffe19717000
heap
00007ffe196f6000
rw-
Process can use OS sbrk call to add and grow a dynamic heap region
0000000000402fff
bss
0000000000402000
0000000000402fff
data
0000000000402000
0000000000402fff
rodata
0000000000402000
0000000000401fff
text
0000000000401000
0000000000400fff
Headers
0000000000400000
0000000000000000
rw-
rw-
r--
r-x
r--
Only Location and size from file
Location, size and values from file
OS exec call uses info in binary file to setup some initial mappings for Process Address Space
Headers
.text
.rodata
.data
additional info
Binary Executable file eg. ./myprogram. Passed to "exec"



14.5. Summary of how we “loading” a process address from a binary





14.6. Code and instructions for exploring the Process Address Space layout

In the lecture notes at this point you will find code and instructions for exploring the address space layout

1. **How we control the mappings based on the sections we add**
2. **How we can add and shrink heap memory using the `brk` system call**

To do the exploration we use :

1. **The `nm` command that print's the symbol table of a binary. As such we can use it to see what addresses the linker placed our bytes at.**
2. **A special file that the OS provides that describes the address space of a running process call the `maps` file.**

Detail are provide in the notes. Don't forget you can also use gdb to probe the various mappings you discover.
Eg. You can use the `x` command to examine the bytes at the addresses of the memory areas you find in the maps file.

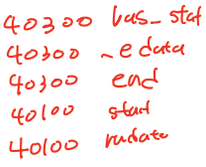
14.6.1. Exploring the Address Space of a process with Code

- **Let's explore the address space of a running program**
- **We will use 5 versions of a program the progressively adds more regions to the address space**
- **The programs will wait for input before exiting so that we can use the OS provided file for check what the process address space looks like**

14.6.1.1. Version 1: text and data

CODE: asm - exploringASlayout1.s

```
        # To use this assemble and link it:
        # as -g exploringASlayout1.s -o exploringASlayout1.o
        # ld -g exploringASlayout1.o -o exploringASlayout1
        # Then using two terminals:
        #   1. use the nm command on the binary to see what addresses the linker
        #      placed all the symbols at Eg.
        #        $ nm exploringASlayout1
        #        0000000000403000 D __bss_start
        #        0000000000403000 D _edata
        #        0000000000403000 D _end
        #        0000000000402000 d rwdata
        #        0000000000401000 T _start
        #      (if you like you can also run: "readelf -S <binary>" to
        #       see more information about the sections in the binary)
        #   2. then run the binary -- its should stop on the read
        #      Eg.
        #      $ ./exploringASlayout1
        #
        #   3. in another terminal use ps and grep to find the process started from the binary
        #      Eg.
        #       $ ps auxgww | grep exploring
        #       jovyan      1697  0.0  0.0    168     4 pts/29   S+   13:41   0:00 ./exploringASlayout1
        #       jovyan      1699  0.0  0.0   6440   720 pts/30   S+   13:42   0:00 grep exploring
        #       $
        #      We see the process id (pid) is 1697
        #   4. using the process id you can now examine the file "/proc/<pid>/maps" to
        #      see the layout of the running processes address space
        #      Eg.
        #      $ cat /proc/1697/maps
        #      00400000-00401000 r--p 00000000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      00401000-00402000 r-xp 00001000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      00402000-00403000 rw-p 00002000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      7ffd13044000-7ffd13065000 rw-p 00000000 00:00 0                          [stack]
        #      7ffd13069000-7ffd1306d000 r--p 00000000 00:00 0                          [vvar]
        #      7ffd1306d000-7ffd1306f000 r-xp 00000000 00:00 0                          [vdso]
        #      ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
        #      $
        #     What we see is that there are three mappings to the binary file, a mapping for the stack and then some
        #     extra's used by the OS.
        #     5. you can now press enter in the terminal that the binary is running in
        #        this will cause the read to finish and the binary will then continue to execute

        # CODE to explore how we control the address space mappings
        # The smallest size of an address space mapping is called the page size
        # on Linux the default is 4096 (4Kb). To make it easier to identify
        # the mappings we will fill each section we add to a full page (4Kb)
        # worth of bytes.
        # This version 1 starts with just two sections
        # One page' of "text" and one page of "data"
        #  - text should get mapped to a region that is readable and executable (r-x)
        #  - data should get mapped to a region that is readable and writable (rw-)

        .intel_syntax noprefix        # assembler syntax to use <directive>
        .section .text                # linker section <directive>

        .global _start                # linker symbol type <directive>
_start:
        # use a read system call to stop the program until user presses enter
        # so that we can examine things before and after the code runs
        mov rax, 0                  # read syscall = 0 -- read(fd, buf, len)
        mov rdi, 0                  # fd = rdi = 0 = stdin
        mov rsi, OFFSET rwdata      # buf = rsi = rwdata memory
        mov rdx, 1                  # len = rdx = 1 wait for one byte "enter"
        syscall

        mov rax, 60                 # exit system call
        mov rdi, 0                  # exit status 0
        syscall

        # fill the text section up with zeros so that it is one page in size.
        .org 4095                   # force the next byte to be at the end of the page (4095
        .byte 0x00                  # put a byte of zero here

        ##### DATA SECTION : read and writable memory s
        .section .data
        .global rwdata
rwdata: .fill 4096,1,0xff


        # the following tells the linux kernel to set permissions the way we expect
        # rather than the defaults which map all sections as executable
        # When using a compiler it will add this for us
        .section        .note.GNU-stack,"",@progbits
```



Use the nm command to see where the linker placed things:

- **Now in one terminal run the binary – start a process from it**

```
./exploringASlayout1
```

- **It will hang on the read so that we have time to explore what the process's address space looks like.**
- **In another terminal use ps and grep to find the process id of the new process**

```
ps auxgww | grep exploring
```

Now we can use the process id to find the special file that the OS provides for us to examine the address space of a process.

```
cat /proc/<pid>/maps
```

1570

You can also use gdb to to explore the regions of memory that the maps file tells us are present. You can also compare what is in memory to what you find in the file.

When you are done you can press return in the terminal that the program in running. This will send a newline to the standard input of the program. This will cause the read to return and the program will go on to exit.

You can find lots of information ab out the OS provided `proc` directory in the manual `man proc`

In the manual page there is a section on the `maps` file specifically that tells us the details of what information it presents regarding the mappings for a running process.

14.6.1.2. Version 2: adding rodata

Read-only data section `.rodata`

```
    ##### DATA SECTION : read and writable memory s
      .section .data
      .global rwdata
rwdata: .fill 4096,1,0xff          # 4096 bytes each initialized with a value of 0xff
```

What would happen if we put all our data in the `.rodata` section?... try it and find out

Here is the code

CODE: asm - exploringASlayout2.s

```
        # To use this assemble and link it:
        # as -g exploringASlayout1.s -o exploringASlayout1.o
        # ld -g exploringASlayout1.o -o exploringASlayout1
        # Then using two terminals:
        #   1. use the nm command on the binary to see what addresses the linker
        #      placed all the symbols at Eg.
        #        $ nm exploringASlayout1
        #        0000000000403000 D __bss_start
        #        0000000000403000 D _edata
        #        0000000000403000 D _end
        #        0000000000402000 d rwdata
        #        0000000000401000 T _start
        #      (if you like you can also run: "readelf -S <binary>" to
        #       see more information about the sections in the binary)
        #   2. then run the binary -- its should stop on the read
        #      Eg.
        #      $ ./exploringASlayout1
        #
        #   3. in another terminal use ps and grep to find the process started from the binary
        #      Eg.
        #       $ ps auxgww | grep exploring
        #       jovyan      1697  0.0  0.0    168     4 pts/29   S+   13:41   0:00 ./exploringASlayout1
        #       jovyan      1699  0.0  0.0   6440   720 pts/30   S+   13:42   0:00 grep exploring
        #       $
        #      We see the process id (pid) is 1697
        #   4. using the process id you can now examine the file "/proc/<pid>/maps" to
        #      see the layout of the running processes address space
        #      Eg.
        #      $ cat /proc/1697/maps
        #      00400000-00401000 r--p 00000000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      00401000-00402000 r-xp 00001000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      00402000-00403000 rw-p 00002000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      7ffd13044000-7ffd13065000 rw-p 00000000 00:00 0                          [stack]
        #      7ffd13069000-7ffd1306d000 r--p 00000000 00:00 0                          [vvar]
        #      7ffd1306d000-7ffd1306f000 r-xp 00000000 00:00 0                          [vdso]
        #      ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
        #      $
        #     What we see is that there are three mappings to the binary file, a mapping for the stack and then some
        #     extra's used by the OS.
        #     5. you can now press enter in the terminal that the binary is running in
        #        this will cause the read to finish and the binary will then continue to execute

        # CODE to explore how we control the address space mappings
        # The smallest size of an address space mapping is called the page size
        # on Linux the default is 4096 (4Kb). To make it easier to identify
        # the mappings we will fill each section we add to a full page (4Kb)
        # worth of bytes.
        # This version 2 adds read only data section
        # One page' of "text", one page of "data" and one pager of "readonly data"
        #  - text should get mapped to a region that is readable and executable (r-x)
        #  - data should get mapped to a region that is readable and writable (rw-)
        #  - rodata should get mapped to a region that is readable only       (r--)

        .intel_syntax noprefix        # assembler syntax to use <directive>
        .section .text                # linker section <directive>

        .global _start                # linker symbol type <directive>
_start:
        # use a read system call to stop the program until user presses enter
        # so that we can examine things before and after the code runs
        mov rax, 0                  # read syscall = 0 -- read(fd, buf, len)
        mov rdi, 0                  # fd = rdi = 0 = stdin
        mov rsi, OFFSET rwdata      # buf = rsi = rwdata memory
        mov rdx, 1                  # len = rdx = 1 wait for one byte "enter"
        syscall

        mov rax, 60                 # exit system call
        mov rdi, 0                  # exit status 0
        syscall

        # fill the text section up with zeros so that it is one page in size.
        .org 4095                   # force the next byte to be at the end of the page (4095
        .byte 0x00                  # put a byte of zero here

        ##### DATA SECTION : read and writable memory s
        .section .data
        .global rwdata
rwdata: .fill 4096,1,0xff           # 4096 bytes each initialized with a value of 0xff

        ##### READ ONLY DATA SECTION : read only memmory
        .section .rodata
        .global rodata
rodata:
        .fill 4096,1,'a             # 4096 bytes each initialized with a value of ASCII lower case a

        # the following tells the linux kernel to set permissions the way we expect
        # rather than the defaults which map all sections as executable
        # When using a compiler it will add this for us
        .section        .note.GNU-stack,"",@progbits
```

read only data

Repeat the steps from version 1 to see what has changed.

14.6.1.3. Version 3: adding bss

The bss section is used to create a mapping for all the data that we need memory for but don't need to specify an initial value that is not zero.

Read-Write memory `.bss`

```
    ##### BSS DATA SECTION: read and write automaitically added and filled with zero values
    .section .bss
    .global zerodata
zerodata:   .fill 4096,1
    # The above could be replaced with the single line: .comm zerodata, 4096, 1
```

Note the `.comm` directive is a short cut that switch the section to .bss adds the symbol and makes it global

Here is the code

CODE: asm - exploringASlayout3.s

```
        # To use this assemble and link it:
        # as -g exploringASlayout1.s -o exploringASlayout1.o
        # ld -g exploringASlayout1.o -o exploringASlayout1
        # Then using two terminals:
        #   1. use the nm command on the binary to see what addresses the linker
        #      placed all the symbols at Eg.
        #        $ nm exploringASlayout1
        #        0000000000403000 D __bss_start
        #        0000000000403000 D _edata
        #        0000000000403000 D _end
        #        0000000000402000 d rwdata
        #        0000000000401000 T _start
        #      (if you like you can also run: "readelf -S <binary>" to
        #       see more information about the sections in the binary)
        #   2. then run the binary -- its should stop on the read
        #      Eg.
        #      $ ./exploringASlayout1
        #
        #   3. in another terminal use ps and grep to find the process started from the binary
        #      Eg.
        #       $ ps auxgww | grep exploring
        #       jovyan      1697  0.0  0.0    168     4 pts/29   S+   13:41   0:00 ./exploringASlayout1
        #       jovyan      1699  0.0  0.0   6440   720 pts/30   S+   13:42   0:00 grep exploring
        #       $
        #      We see the process id (pid) is 1697
        #   4. using the process id you can now examine the file "/proc/<pid>/maps" to
        #      see the layout of the running processes address space
        #      Eg.
        #      $ cat /proc/1697/maps
        #      00400000-00401000 r--p 00000000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      00401000-00402000 r-xp 00001000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      00402000-00403000 rw-p 00002000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      7ffd13044000-7ffd13065000 rw-p 00000000 00:00 0                          [stack]
        #      7ffd13069000-7ffd1306d000 r--p 00000000 00:00 0                          [vvar]
        #      7ffd1306d000-7ffd1306f000 r-xp 00000000 00:00 0                          [vdso]
        #      ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
        #      $
        #     What we see is that there are three mappings to the binary file, a mapping for the stack and then some
        #     extra's used by the OS.
        #     5. you can now press enter in the terminal that the binary is running in
        #        this will cause the read to finish and the binary will then continue to execute

        # CODE to explore how we control the address space mappings
        # The smallest size of an address space mapping is called the page size
        # on Linux the default is 4096 (4Kb). To make it easier to identify
        # the mappings we will fill each section we add to a full page (4Kb)
        # worth of bytes.
        # This version 3 adds bss section
        # One page' of "text", one page of "data" and one pager of "readonly data"
        #  - text should get mapped to a region that is readable and executable (r-x)
        #  - data should get mapped to a region that is readable and writable (rw-)
        #  - rodata should get mapped to a region that is readable only       (r--)
        #  - bss    should get mapped to a region that is readable and writable (rw-)

        .intel_syntax noprefix        # assembler syntax to use <directive>
        .section .text                # linker section <directive>

        .global _start                # linker symbol type <directive>
_start:
        # use a read system call to stop the program until user presses enter
        # so that we can examine things before and after the code runs
        mov rax, 0                  # read syscall = 0 -- read(fd, buf, len)
        mov rdi, 0                  # fd = rdi = 0 = stdin
        mov rsi, OFFSET rwdata      # buf = rsi = rwdata memory
        mov rdx, 1                  # len = rdx = 1 wait for one byte "enter"
        syscall

        mov rax, 60                 # exit system call
        mov rdi, 0                  # exit status 0
        syscall

        # fill the text section up with zeros so that it is one page in size.
        .org 4095                   # force the next byte to be at the end of the page (4095
        .byte 0x00                  # put a byte of zero here

        ##### DATA SECTION : read and writable memory s
        .section .data
        .global rwdata
rwdata: .fill 4096,1,0xff           # 4096 bytes each initialized with a value of 0xff

        ##### READ ONLY DATA SECTION : read only data
        .section .rodata
        .global rodata
rodata:
        .fill 4096,1,'a             # 4096 bytes each initialized with a value of ASCII lower case a

        ##### BSS DATA SECTION: read and write automaitically added and filled with zero values
        .section .bss
        .global zerodata
zerodata:       .fill 4096,1
        # The above could be replaced with the single line: .comm zerodata, 4096, 1

        # the following tells the linux kernel to set permissions the way we expect
        # rather than the defaults which map all sections as executable
        # When using a compiler it will add this for us
        .section        .note.GNU-stack,"",@progbits
```

Repeat the steps from version 1 and see what what has changed compared to versions 1 and 2

14.6.1.4. Version 4: And and Grow our Heap

Now let's explore how while our program is running we can get more memory added to our process.

The `brk` system call - move "break pointer"

- **initially the data segment of your process will end at some location based on what data sections you defined**
  - **this location is called the program break pointer**
  - **the location where valid data memory ends and the un-allocated virtual address space**
- **with `brk` you can set the end to a value bigger than it starting location**
- **you can also shrink the break pointer back to remove memory**

This area is called the HEAP

1. **call `brk` with argument of 0 and Linux kernel returns current break address**

2. call `brk` **"with a reasonable address" and the kernel will allocated or de-allocated memory for us**

- **in reality once we move to C we never directly call "brk" rather we will use a library routines called : `malloc` and `free` which will call `brk` for us.**

As usual you can find more info in the manual

You can find the system call number and what the arguments are here:

Lets add this code

What does it do?

```
    # Now add a Heap
    # get current break pointer
    xor rdi, rdi               # pass 0 to brk (invalid request)
    mov rax, 12                # brk syscall number 12
    syscall                    # call brk with 0

    mov r15, rax               # keep a copy of where our new memory starts
                               # so we can use it later

    # now add some memory by requesting to increase the break
    # address by 32 bytes
    mov     rdi, rax            # mov current break into rdi
    add     rdi, 4096           # ask for 4096 bytes rdi=rdi+4096
    mov     rax, 12             # brk sycall 12
    syscall
```

CODE: asm - exploringASlayout4.s

Filippo.io

Searchable Linux Syscall Table for x86 and x86_64

There are some tables like this around, but they are usually cool auto-generated hacks and that has the downfall of not distinguishing what of the different implementations is the correct one, etc.

So, here is a lovingly hand-crafted Linux Syscall table for the x86[-64] architecture, with arguments, calling convention and links to the code included. Also, fuzzy search!

64-bit

32-bit

(Coming soon)

Instruction: `syscall`

Return value found in: `%rax`

Syscalls are implemented in functions named as in the *Entry point* column, or with the `DEFINE_SYSCALLx(%name%` macro.

Relevant man pages: `syscall(2)`, `syscalls(2)`

Double click on a row to reveal the arguments list. Search using the fuzzy filter box.

Filter:

%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
			fs/open.c
			fs/open.c
			fs/stat.c
			fs/stat.c
			fs/stat.c
			fs/select.c
			fs/read_write.c
			arch/x86/kernel/sys_x86_64.c
			mm/mprotect.c
			mm/mmap.c
12	brk	sys_brk	mm/mmap.c
13	rt_sigaction	sys_rt_sigaction	kernel/signal.c
14	rt_sigprocmask	sys_rt_sigprocmask	kernel/signal.c
15	rt_sigreturn	stub_rt_sigreturn	arch/x86/kernel/signal.c
16	ioctl	sys_ioctl	fs/ioctl.c
17	pread64	sys_pread64	fs/read_write.c
18	pwrite64	sys_pwrite64	fs/read_write.c
19	readv	sys_readv	fs/read_write.c
20	writev	sys_writev	fs/read_write.c
21	access	sys_access	fs/open.c
22	pipe	sys_pipe	fs/pipe.c
23	select	sys_select	fs/select.c
24	sched_yield	sys_sched_yield	kernel/sched/core.c
25	mremap	sys_mremap	mm/mmap.c
26	msync	sys_msync	mm/msync.c
27	mincore	sys_mincore	mm/mincore.c
28	madvise	sys_madvise	mm/madvise.c
29	shmget	sys_shmget	ipc/shm.c
30	shmat	sys_shmat	ipc/shm.c

```
        # To use this assemble and link it:
        # as -g exploringASlayout1.s -o exploringASlayout1.o
        # ld -g exploringASlayout1.o -o exploringASlayout1
        # Then using two terminals:
        #   1. use the nm command on the binary to see what addresses the linker
        #      placed all the symbols at Eg.
        #        $ nm exploringASlayout1
        #        0000000000403000 D __bss_start
        #        0000000000403000 D _edata
        #        0000000000403000 D _end
        #        0000000000402000 d rwdata
        #        0000000000401000 T _start
        #      (if you like you can also run: "readelf -S <binary>" to
        #       see more information about the sections in the binary)
        #   2. then run the binary -- its should stop on the read
        #      Eg.
        #      $ ./exploringASlayout1
        #
        #   3. in another terminal use ps and grep to find the process started from the binary
        #      Eg.
        #       $ ps auxgww | grep exploring
        #       jovyan      1697  0.0  0.0    168     4 pts/29   S+   13:41   0:00 ./exploringASlayout1
        #       jovyan      1699  0.0  0.0   6440   720 pts/30   S+   13:42   0:00 grep exploring
        #       $
        #      We see the process id (pid) is 1697
        #   4. using the process id you can now examine the file "/proc/<pid>/maps" to
        #      see the layout of the running processes address space
        #      Eg.
        #      $ cat /proc/1697/maps
        #      00400000-00401000 r--p 00000000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      00401000-00402000 r-xp 00001000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      00402000-00403000 rw-p 00002000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      7ffd13044000-7ffd13065000 rw-p 00000000 00:00 0                          [stack]
        #      7ffd13069000-7ffd1306d000 r--p 00000000 00:00 0                          [vvar]
        #      7ffd1306d000-7ffd1306f000 r-xp 00000000 00:00 0                          [vdso]
        #      ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
        #      $
        #     What we see is that there are three mappings to the binary file, a mapping for the stack and then some
        #     extra's used by the OS.
        #     5. you can now press enter in the terminal that the binary is running in
        #        this will cause the read to finish and the binary will then continue to execute

        # CODE to explore how we control the address space mappings
        # The smallest size of an address space mapping is called the page size
        # on Linux the default is 4096 (4Kb). To make it easier to identify
        # the mappings we will fill each section we add to a full page (4Kb)
        # worth of bytes.
        # This version 4 dyamically adds a "heap" area of memory
        # One page' of "text", one page of "data" and one pager of "readonly data"
        #  - text should get mapped to a region that is readable and executable (r-x)
        #  - data should get mapped to a region that is readable and writable (rw-)
        #  - rodata should get mapped to a region that is readable only       (r--)
        #  - bss    should get mapped to a region that is readable and writable (rw-)
        #  - heap   the brk system call is used to add a heap region that is

        .intel_syntax noprefix        # assembler syntax to use <directive>
        .section .text                # linker section <directive>

        .global _start                # linker symbol type <directive>
_start:
        # use a read system call to stop the program until user presses enter
        # so that we can examine things before and after the code runs
        mov rax, 0                  # read syscall = 0 -- read(fd, buf, len)
        mov rdi, 0                  # fd = rdi = 0 = stdin
        mov rsi, OFFSET rwdata      # buf = rsi = rwdata memory
        mov rdx, 1                  # len = rdx = 1 wait for one byte "enter"
        syscall

        # Now add a Heap
        # get current break pointer
        xor rdi, rdi                # pass 0 to brk (invalid request)
        mov rax, 12                 # brk syscall number 12
        syscall                     # call brk with 0

        mov r15, rax                # keep a copy of where our new memory starts
                                    # so we can use it later

        # now add some memory by requesting to increase the break
        # address by 32 bytes
        mov     rdi, rax             # mov current break into rdi
        add     rdi, 4096            # ask for 4096 bytes rdi=rdi+4096
        mov     rax, 12              # brk sycall 12
        syscall

        # use a read system call to stop the program until user presses enter
        # so that we can examine things before and after the code runs
        mov rax, 0                  # read syscall = 0 -- read(fd, buf, len)
        mov rdi, 0                  # fd = rdi = 0 = stdin
        mov rsi, OFFSET rwdata      # buf = rsi = rwdata memory
        mov rdx, 1                  # len = rdx = 1 wait for one byte "enter"
        syscall

        mov rax, 60                 # exit system call
        mov rdi, 0                  # exit status 0
        syscall

        # fill the text section up with zeros so that it is one page in size.
        .org 4095                   # force the next byte to be at the end of the page (4095
        .byte 0x00                  # put a byte of zero here

        ##### DATA SECTION : read and writable memory s
        .section .data
        .global rwdata
rwdata: .fill 4096,1,0xff           # 4096 bytes each initialized with a value of 0xff

        ##### READ ONLY DATA SECTION : read only data
        .section .rodata
        .global rodata
rodata:
        .fill 4096,1,'a             # 4096 bytes each initialized with a value of ASCII lower case a

        ##### BSS DATA SECTION: read and write automaitically added and filled with zero values
        .section .bss
        .global zerodata
zerodata:       .fill 4096,1
        # The above could be replaced with the single line: .comm zerodata, 4096, 1

        # the following tells the linux kernel to set permissions the way we expect
        # rather than the defaults which map all sections as executable
        # When using a compiler it will add this for us
        .section        .note.GNU-stack,"",@progbits
```

We have added more code so that you can pause the program before and after it makes the calls to brk so that you can see that heap get added.

1. **start the program running**
2. **follow the same steps as the other versions to look at the address space layout**
3. **then press enter – this will let the program continue. It will then run the code we added to add the heap area and pause again**
4. **again examine the address space layout – do you see the difference?**
5. **press enter again – this will again let the program continue and it will exit**

14.6.1.5. Version 5: giving memory back – shrink the heap back down

In this final version we add a final step to the program to shrink the heap back to its original size (0). This removes it from our process and give the memory back to the OS.

Here is the code we add – remember when we first called `brk` we stored the original location in `r15`

```
    # remove memory --  shrink heap back down to its original
    mov rdi, r15              # set break pointer back to the original location
    mov rax, 12               # brk syscall number 12
    syscall
```

CODE: asm - exploringASlayout5.s

```
        # To use this assemble and link it:
        # as -g exploringASlayout1.s -o exploringASlayout1.o
        # ld -g exploringASlayout1.o -o exploringASlayout1
        # Then using two terminals:
        #   1. use the nm command on the binary to see what addresses the linker
        #      placed all the symbols at Eg.
        #        $ nm exploringASlayout1
        #        0000000000403000 D __bss_start
        #        0000000000403000 D _edata
        #        0000000000403000 D _end
        #        0000000000402000 d rwdata
        #        0000000000401000 T _start
        #      (if you like you can also run: "readelf -S <binary>" to
        #       see more information about the sections in the binary)
        #   2. then run the binary -- its should stop on the read
        #      Eg.
        #      $ ./exploringASlayout1
        #
        #   3. in another terminal use ps and grep to find the process started from the binary
        #      Eg.
        #       $ ps auxgww | grep exploring
        #       jovyan      1697  0.0  0.0    168     4 pts/29   S+   13:41   0:00 ./exploringASlayout1
        #       jovyan      1699  0.0  0.0   6440   720 pts/30   S+   13:42   0:00 grep exploring
        #       $
        #      We see the process id (pid) is 1697
        #   4. using the process id you can now examine the file "/proc/<pid>/maps" to
        #      see the layout of the running processes address space
        #      Eg.
        #      $ cat /proc/1697/maps
        #      00400000-00401000 r--p 00000000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      00401000-00402000 r-xp 00001000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      00402000-00403000 rw-p 00002000 103:05 1189363                           /home/jovyan/exploringASlayout1
        #      7ffd13044000-7ffd13065000 rw-p 00000000 00:00 0                          [stack]
        #      7ffd13069000-7ffd1306d000 r--p 00000000 00:00 0                          [vvar]
        #      7ffd1306d000-7ffd1306f000 r-xp 00000000 00:00 0                          [vdso]
        #      ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
        #      $
        #     What we see is that there are three mappings to the binary file, a mapping for the stack and then some
        #     extra's used by the OS.
        #     5. you can now press enter in the terminal that the binary is running in
        #        this will cause the read to finish and the binary will then continue to execute

        # CODE to explore how we control the address space mappings
        # The smallest size of an address space mapping is called the page size
        # on Linux the default is 4096 (4Kb). To make it easier to identify
        # the mappings we will fill each section we add to a full page (4Kb)
        # worth of bytes.
        # This version 4 dyamically adds a "heap" area of memory
        # One page' of "text", one page of "data" and one pager of "readonly data"
        #  - text should get mapped to a region that is readable and executable (r-x)
        #  - data should get mapped to a region that is readable and writable (rw-)
        #  - rodata should get mapped to a region that is readable only       (r--)
        #  - bss    should get mapped to a region that is readable and writable (rw-)
        #  - heap   the brk system call is used to add a heap region that is

        .intel_syntax noprefix        # assembler syntax to use <directive>
        .section .text                # linker section <directive>

        .global _start                # linker symbol type <directive>
_start:
        # use a read system call to stop the program until user presses enter
        # so that we can examine things before and after the code runs
        mov rax, 0                # read syscall = 0 -- read(fd, buf, len)
        mov rdi, 0                # fd = rdi = 0 = stdin
        mov rsi, OFFSET rwdata    # buf = rsi = rwdata memory
        mov rdx, 1                # len = rdx = 1 wait for one byte "enter"
        syscall

        # Now add a Heap
        # get current break pointer
        xor rdi, rdi              # pass 0 to brk (invalid request)
        mov rax, 12               # brk syscall number 12
        syscall                   # call brk with 0

        mov r15, rax              # keep a copy of where our new memory starts
                                  # so we can use it later

        # now add some memory by requesting to increase the break
        # address by 32 bytes
        mov     rdi, rax           # mov current break into rdi
        add     rdi, 4096          # ask for 4096 bytes rdi=rdi+4096
        mov     rax, 12            # brk sycall 12
        syscall

        # use a read system call to stop the program until user presses enter
        # so that we can examine things before and after the code runs
        mov rax, 0                # read syscall = 0 -- read(fd, buf, len)
        mov rdi, 0                # fd = rdi = 0 = stdin
        mov rsi, OFFSET rwdata    # buf = rsi = rwdata memory
        mov rdx, 1                # len = rdx = 1 wait for one byte "enter"
        syscall

        # remove memory --  shrink heap back down to its original
        mov rdi, r15              # set break pointer back to the original location
        mov rax, 12               # brk syscall number 12
        syscall

        # one last read so that we can see what the as looks like prior to exiting
        # use a read system call to stop the program until user presses enter
        # so that we can examine things before and after the code runs
        mov rax, 0                # read syscall = 0 -- read(fd, buf, len)
        mov rdi, 0                # fd = rdi = 0 = stdin
        mov rsi, OFFSET rwdata    # buf = rsi = rwdata memory
        mov rdx, 1                # len = rdx = 1 wait for one byte "enter"
        syscall

        mov rax, 60               # exit system call
        mov rdi, 0                # exit status 0
        syscall

        # fill the text section up with zeros so that it is one page in size.
        .org 4095                 # force the next byte to be at the end of the page (4095
        .byte 0x00                # put a byte of zero here

        ##### DATA SECTION : read and writable memory s
        .section .data
        .global rwdata
rwdata: .fill 4096,1,0xff          # 4096 bytes each initialized with a value of 0xff

        ##### READ ONLY DATA SECTION : read only data
        .section .rodata
        .global rodata
rodata:
        .fill 4096,1,'a            # 4096 bytes each initialized with a value of ASCII lower case a

        ##### BSS DATA SECTION: read and write automaitically added and filled with zero values
        .section .bss
        .global zerodata
```

```
zerodata:       .fill 4096,1
        # The above could be replaced with the single line: .comm zerodata, 4096, 1

        # the following tells the linux kernel to set permissions the way we expect
        # rather than the defaults which map all sections as executable
        # When using a compiler it will add this for us
        .section        .note.GNU-stack,"",@progbits
```

Again we added another read so that you can explore what the address space looks like after we shrink the heap but before we exit.

So that's it -- we have seen how memory gets added to the process based on what sections we added in our source code and how we can make calls to the OS to add and remove "heap" memory while our program is running. There is a more advanced call in UNIX called `mmap` but we will leave it's discussion for another time.

By Jonathan Appavoo