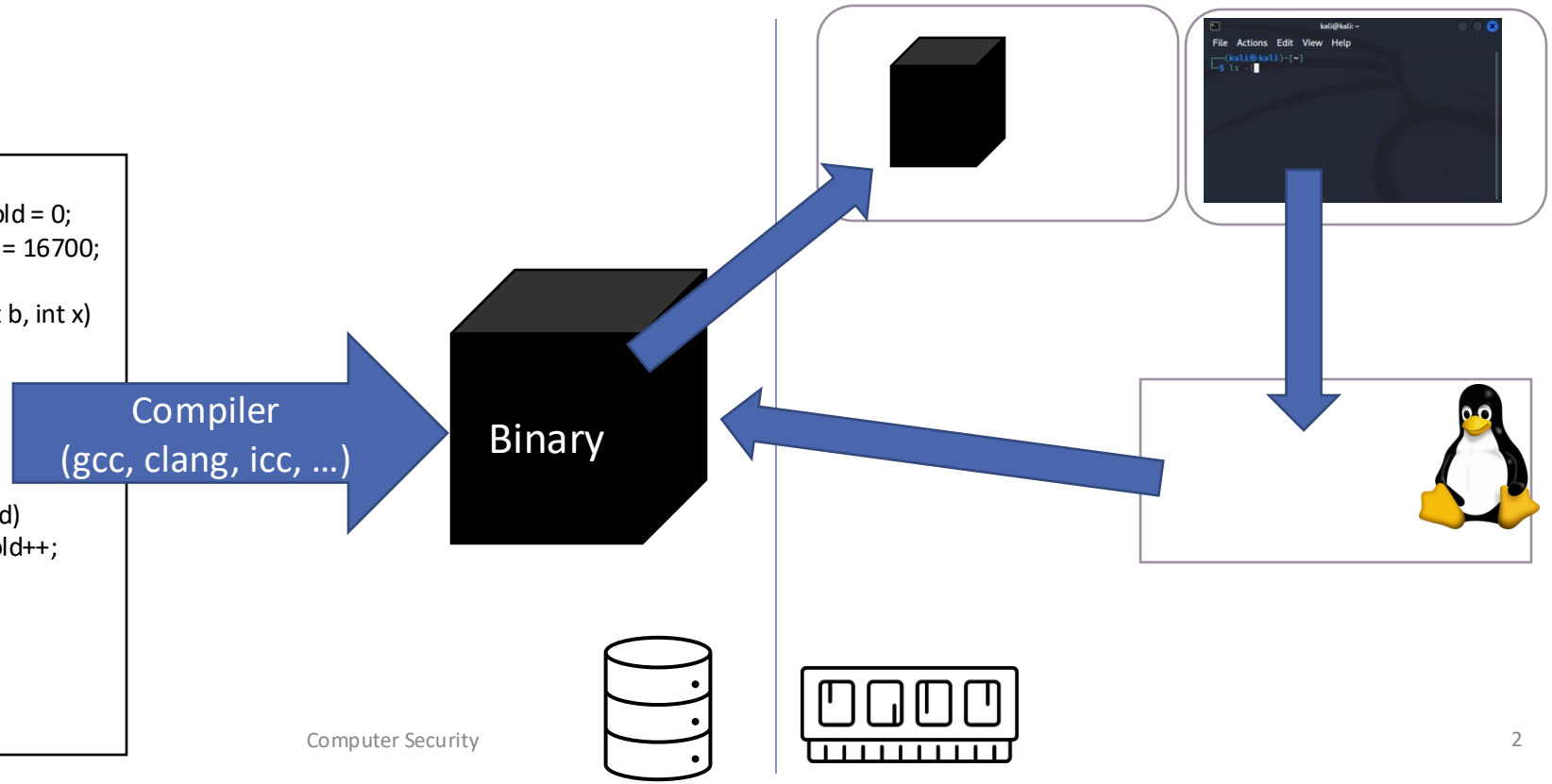
ELF & Program Anatomy

Georgios (George) Portokalidis

Recap

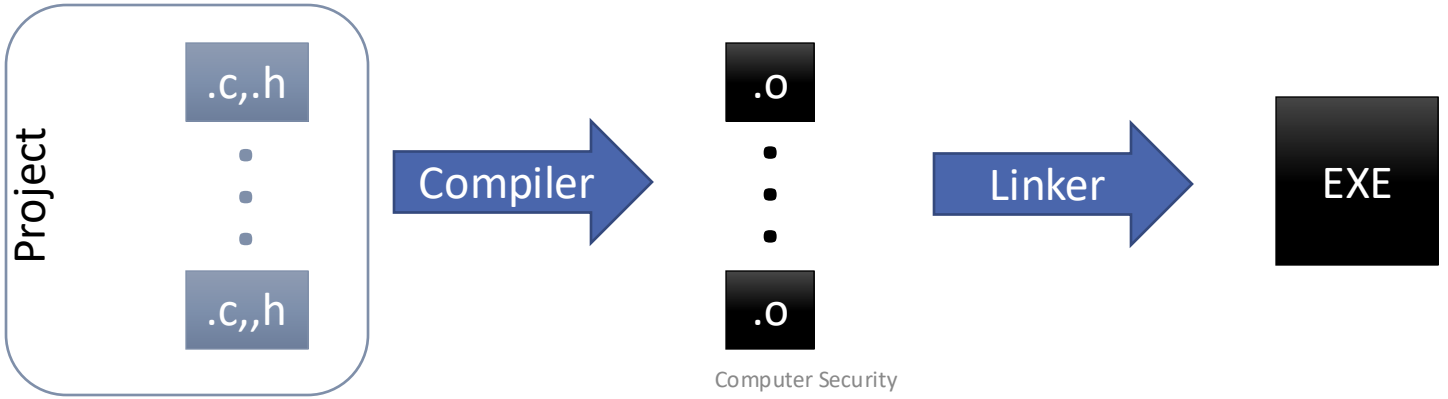
- Computer can only execute machine (binary) code
- C/C++ programs are compiled to binary code
- Binary (executable) programs are loaded to memory and executed by the operating system (OS)
  - Using a process

```
static int total;
static int total_over_threshold = 0;
static int arbitrary_theshold = 16700;

int simple_function(int a, int b, int x)
{
      int y;

      y = a*x*x + b*x;

      if (y > arbitrary_theshold)
            total_over_threshold++;

      total++;

      return total;
}
```



Compilation Process

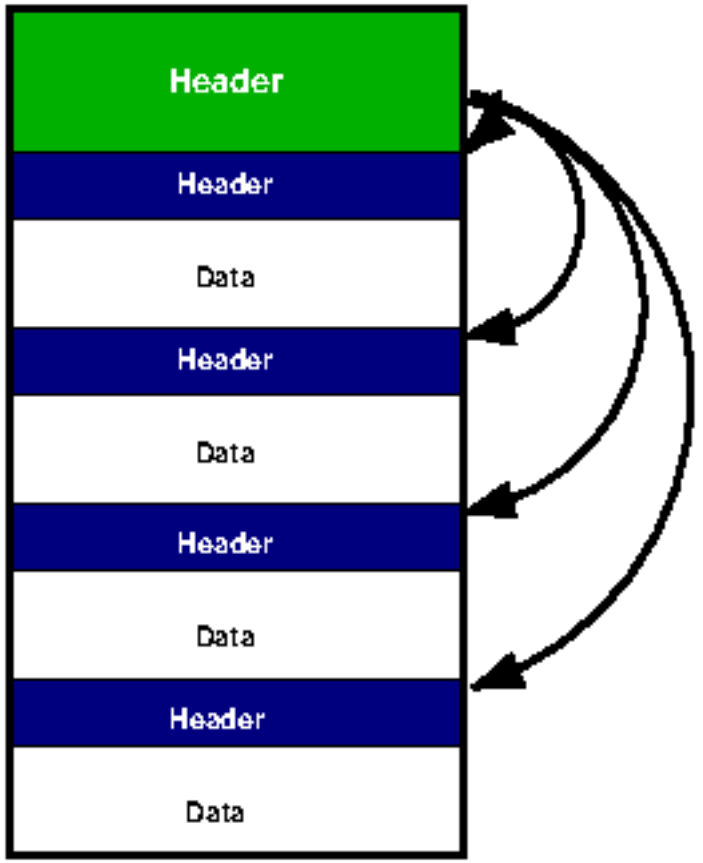
- Multiple stages
- The front-end parses source code into an internal representation (IR)
- Plenty of optimizations can be applied on the IR
  - May eliminate code, optimize control-flow, etc.
- The back-end generates machine code and stores it in an **object file**
  - Additional optimizations can occur at this point
- The linker combines multiple object files into an **executable file** including all metadata required to run program
- Common formats for executables and object files
  - **ELF: Executable and linkable format (modern *nix)**
  - COFF: Common Object File Format (old *nix)
  - PE: Portable Executable (Windows)



ELF – Executable Linker Format

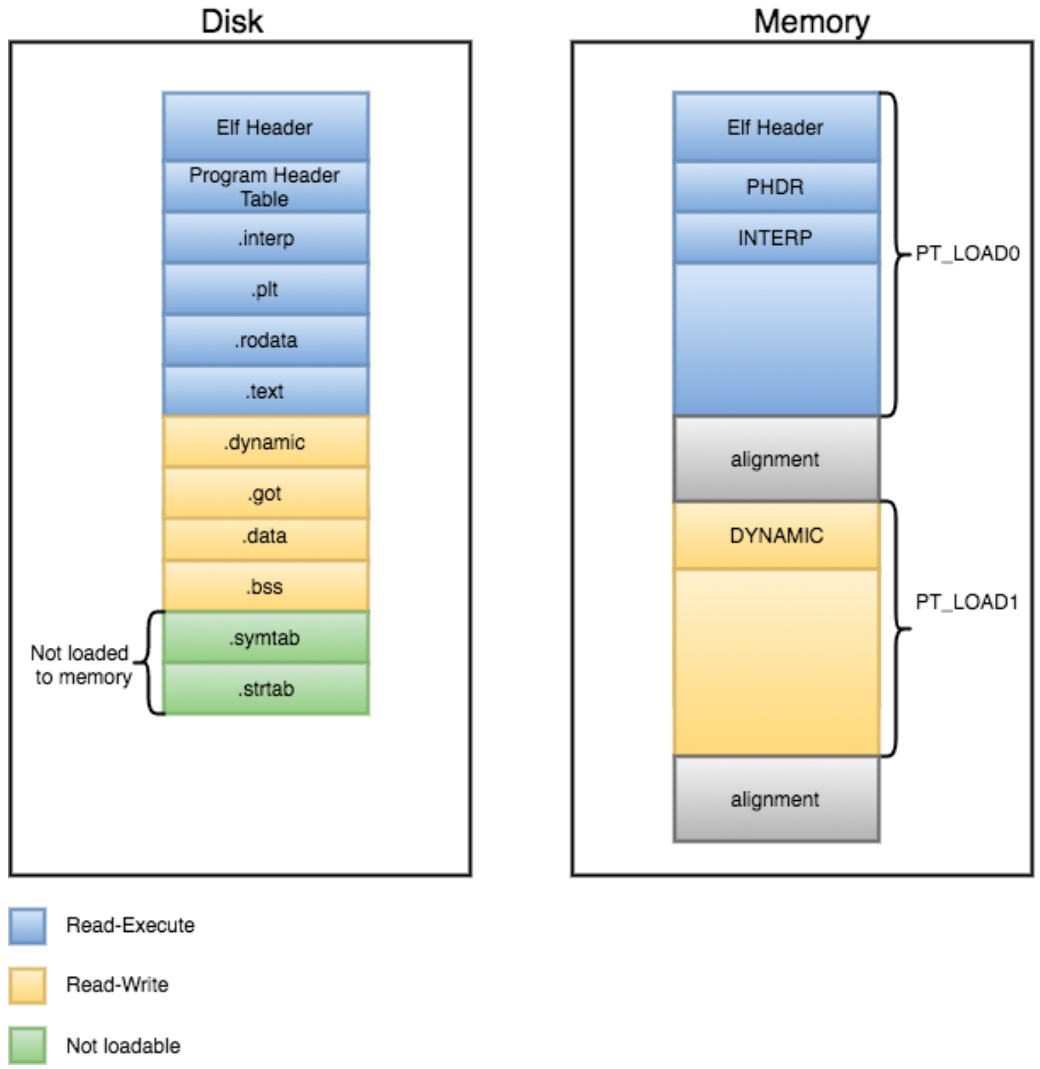
- A flexible format used in most *nix systems
  - Superseded the common object file format (COFF)
    - Portable Executable (PE) format used on Windows also evolved from COFF
- Used both for executables and libraries
- All ELF files begin with the magic number 0x7F 'E' 'L' 'F’

```
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |.ELF............|
00000010 02 00 3e 00 01 00 00 00 c5 48 40 00 00 00 00 00 |..>.....H@.....|
```



Section and Segments

- Section organize the binary into logical sections
  - Used by the linker and loader
- Segments define the parts that should be loaded to memory and how



Common Sections

Name	Description
.text	Code
.plt	PLT (Procedure Linkage Table)
.got	GOT entries dedicated to dynamically linked global variables
.got.plt	GOT entries dedicated to dynamically linked functions
.bss	Uninitialized data
.rodata	Initialized read-only data
.data	Initialized data

readelf

Running an Executable

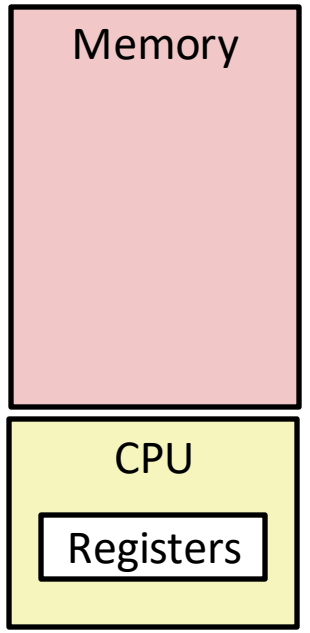
- The OS creates a process
- The loader loads the executable and library dependencies in the process address space
- Execution starts

Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- Process provides each program with two key abstractions:
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*

- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

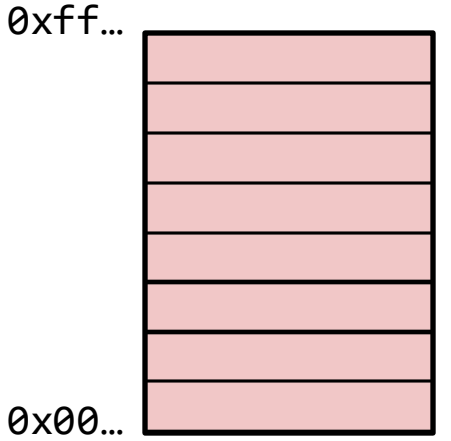


Virtual Memory

- A powerful concept for the address-space abstraction
- Decouples address space from physical memory
- Illusion of large and private address space
  - Address space can be larger than physical memory
- Programs can execute even if partially loaded in memory
- Frequently enables different memory protection policies

Virtual Memory and Paging

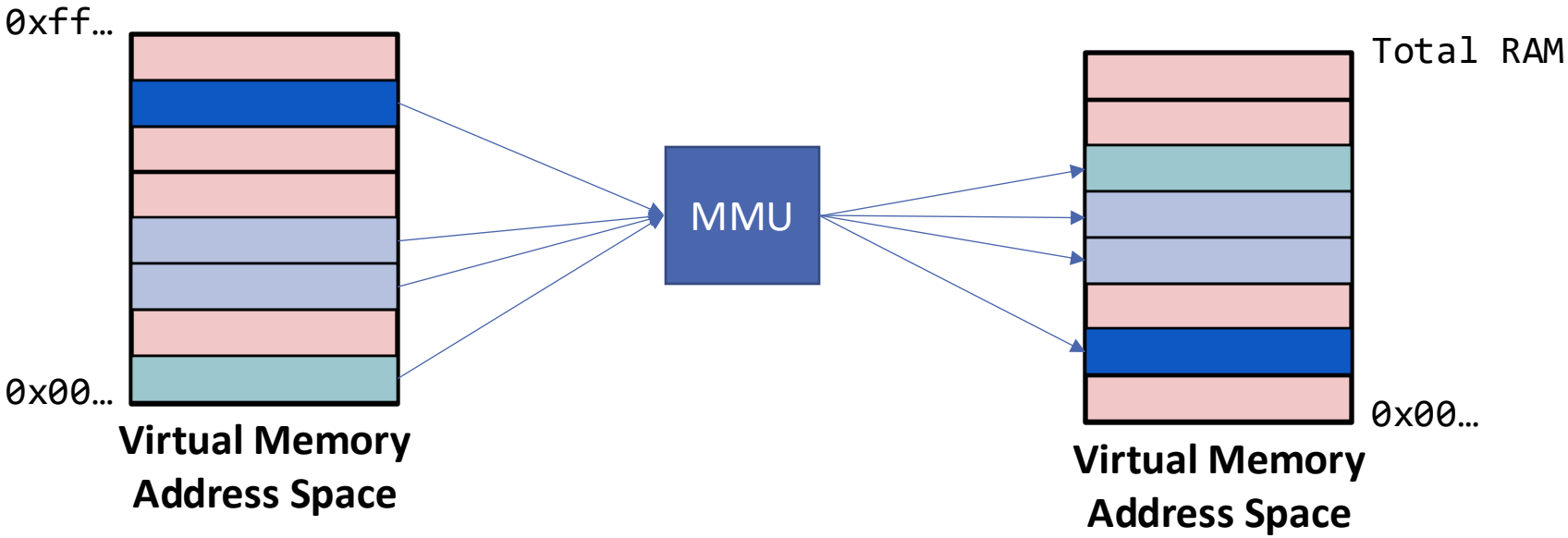
- The virtual address space is broken into **pages**
  - A page is a contiguous range of addresses
    - Mapped onto a contiguous range of physical memory
- Processes always sees the entire virtual address space
- Not all pages need to be in physical memory to run a program
  - Page switching (aka mapping/un-mapping) hidden from processes by the OS



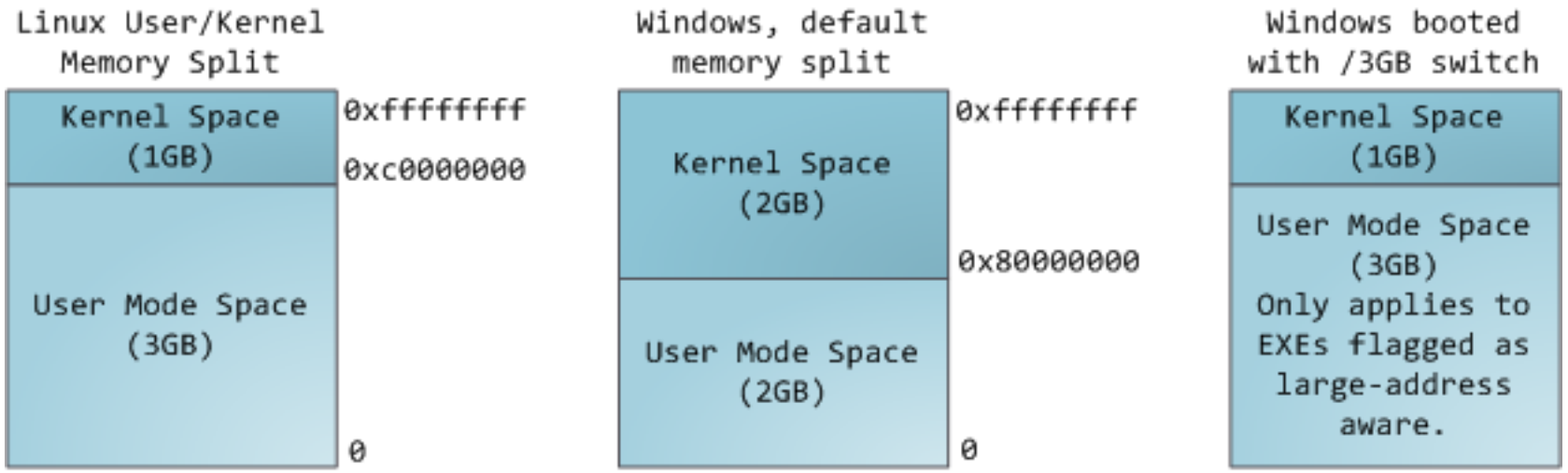
Virtual Memory Address Space

Virtual Memory and Paging

- The corresponding page units in physical memory are called **page frames**
- Pages and page frames are of the same size → usually, 4KB or 2MB
- The hardware memory management unit (MMU) maps pages to page frames



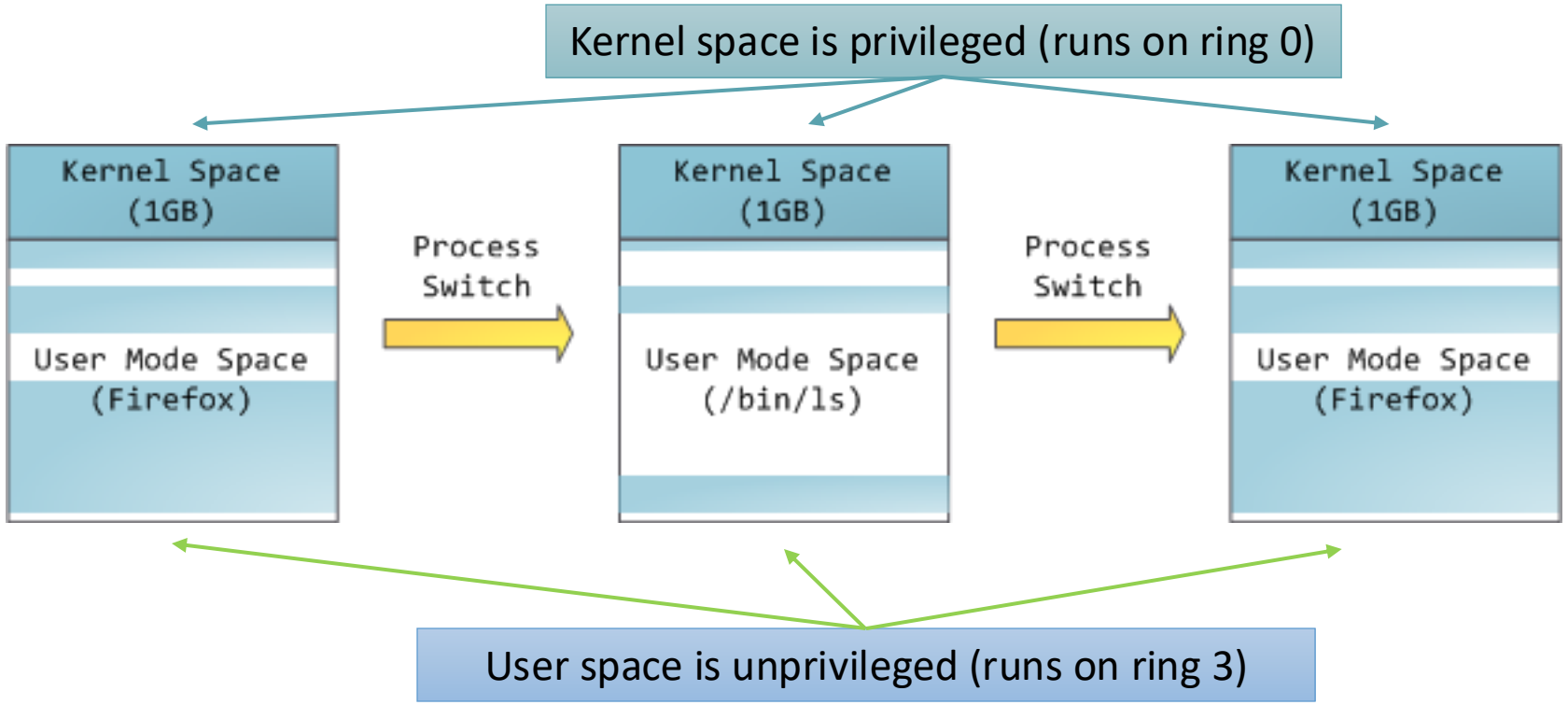
Anatomy of a Process in Memory (32-bit OS)



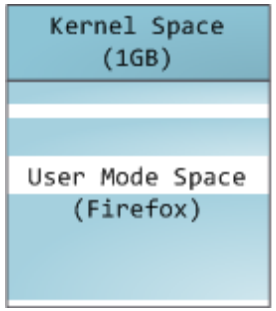
The kernel is always mapped!

Why Keep the Kernel Mapped?

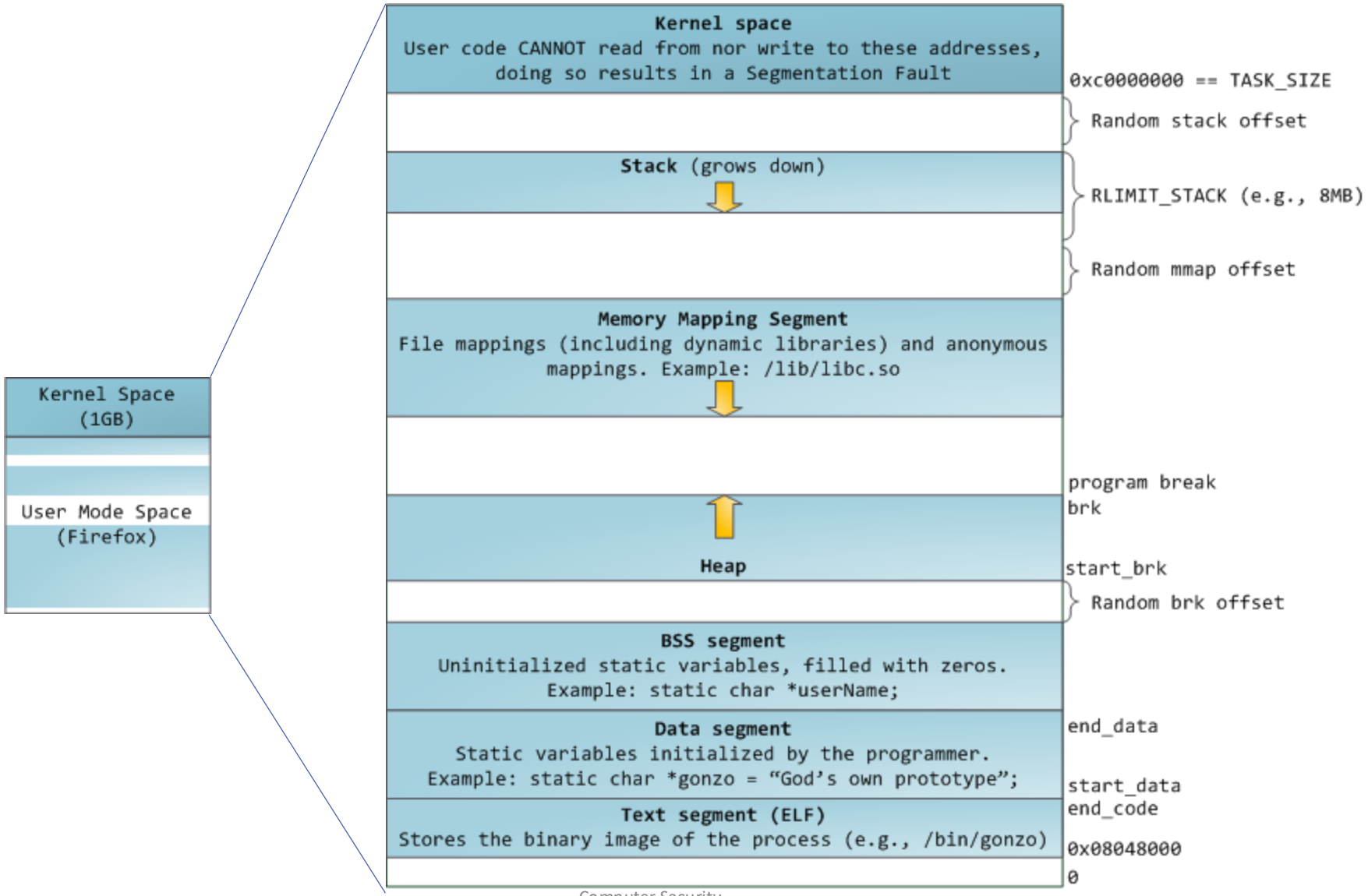
- The kernel can easily handle interrupts, system calls, etc.
- Process mapped each time changes to enable multitasking



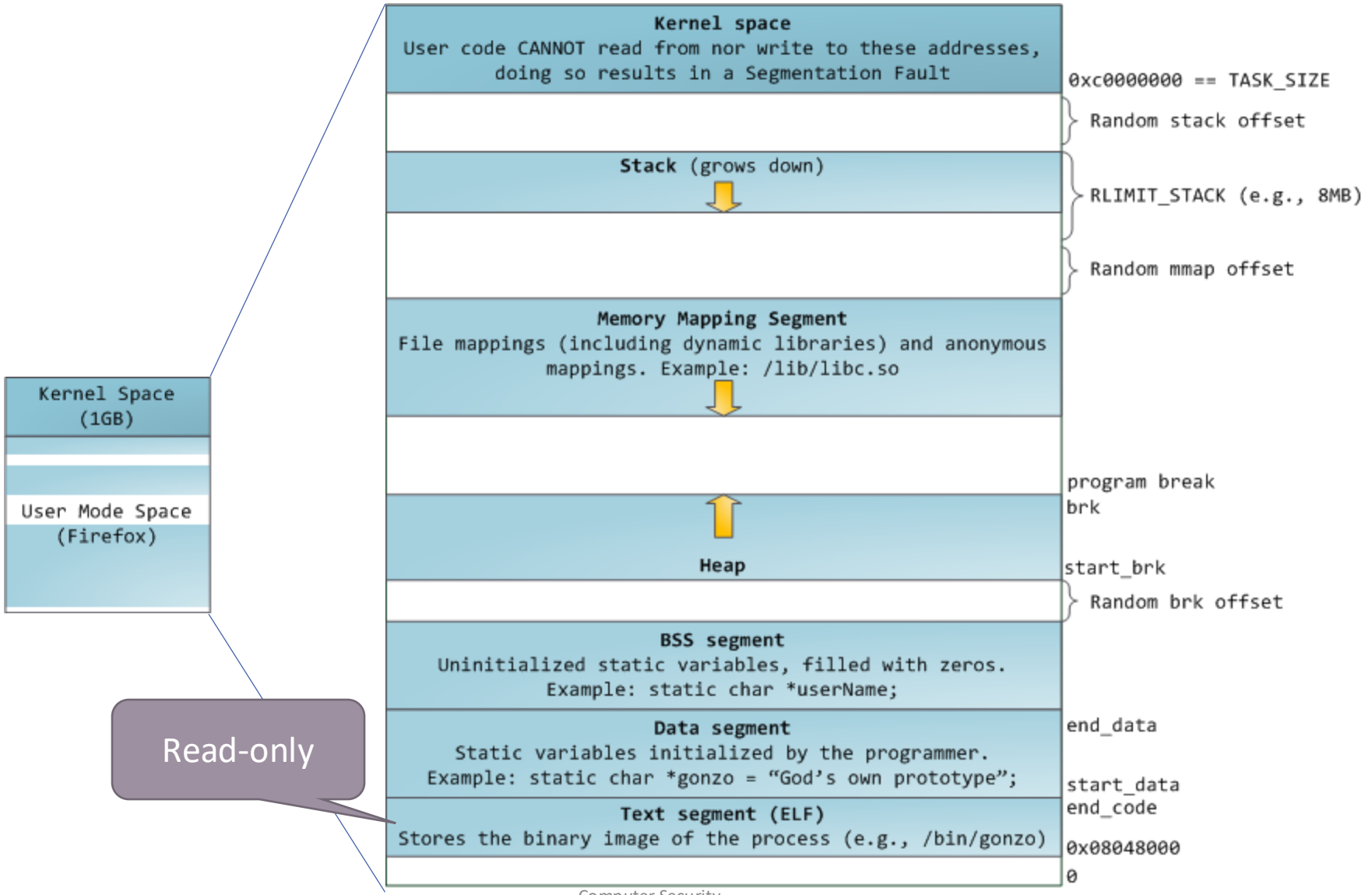
Anatomy of a Program in Memory



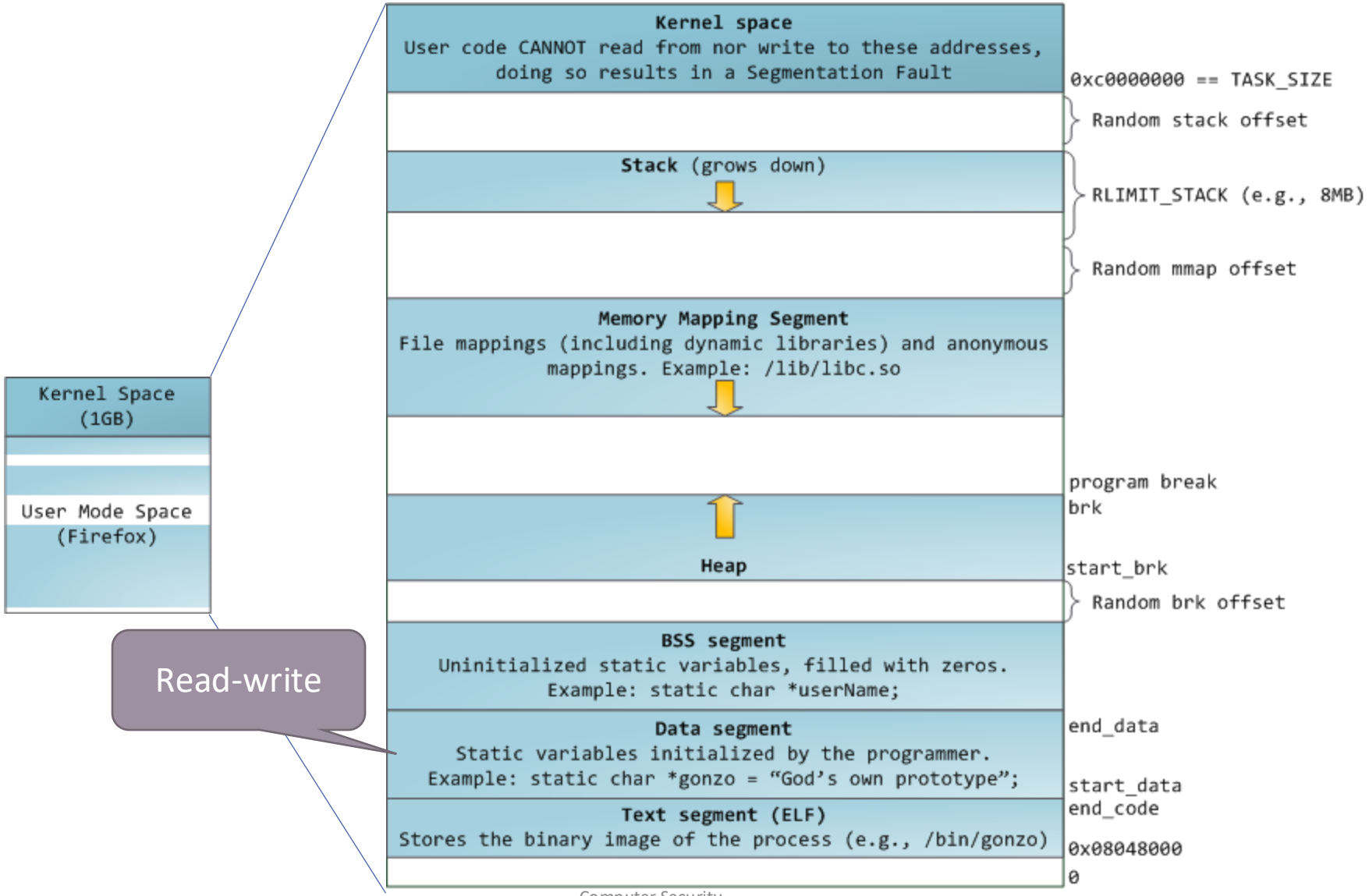
Anatomy of a Program in Memory



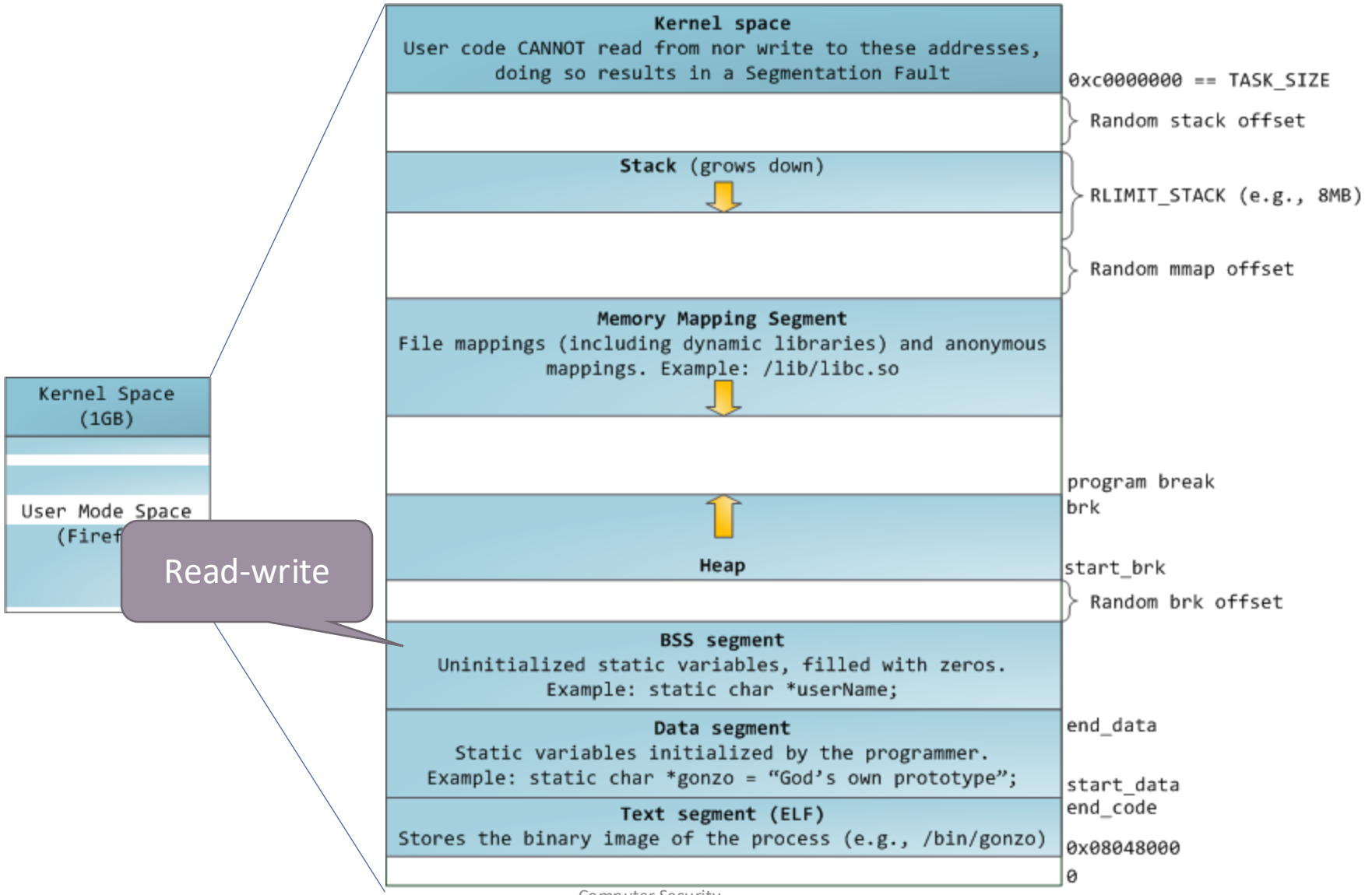
Anatomy of a Program in Memory



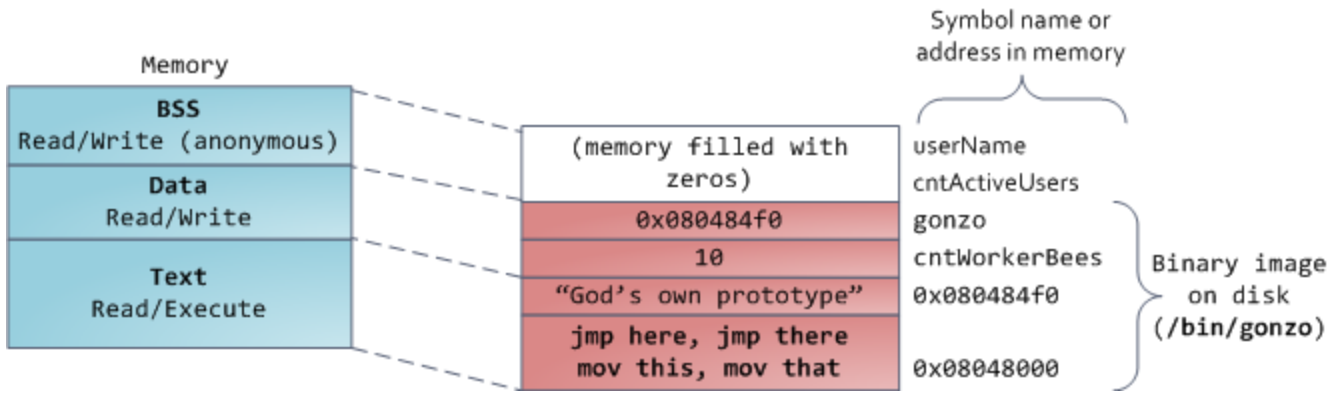
Anatomy of a Program in Memory



Anatomy of a Program in Memory

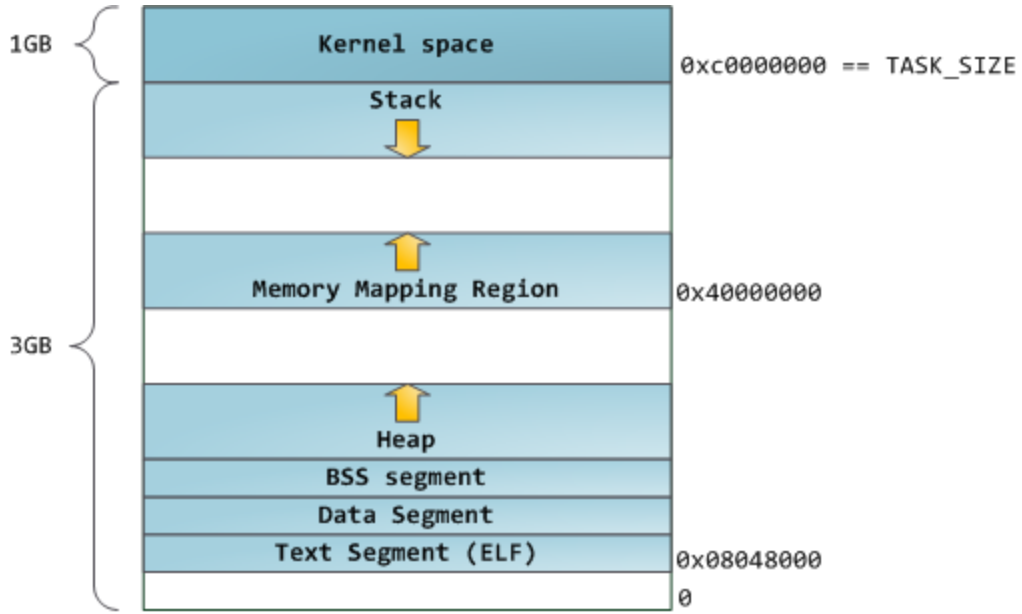


Initialized vs Uninitialized Data



Stack

- Contains arguments, local variables, function return addresses
- Read-write

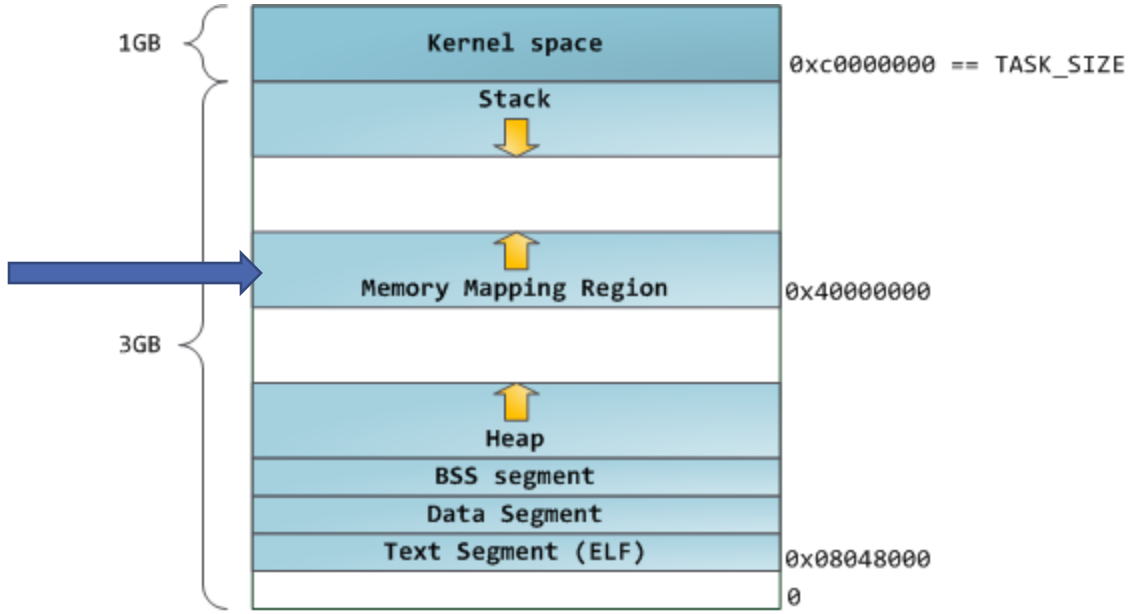


Heap

- Contains dynamically allocated data
- Allocated by the OS using brk() and sbrk()
- Programmers access it usually through malloc(), realloc(), calloc(), free()
- Read-write

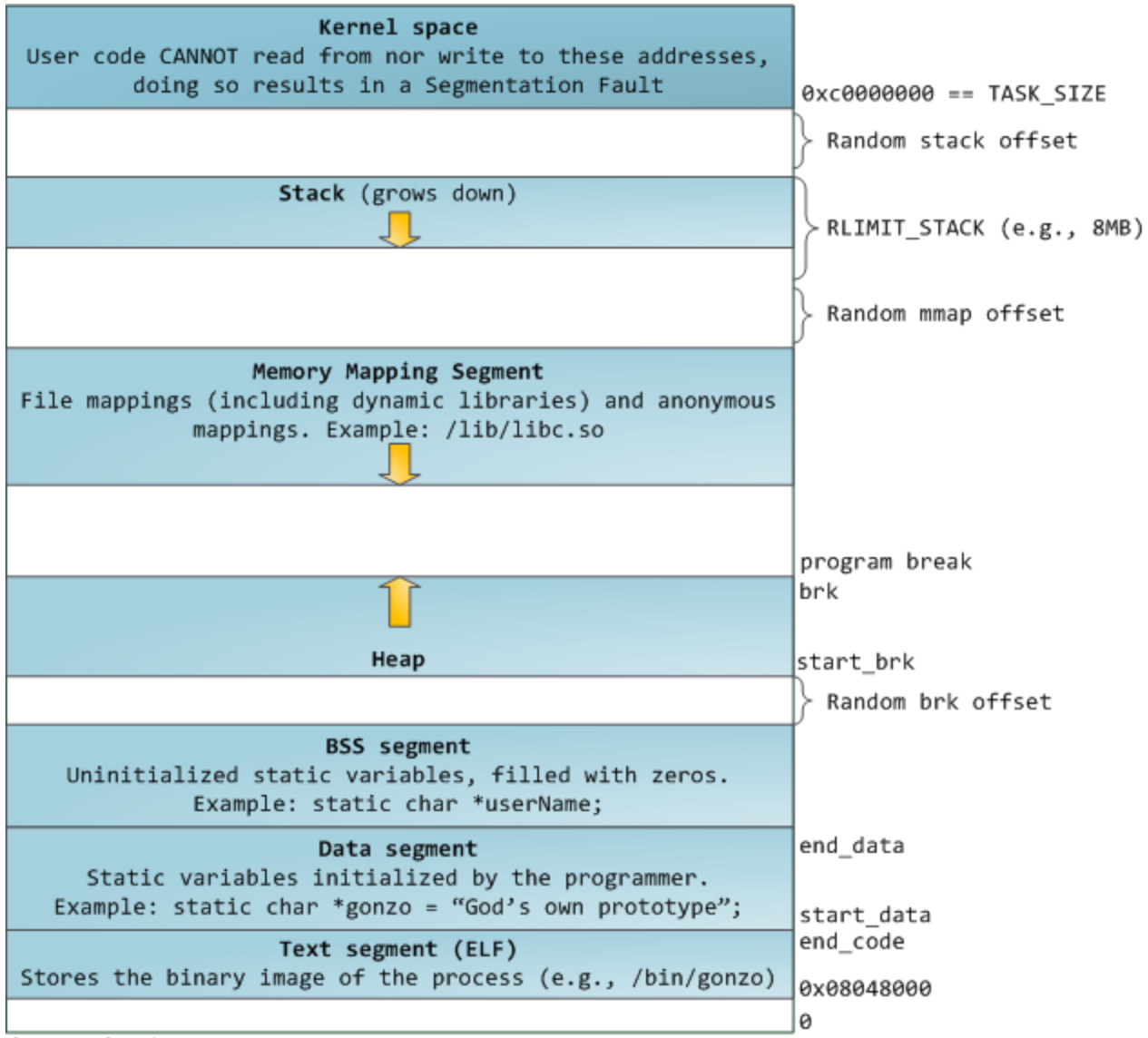
Large Heap Objects

- Allocators may decide to create an anonymous, private mapping to store large objects instead of directly storing into the heap

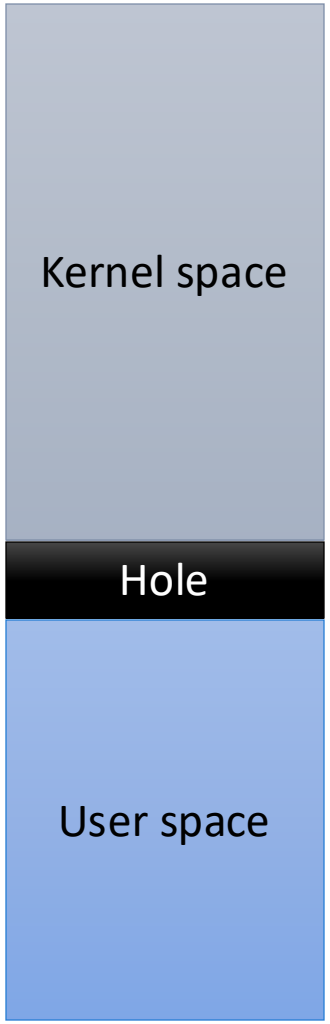


Anatomy of a Program in Memory

- Libraries are loaded in the memory mapping segment
- Each library has its own sections
  - .bss, .text, .data, .plt, etc.



Anatomy of a Process in Memory (64-bit Linux)




```
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole

ffffffffff600000 - ffffffffff600fff (=4 kB) legacy vsyscall ABI


... (rest of the kernel)



ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor

hole caused by [47:63] sign extension


0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
```

Memory Layout of Running Processes

- The /proc pseudo filesystem exposes different aspects of running processes
- Example:
  - The memory mappings of process with *pid* are in /proc/<pid>/maps
  - And the ones of the current process under /proc/self/maps

```
55d284dab000-55d284db3000 r-xp 00000000 08:01 2621442                    /bin/cat
55d284fb2000-55d284fb3000 r--p 00007000 08:01 2621442                    /bin/cat
55d284fb3000-55d284fb4000 rw-p 00008000 08:01 2621442                    /bin/cat
55d2855a8000-55d2855c9000 rw-p 00000000 00:00 0                          [heap]
7ff77787d000-7ff777b5b000 r--p 00000000 08:01 3805790                    /usr/lib/locale/locale-archive
7ff777b5b000-7ff777d42000 r-xp 00000000 08:01 1966683                    /lib/x86_64-linux-gnu/libc-2.27.so
7ff777d42000-7ff777f42000 ---p 001e7000 08:01 1966683                    /lib/x86_64-linux-gnu/libc-2.27.so
7ff777f42000-7ff777f46000 r--p 001e7000 08:01 1966683                    /lib/x86_64-linux-gnu/libc-2.27.so
7ff777f46000-7ff777f48000 rw-p 001eb000 08:01 1966683                    /lib/x86_64-linux-gnu/libc-2.27.so
7ff777f48000-7ff777f4c000 rw-p 00000000 00:00 0
7ff777f4c000-7ff777f73000 r-xp 00000000 08:01 1966133                    /lib/x86_64-linux-gnu/ld-2.27.so
7ff778133000-7ff778157000 rw-p 00000000 00:00 0
7ff778173000-7ff778174000 r--p 00027000 08:01 1966133                    /lib/x86_64-linux-gnu/ld-2.27.so
7ff778174000-7ff778175000 rw-p 00028000 08:01 1966133                    /lib/x86_64-linux-gnu/ld-2.27.so
7ff778175000-7ff778176000 rw-p 00000000 00:00 0
7ffe386b4000-7ffe386d5000 rw-p 00000000 00:00 0                          [stack]
7ffe387f9000-7ffe387fc000 r--p 00000000 00:00 0                          [vvar]
7ffe387fc000-7ffe387fe000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```