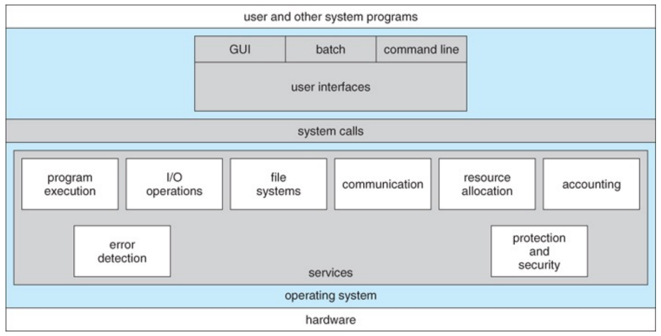
Operating Systems and Systems Programming (EECE 4029)



Spring Semester, 2024 – John C. Gallagher

University of CINCINNATI

1

Contact Information

Required Lectures:	M/W	8:00 AM – 8:55 AM	(Teachers-Dyer 350)
	F	8:00 AM – 8:55 AM	(Online with Zoom)
Instructor:	John C. Gallagher		
Email:	john.gallagher@uc.edu		
Office:	532 Mantei Center		
Office Hours:	TBD and by Appointment		
Online Chat:	Class Discord Server		

University of CINCINNATI

2

Course Overview

Introduction to concepts of modern operating systems and systems programming. Emphasis is on the concepts, algorithms and architectures of modern operating systems. Students will also learn Unix system programming such as synchronizing, inter-process communication, and networking.

3

Course Format

The course will be conducted as a series of modules. Each module will generally contain each of the following:

1) An assigned reading from the textbook
2) One or more timed, online, open-book quiz covering the assigned reading
3) One or more homework assignments requiring the implementation and/or evaluation of a key OS concept
4) Several lectures on key topics related to both the reading and the homework. *These lectures are intended to amplify and reinforce key concepts from the reading and to provide an opportunity for discussion and questions. They are not a replacement for the reading*.

There will also be a comprehensive final examination, on campus, during the class' normally scheduled exam period (Wednesday, April 24 – 8:00 AM to 10:00 AM).

4

Textbooks / References

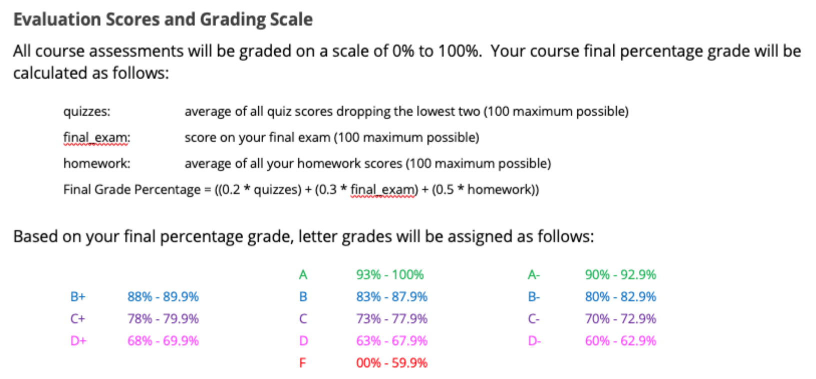
Required: *Operating System Concepts (10th edition)*
by Abraham Silberschatz, Peter B. Galvin, Greg Gagne;
ISBN-10:0470128720, ISBN: 978-1-119-32091-3.
https://www.os-book.com/OS10/

There will be assigned readings from the book on topics not directly covered in lectures and the book goes into more detail than we will be able to in lecture. This is NOT a bad book to have as a reference going forward.

University of CINCINNATI

5

Grading and Evaluation



Evaluation Scores and Grading Scale

All course assessments will be graded on a scale of 0% to 100%. Your course final percentage grade will be calculated as follows:

quizzes:	average of all quiz scores dropping the lowest two (100 maximum possible)
final_exam:	score on your final exam (100 maximum possible)
homework:	average of all your homework scores (100 maximum possible)

Final Grade Percentage = ((0.2 * quizzes) + (0.3 * final_exam) + (0.5 * homework))

Based on your final percentage grade, letter grades will be assigned as follows:

		A	93% - 100%	A-	90% - 92.9%
B+	88% - 89.9%	B	83% - 87.9%	B-	80% - 82.9%
C+	78% - 79.9%	C	73% - 77.9%	C-	70% - 72.9%
D+	68% - 69.9%	D	63% - 67.9%	D-	60% - 62.9%
		F	00% - 59.9%		

University of CINCINNATI

6

Module 01: Course Introduction, Concept Review, and OS Structures

University of CINCINNATI

7

Some thoughts before we begin…

University of CINCINNATI

8

This is an EXPANSIVE topic (Part 1)

- We will NOT cover “everything”. There is no way to do “everything” in the context of a semester-long first course
- We will visit a number of canonically important major topics, use the book readings to provide common language and concepts, and do exercises that are emblematic of each topic.

University of CINCINNATI

9

This is an EXPANSIVE topic (Part 2)

- Even the things we do cover will not be covered in full detail (E.G. all of the options possible for a specific system call, etc.)
- The end effect should be sufficient familiarity with major OS and systems programing paradigms to enable “deep dives” on details. You’ll be making such dives your whole professional life.

University of CINCINNATI

10

What is an Operating System?

University of CINCINNATI

11

What is an Operating System?

- There is no universally accepted definition of the term. The definition you get often depends on the perspective of the person giving the definition.
- A layman's definition might be: "The OS is whatever ships with the device" or "whatever is installed when I install an OS" (love the circular definition?)
- A more technical, but narrow definition, might be: "The code that is always running and manages the operation of all running code". This program is often referred to as the kernel.
- A more technical, but more broad definition, might be, the kernel and all system programs that are designed to assist the kernel and/or allow users to adjust how the kernel functions and makes decisions.

University of CINCINNATI

12

What is an Operating System?

- Going with the broader technical viewpoint, the OS is a program that acts as an intermediary between users of a computer and one or more programs AND as an intermediary between programs and hardware resources.
- Operating system goals:
  - Execute users' code in a manner that is safe, secure, efficient, and reliable.
  - Provide users with a convenient interface through which to interact with their running code.
  - Provide users with a convenient interface through which to modify how their running programs interact with one another (permissions, communication, resource allocation, etc.)

University of CINCINNATI

13

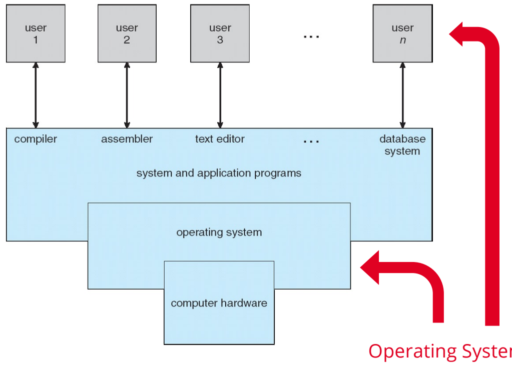
What is an Operating System?

- The OS is a **resource allocator**
  - It manages all resources. These may include I/O devices, CPU, memory, etc.
  - It decides what resources get assigned to which programs when to minimize conflict and maintain fair use of resources. It also ensures that ALL programs get needed resources eventually and that programs don't "deadlock" because of poorly planned allocations of resources
- The OS is a **control program**
  - It controls the execution of ALL other programs to intercept and prevent improper use of the computer.
  - It takes control in user program error states to provide a graceful shutdown and/or isolation of the problem causing code so that other programs are not damaged or themselves put into error states.

University of CINCINNATI

14

What is an Operating System?

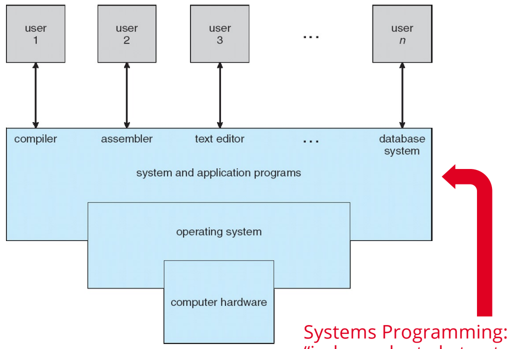


Operating Systems – How does it work? How can we use that knowledge to prevent us from making bad decisions up here

University of CINCINNATI

15

What is Systems Programming?



Systems Programming: Instead of viewing our code as "independent abstract entities" We use our OS knowledge and knowledge of specific interfaces to evoke useful effects

University of CINCINNATI

16

Computing Systems Concept Review

17

Traps, Interrupts, and Faults

- Operating Systems need to do each of the following:
  - Allow a *secure* way for user-level code to ask the OS to perform privileged operations for it
  - Be able to properly respond to asynchronous requests for service from external hardware
  - Be able to detect and deal with error and fault conditions that can happen at any time.
- Traps, Interrupts, and Faults are CPU-level, hardware tools for supporting each of the above needs.

18

Traps, Interrupts, and Faults

- All three mechanisms are *very* similar in how they are implemented – so there is often confusion about the terminology. We'll adopt terms that are a slight expansion of what the book uses as definitions. Please note that you won't always see this terminology used in exactly this way, though it's what we'll adopt for our purposes.

University of CINCINNATI

19

Traps, Interrupts, and Faults

- What do they have in common?
  - They are all triggered by "events" that the CPU can directly sense
  - They all result in "vectored" subroutine calls, again, at the CPU/Assembly Level
- What is different about them?
  - The events that trigger the vectored call
  - Where exactly control is "returned" after the vectored call completes

University of CINCINNATI

20

Traps, Interrupts, and Faults

- What are vectored subroutines?
  - Events are associated with memory locations that contain an address where lives a ”servicing routine”. All of the addresses of all of the servicing routines for each event are stored in a ”trap” or “vector” or “interrupt” table in memory locations hardwired into the CPU.
  - When the event occurs, the current program counter is saved and the program counter is set to the address contained in the table slot associated with the event. Often, ALL program state is saved, either by the CPU or by the author of the service routine. If the CPU has user and privileged modes (more on this later), privilege is generally escalated by a vector event.
  - This is EXACTLY like a subroutine call except that the target to which the PC jumps is fetched from a table entry associated with the event
  - The idea is that when events happen, there is a link to a routine to “service” the event. That table of service routines is presumably loaded at the time the OS loads and then is protected from changes by any mere user. The OS can then respond to events in a defined way that in a protected mode.

University of CINCINNATI

21

Traps, Interrupts, and Faults

- What are the events?
  - TRAPS: The event is triggered by software. Most CPUs have a “trap” assembly language instruction that allows a user to request that a specific, numbered, trap routine is run. When the TRAP servicing routine returns, it returns to the address JUST AFTER the address of the TRAP call
  - FAULT: The event is triggered by a software request that resulted in an error condition of some kind. When the FAULT servicing routine returns, it returns to the ADDRESS OF THE ASSEMBLY INSTRUCTION THAT CAUSE THE FAULT

University of CINCINNATI

22

Traps, Interrupts, and Faults

- What are the events?
  - INTERRUPTS: The event is triggered by a message from some hardware device outside the CPU. The message is received either from a dedicated line on the CPU or via a common communications bus. When the interrupt servicing routine returns, control is returned to the instruction that would have run if the interrupt hadn't occurred.

University of CINCINNATI

23

Now, From where CoNfuSion?

- Some people (old types like me) tend to use the word "interrupt" for all the things. They MIGHT refer to "hardware interrupts" and "software interrupts" to mean what we previously termed "interrupts" and "traps"
- What happens if an interrupt happens while an interrupt is being processed?

University of CINCINNATI

24

What is CPU Privilege Mode?

- In old school processors, any program could execute any instruction and access any part of memory. This made it difficult to program kernels to impose security. If every process can do ALL THE THINGS at the assembly language level, how could anyone ever think the kernel could keep processes safe from one another?
- In modern processors, there is usually some hardware switch in the CPU that defines the level of privilege. The so-called "privileged" mode allows any instruction to be run and/or any memory access to be done. One or more lower modes, often referred to as "user" modes, only allow SOME CPU instructions to be ran. Also, there may be some hardware supported mapping of memory addresses that includes address rewriting and/or marking of certain areas of memory off-limits.
- The kernel runs in privileged mode. User processes run in a "user" or non-privilege mode. This means user processes can not do "all the things". When they need those special things, they ask the kernel via system calls, that ultimately map into TRAP instructions into kernel code. Let's go into more detail on that

University of CINCINNATI

25

Interrupts and Traps and Privilege

Key Concept: TRAPS AND INTERRUPTS ELEVATE PRIVILEGE and TRANSFER CONTROL TO THE SUBROUTINE as one "ATOMIC" step that cannot be interrupted at the assembly code level.

IF TRAPS/INTERRUPTS ARE THE ONLY WAY TO ELEVATE PRIVILEDGE, THEN REQUESTING HIGHER PRIVILEDGE ALWAYS GIVES CONTROL BACK TO KERNEL CODE IF EVERY ITEM IN THE VECTOR TABLE POINTS TO KERNEL CODE AND THE KERNEL PROTECTS FROM CHANGES TO THE VECTOR TABLE

IF ALL SERVICING ROUTINES LOWER PRIVLEGE BEFORE LEAVING… WHAT IS THE OVERALL IMPLICATION?

University of CINCINNATI

26

TRAPs and Privilege

There will also be at least one assembly instruction that is meant to lower privilege. For sake of discussion, let's call this instruction: USRPR for "set CPU to user privilege mode"

System calls would map to specific TRAPs. Each TRAP service routine would look like this:

27

TRAPs and Privilege

Conceptual Kernel Service Routine

```
service_start:  OPCODE  OPERAND       // Note, if we got here via a trap from some
                OPCODE  OPERAND       // "user code", our privilege is already elevated.
                OPCODE  OPERAND       // Generally the opening block would do the following
                OPCODE  OPERAND       // a) Get parameters from memory controlled by the
                OPCODE  OPERAND       // user code and b) maybe mask some specific
                OPCODE  OPERAND       // interrupts or traps

                OPCODE  OPERAND       // The next block would do the work of the system
                OPCODE  OPERAND       // call, then return results into memory that the
                OPCODE  OPERAND       // user code controls.

                USRPR                 // Put CPU into user privilege mode
                RETURN                // Return the program counter to where it was when
                                      // the event happened.  If user code was running when
                                      // the event happened, then you're going back to user
                                      // code… hurrah!
```

28

TRAPs and Privilege

Conceptual User Code System Call

```
OPCODE   OPERAND          // The user code doing user code things
OPCODE   OPERAND          // More user code things

OPCODE   OPERAND          // Set up MY OWN MEMORY with parameters to the
OPCODE   OPERAND          // system call I'm going to make

TRAP     TRAPNUM          // Make the system call

OPCODE   OPERAND          // Arrange whatever came back (in memory I control)
OPCODE   OPERAND          // for use by this user code

OPCODE   OPERAND          // The user code going about its normal business
OPCODE   OPERAND          // More going about its business
```

29

TRAPs and Privilege

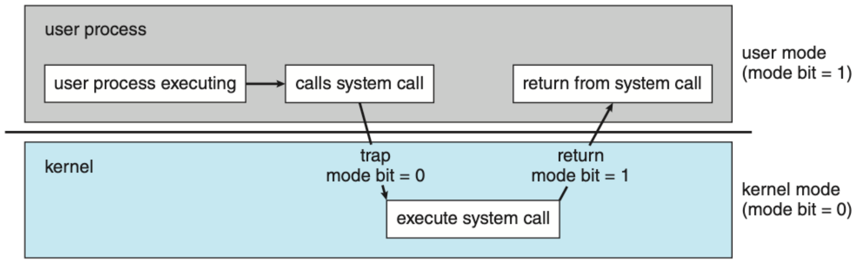


Figure 1.13 Transition from user to kernel mode.

30

Interrupts and Keeping Control

What's to keep a user program from grabbing control and never giving it back to the kernel? Interrupts to the rescue ☺

A hardware timer could generate an interrupt event to the CPU on a periodic basis. The service routine for the timer event could "decide" if control needs to be given to the same user process that's running, a new one, or even the kernel.

University of CINCINNATI

31

Faults and Virtual Memory Faults

Later we'll talk about virtual memory systems in which we create the illusion of having much more memory available than there really is in hardware

Sometimes a process will request access to "memory" that is not actually IN physical memory. This would generate a FAULT, the kernel would take control, put the requested info IN physical memory, then restart the user process in a way that it tries to redo the request – this time not faulting

University of CINCINNATI

32

Interrupts and Dealing with I/O

- I/O is orders of magnitude slower than CPU operations.
- Why should a CPU be micromanaging I/O when it could be doing something else?
- The CPU could ask a DMA controller do do some work and go off to do other things. An interrupt from the DMA would tell the CPU to now deal with the results of that much slower operation
- We'll really dig into this use of interrupts later…

University of CINCINNATI

33

So, Where are We At?

- In a VERY simple world, there is only one program running on a CPU. This program is the only game in town, it has access to all resources, and can do anything. You would have used this model in you Assembly Courses
- In a slightly more complex world, there would be a bunch of pre-written routines that were loaded into memory at boot. There is still only one program running on the CPU, but those utility routines are "just there" for use. Sometimes these "utility" routines are implemented as TRAPS. This may be something like and OK, but only barely

University of CINCINNATI

34

So, Where are We At?

- A fully featured OS would be more than just a collection of subroutines. It would
  - Actively manage the running of multiple programs in multi-tasking/multi-programming mode. Timer interrupts are UNDOUBTEDLY involved in giving the kernel control periodically to do management
  - Provide SAFE and SECURE mechanisms for user programs to request services (TRAPs are definitely involved here)

University of CINCINNATI

35

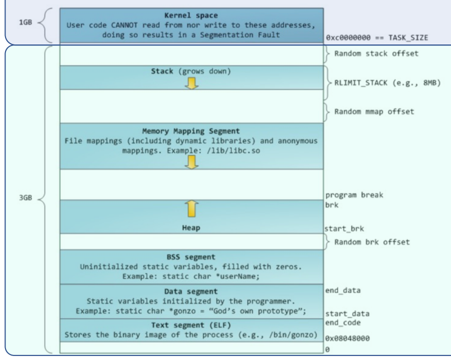
So, Where Do We Need to Go?

- Our next steps are:
  - To understand how we can even get this "kernel" and "OS" loaded onto the machine and get it running so it's there to manage things. This is NOT an unsubtle thing to do. Getting this wrong mean… no security
  - To understand the basic model of a process. Once the kernel is running, what exactly is it managing? Well… processes, which are collections of code and runtime state

University of CINCINNATI

36

So, What IS the Kernel and Where does it Live?


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Virtual Addresses == Physical Addresses

Virtual Addresses != Physical Addresses

Virtual Addresses are References into Virtual Memory System

The PROCESS sees the virtual addresses and believes them to be real. Behind the scenes, the kernel and memory mapping hardware in the CPU are on the fly translating PROCESS virtual addresses into references to physical memory. The kernel may also be moving process memory to long term slow storage, bringing it in only if needed. We will cover this extensively in a later unit. For now we'll just note it is going on.

37

Loading a kernel…

1. Somehow the kernel gets loaded into kernel space and starts running. Once the kernel is running, it imposes a model of user (and system) code as "processes" that get loaded and unloaded (using virtual memory methods) into the rest of user space (note, what I just said was a bit of a lie, we'll come back to that)
2. Upon loading, the kernel will notice that some system level helper processes are not there, and will launch the "master process" that will instantiate the rest of the OS (as processes)

38

Loading a kernel…

3. One of those launched processes will be a login interface that users can use to get in and launch their own processes. That deep into the process tree, these will be running with lower privilege.

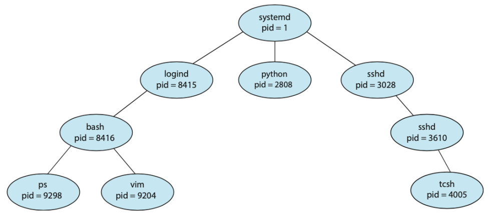
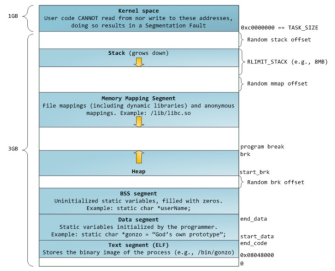


Figure 3.7 A tree of processes on a typical Linux system.


1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;
Data segment
Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";
end_data
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

39

Loading a kernel

- To load and activate a kernel, we have to go from a situation where the processor is first powered up, with nothing in memory, to a situation where the actual physical memory where the kernel is to live has the kernel in it, and the kernel and system helper processes are running and ready to accept user-level work, which will launch user-level processes to do that work.
- From there, the kernel will manage which processes (user or system) are getting access to memory and CPU. User processes can't interfere because of the control of privilege. Any user process trying to do privileged things trigger a trap… which gives control to the kernel to presumably not allow those things.
- To get to the dance, we need to go through a boot sequence….

40

Boot Sequence

Boot sequences can differ in the details, however, in general, all look very similar across operating systems. Below we'll note both the general concept and the specific implementation of that concept on the machines you'd be using for this course.

1. When a machine first boots, it generally runs some local firmware to run basic diagnostics and then load, from a very specific memory device, a very minimal bootloader. For the machines used in this class, this basic firmware is called the "Power On Self Test (POST). POST will run the basic hardware diagnostics. If they fail, it is assumed there is some major hardware fault and the rest of the process is terminated. If the machine does pass the diagnostics.

2. After diagnostics, control is passed to a firmware bootloader. This bootloader will be VERY basic and have only minimal knowledge of how to access disks. Generally it knows only how to access a small number of raw disk blocks, sometimes only ONE such block. The primary bootloader will get a secondary, slightly more complex bootloader from that block. That secondary bootloader may itself load the kernel, or it may itself be one of a sequence of loads of successively more complex loaders until one capable of loading the kernel itself is resident in memory. For the machines used in this class, the Basic Input Output System (BIOS) is resident on firmware and contains the initial bootloader as well as some code one can use to configure some low-level aspects of the initial boot (I.E. which disk device to look at first, some security settings) and has some old legacy interrupt (trap) code that would have been used basic I/O from MS-DOS. The "last action of POST" is to trigger the Intel x86 architecture INT x13 instruction, which is a software trap to slot x13 of the defined trap vector table. This gives control to the BIOS routine that goes to the "default" boot device and gets its "Master Boot Record" (MBR). The MBR is only 446 bytes long. BIOS gets those 446 bytes and runs it. That 446 byte program's only job is to get a more complex "next stage" boot loader and run it.

University of CINCINNATI

41

Boot Sequence

3. The primary boot loader would have been put into the "Master Boot Record" by whoever did the OS install to that disk device. This is fetched by the BIOS and will do its thing, whatever that is. On modern Linux installs, this is most often GRUB2 (GRand Unified Bootloader), although GRUB and LILO are still seen in the field. GRUB2 puts a "stage 1" bootloader into the MBR. That first stage's only job is to find a GRUB "stage 1.5" code block that exists in the space between the MBR and the beginning of the first logical, filesystem containing, partition on the disk device. This stage 1.5 is complex enough to actually understand a few common filesystems. This "new stage, unless configured otherwise, will searche the filesystem's logical structure to find the "stage 2" loader in /boot/grub2.

4. Eventually a stage is launched that is "smart enough" to find and launch a Linux kernel. Often this final stage launcher has more advanced features like a menu from which to select from among various kernels. Stage 2 of GRUB2, if configured properly (see lab/homework assignment) presents a menu allowing choice of kernel and some other things. Its job is, again, to launch the kernel.

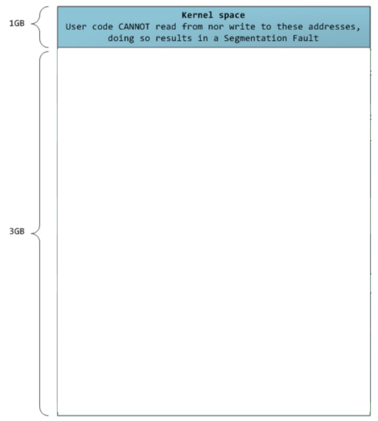
5. When the kernel is running, it will, if it sees that `systemd` is not present, launch it as "the first process". That process will load into "user space" – but with the powers of a system operator, and it will, using configuration files in /etc, build out a tree structure of related processes that include the "real" user processes. The kernel will manage all the processes in the tree by swapping them in and out of memory as needed and managing relationships among them and between then and physical resources. Occasionally timer events will give control to the kernel, which will take control and decide what processes in the tree get to be loaded and active. Yes… this is complex. That's why it's a whole course ☺

University of CINCINNATI

42

Boot Sequence – The End of the Beginning

At the end of boot, but before the kernel does anything, memory (on a 32-bit system) will look like this

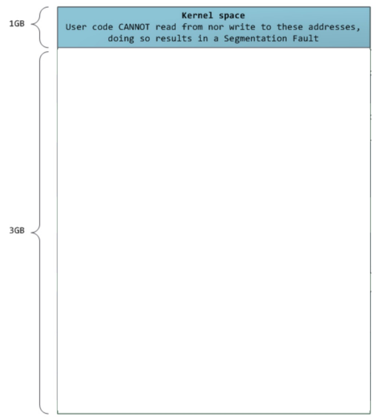

1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
3GB

43

Boot Sequence – The End of the Beginning

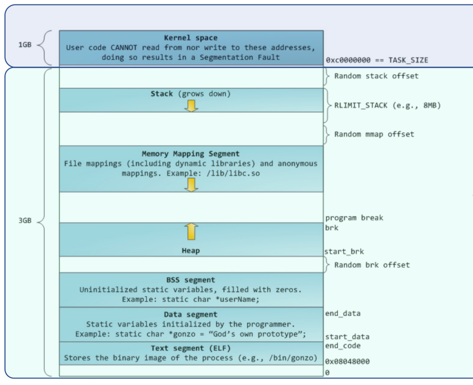
The final stage bootloader will pass program control to the kernel. It will be operating in privileged mode. The first thing it does is bring into existence the mother of all processes: **`systemd`**

(https://en.wikipedia.org/wiki/Systemd)


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
3GB

44

What is a process – Memory View


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
3GB
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Virtual Addresses == Physical Addresses

Virtual Addresses != Physical Addresses

Virtual Addresses are References into Virtual Memory System

The PROCESS sees the virtual addresses and believes them to be real. Behind the scenes, the kernel and memory mapping hardware in the CPU are on the fly translating PROCESS virtual addresses into references to physical memory. The kernel may also be moving process memory to long term slow storage, bringing it in only if needed. We will cover this extensively in a later unit. For now we'll just note it is going on.

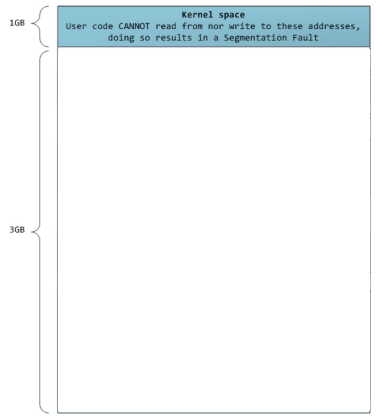
University of CINCINNATI

45

Boot Sequence – The End of the Beginning

Side note: `systemd` is not the only game in town. A comparison of other "mother-of-all processes" can be found at:

https://wiki.gentoo.org/wiki/Comparison_of_init_systems


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
3GB

46

Boot Sequence

- Yes, if you can get the bootstrap program to load something that is compromised… you have p0wned the system.
- The kernel, once it's bootstrapped, is in control and running in highest privilege mode. Eventually it's going to want to give control to a user process. That's no problem. It loads the user process, lowers the CPU privilege, and jumps to the user code.
- HOW DOES THE USER PROCESS GIVE CONTROL BACK TO THE KERNEL IN A WAY THAT RE-ELEVATES PRIV? WE BETTER NOT LET USER PROCESSES ELEVATE PRIVILEGE AND THEIR OWN ☺

University of CINCINNATI

47

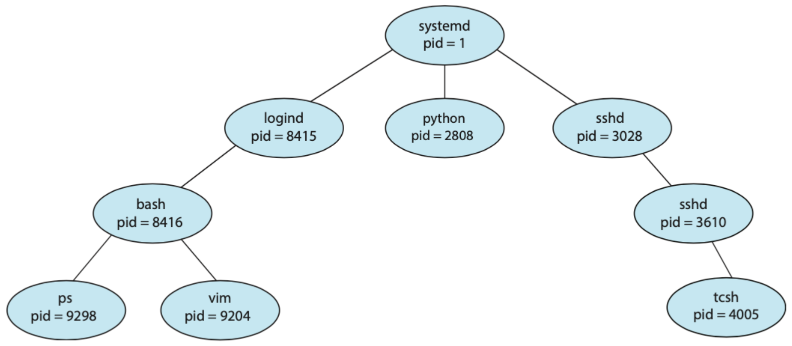


Figure 3.7 A tree of processes on a typical Linux system.

University of CINCINNATI

48

So, Where Do We Need to Go?

- Our next steps are:
  - To understand how we can even get this "kernel" and "OS" loaded onto the machine and get it running so it's there to manage things. This is NOT an unsubtle thing to do. Getting this wrong mean… no security
  - To understand the basic model of a process. Once the kernel is running, what exactly is it managing? Well… processes, which are collections of code and runtime state

University of CINCINNATI

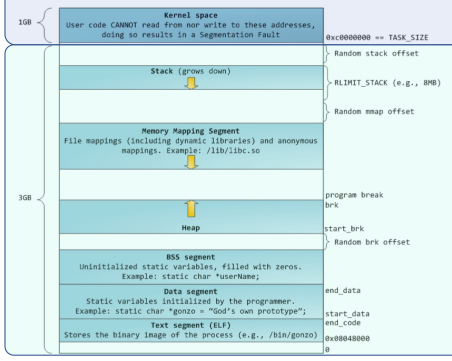
49

SPARE SLIDES WE MAY OR MAY NOT USE FOLLOW THIS SLIDE

University of CINCINNATI

50

What is a process – Memory View


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
3GB
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Virtual Addresses == Physical Addresses

Virtual Addresses != Physical Addresses

Virtual Addresses are References into Virtual Memory System

The PROCESS sees the virtual addresses and believes them to be real. Behind the scenes, the kernel and memory mapping hardware in the CPU are on the fly translating PROCESS virtual addresses into references to physical memory. The kernel may also be moving process memory to long term slow storage, bringing it in only if needed. We will cover this extensively in a later unit. For now we'll just note it is going on.

University of CINCINNATI

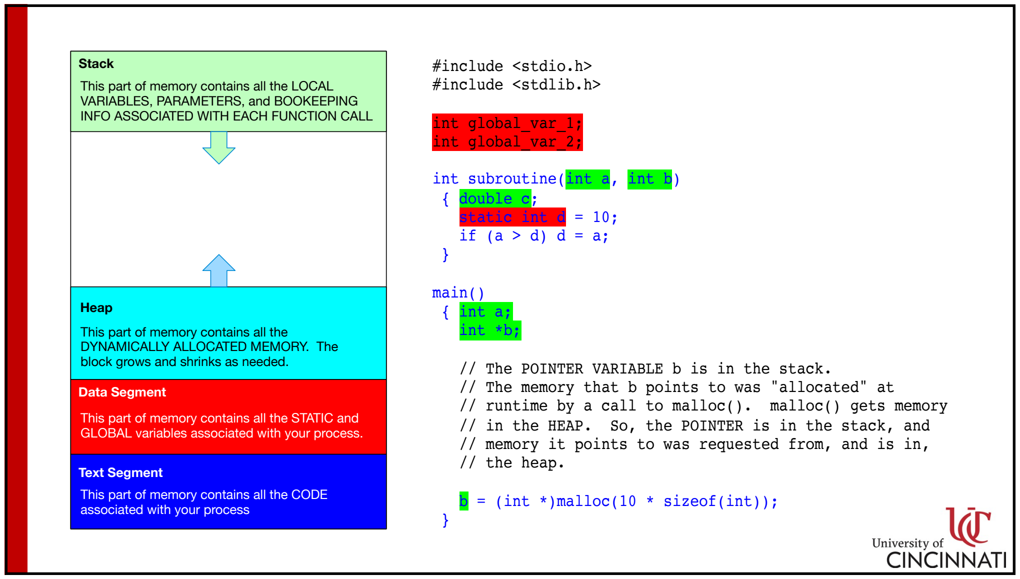
51

	x6FEF		
	x6FF0		
R6 --->	x6FF1	x????	e
	x6FF2	x????	arr[0]
	x6FF3	x????	arr[1]
	x6FF4	x????	arr[2]
	x6FF5	x????	arr[3]
R5 --->	x6FF6	x????	d
	x6FF7	x6fff	dynamic link
	x6FF8	x????	return address from mysub
	x6FF9	x????	return value from mysub
	x6FFA	x000c	w
	x6FFB	xffff	x
	x6FFC	x6ffe	y
	x6FFD	xffff	•
	x6FFE	x001d	•
	x6FFF	x000c	•
	x7000		

```
int main()
{ int a = 12, b = 29, c = -1;
  c = mysub(a, c, &b);
  ... etc ...
}

int mysub(int w, int x, int *y)
{ int d; int arr[4]; int e;
  ... more code ...
  arr[d] = *y;
  return d;
}
```

University of CINCINNATI

52



Stack

This part of memory contains all the LOCAL VARIABLES, PARAMETERS, and BOOKEEPING INFO ASSOCIATED WITH EACH FUNCTION CALL

Heap

This part of memory contains all the DYNAMICALLY ALLOCATED MEMORY. The block grows and shrinks as needed.

Data Segment

This part of memory contains all the STATIC and GLOBAL variables associated with your process.

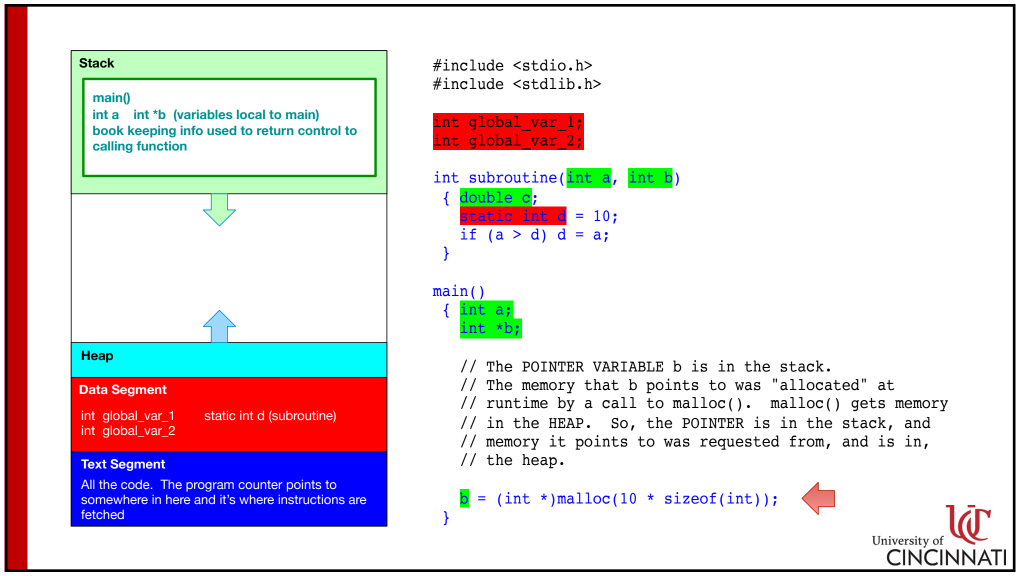
Text Segment

This part of memory contains all the CODE associated with your process

```
#include <stdio.h>
#include <stdlib.h>

int global_var_1;
int global_var_2;

int subroutine(int a, int b)
 { double c;
   static int d = 10;
   if (a > d) d = a;
 }

main()
 { int a;
   int *b;

   // The POINTER VARIABLE b is in the stack.
   // The memory that b points to was "allocated" at
   // runtime by a call to malloc().  malloc() gets memory
   // in the HEAP.  So, the POINTER is in the stack, and
   // memory it points to was requested from, and is in,
   // the heap.

   b = (int *)malloc(10 * sizeof(int));
 }
```

University of CINCINNATI

53



Stack

main()
int a   int *b (variables local to main)
book keeping info used to return control to calling function

Heap

Data Segment

int global_var_1   static int d (subroutine)
int global_var_2

Text Segment

All the code. The program counter points to somewhere in here and it's where instructions are fetched

```
#include <stdio.h>
#include <stdlib.h>

int global_var_1;
int global_var_2;

int subroutine(int a, int b)
 { double c;
   static int d = 10;
   if (a > d) d = a;
 }

main()
 { int a;
   int *b;

   // The POINTER VARIABLE b is in the stack.
   // The memory that b points to was "allocated" at
   // runtime by a call to malloc().  malloc() gets memory
   // in the HEAP.  So, the POINTER is in the stack, and
   // memory it points to was requested from, and is in,
   // the heap.

   b = (int *)malloc(10 * sizeof(int));
 }
```

University of CINCINNATI

54

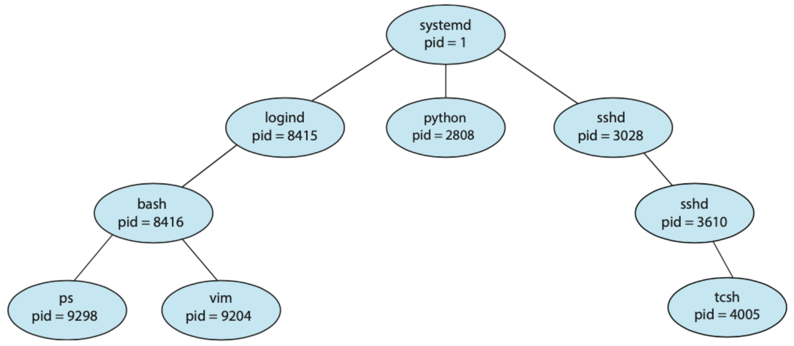
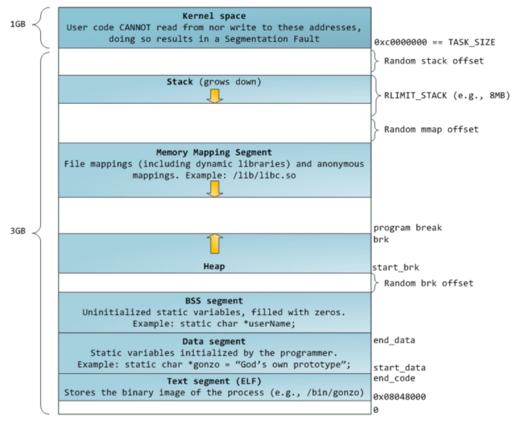


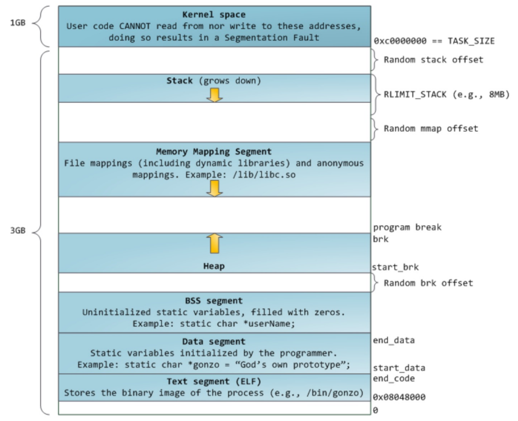
Figure 3.7 A tree of processes on a typical Linux system.

55


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
end_data
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Process memory map for a 32-bit Linux Kernel. *Note there's virtual memory games going on here…* which we WILL talk about very soon. For now the salient point is that the "kernel space" is the place where the kernel lives and ALL PROCESSES "see" the kernel in exactly that spot and with those addresses. No matter what process you are, the first 1GB IS THE KERNEL in its entirety.

56


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

The kernel space contains the TRAP table, all trap servicing routines, all data structures needed by the kernel to keep track of user "processes", and all other kernel code. It is always in memory at the actual physical addresses shown in the table (other addresses are VM games)

User code "goes into kernel" by calling a system call that maps to some trap.

University of CINCINNATI

57