CSE 506: Operating Systems

What Software Expects of the OS

What Software Expects of the OS

- Shell
- Memory Address Space for Process
- System Calls
- System Services
- Launching “Program” Executables

Shell

- Gives user ability to interact with machine
  - Can be text or graphical
- Traditionally text-based
  - Two families – ***sh*** (bash, zsh, ksh) and ***csh*** (tcsh)
- Interprets commands, one by one
  - Commands are either ***shell built-ins*** or ***executables***
  - Shells include many user-friendly features
    - PATH env variable, tab completion, …
- Systems start with ***/etc/rc*** ("run commands")
  - Starts with #!/bin/sh

What Software Expects of the OS

- Shell
- Memory Address Space for Process
- System Calls
- System Services
- Launching "Program" Executables

Memory Abstraction

- OS provides memory space to application
  - Application observes a contiguous "private" memory space
  - OS prevents "illegal" actions (e.g., no-exec, read-only)

- Memory typically includes several "sections"

  .text – program area

  .rodata – read-only variables

  .data – variables that have an initial value

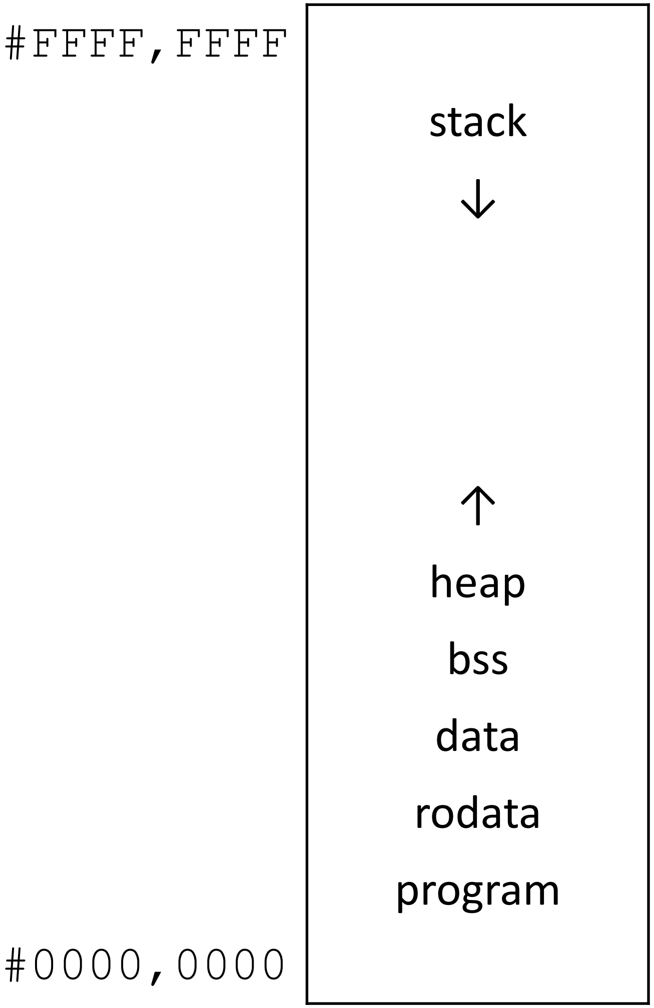
  .bss – variables that are initially zero

  heap

  stack

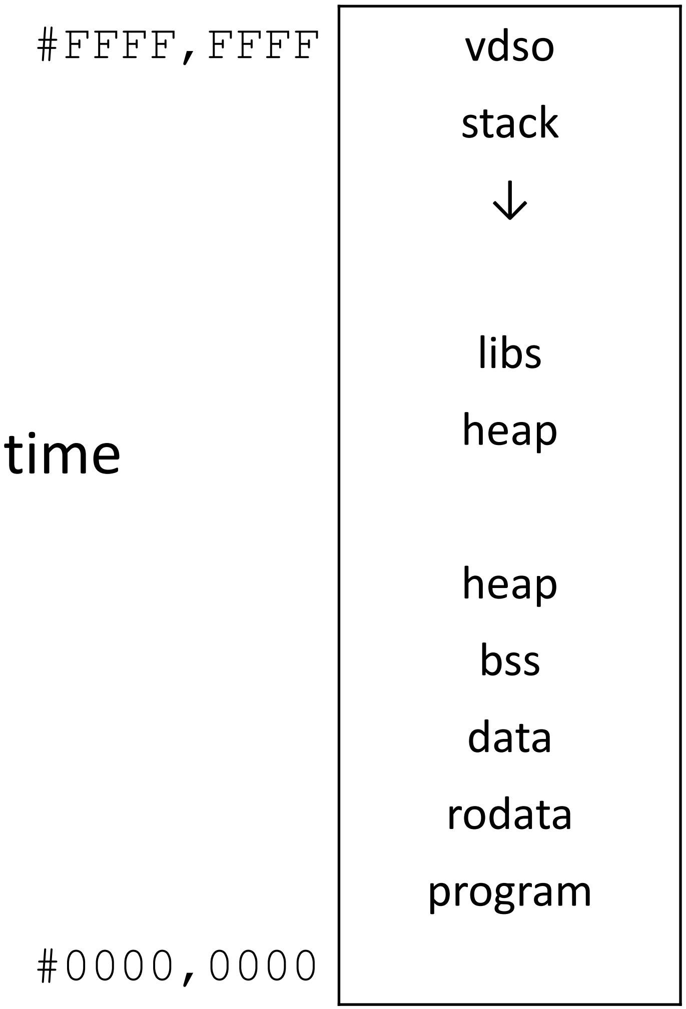
Traditional Memory View

- Don’t use addrs. close to 0
  - Allows to detect bad accesses
- Heap grows upward
  - Increases when app asks for mem.
- Stack grows ***downward***
  - Function calls push return addr.
  - Local variables go on stack
    - Main source of stack smash attacks
  - Must reserve stack space
    - Ensure that heap doesn’t hit stack



Modern Memory View

- Heap no longer allocated up
  - Although it still can be
- Shared libraries appear
- [vdso] adds magic memory
  - Example: always contains current time



What Software Expects of the OS

- Shell
- Memory Address Space for Process
- System Calls
- System Services
- Launching “Program” Executables

System Calls

- Mechanism for app to interact with the OS
  - Similar to function calls
  - Code securely implemented in the OS
  - Follows predefined interface
    - Called “ABI” – Application Binary Interface
    - Functions referenced by predefined number
- Example syscall triggers
  - “trap” instruction
  - Special “syscall” instruction

Example System Calls

- getpid()
  - Return process's ID
  - Function 39 in 64-bit Linux, 20 in FreeBSD
- brk()
  - Return current "top" of heap
  - Function 12 in 64-bit Linux, 69 in FreeBSD

Linux has ~650, FreeBSD ~400 syscalls

What Software Expects of the OS

- Shell,
- Memory Address Space for Process
- System Calls
- System Services
- Launching "Program" Executables

Services Provided by OS

- Usually through syscalls
  - Variants: vdso provides time of day, stack grows on faults
- Typical "important" services
  - Scheduler
  - Memory management
  - Threads
  - Terminal
  - File system
  - Pipes
  - Network
  - Limits

Numerous services exist; we'll stick to the above.

Process Control and Threads (1/2)

- Create and control processes and threads
  - Same syscall for both in Linux: ***clone()***
  - FreeBSD uses ***fork()*** for procs and ***thr_create()*** for threads
- Difference between processes and threads?
  - Fundamentally similar, separate “threads” of control
  - Threads share same memory space
    - But have their own stack pointers
  - All threads should share one PID, but have own TIDs
- Exert control over processes
  - ***kill()/signal()*** to KILL, TERMinate, STOP, CONTinue

Process Control and Threads (2/2)

- Create and control processes and threads
  - clone() in Linux, fork() in FreeBSD
- Creates an identical copy of the process:

```
int pid = fork();
if (pid == 0) {
   // child code
} else if (pid > 0) {
   // parent code
} else {
  // error (pid == -1)
}
```

Scheduler

- If there are multiple processes
  - Something has to decide what should run
- Takes into account many parameters
  - Readiness to run, priority, history
- Can be invoked by various triggers
  - On syscall – called ***cooperative*** multi-tasking
    - Low overhead
    - What happens when there are no system calls?
  - On timer – called ***preemptive*** multi-tasking
    - Processes can get de-scheduled at any time
    - Can still be slightly cooperative by calling ***yield()*** syscall

Memory Management

- Each process sees its own memory space
  - Called ***virtual*** memory
- Computer has DRAM chips plugged into it
  - Called ***physical*** memory
- OS manages physical memory
  - Operates on contiguous ***pages*** of memory (typically 4KB)
  - Maintains a virtual-to-physical mapping
    - Physical pages are allocated on demand
  - Supports ***paging*** (saving physical page contents to disk)
    - Sometimes used interchangeably with ***swapping*** (entire apps)

Memory Management

- Process starts with some memory
  – text, data, stack, heap
- Stack grows automatically
  – On an access below the stack
  – Allocate up to and including the demanded page
- Heap grows on request
  – Traditionally, ***brk(new_value)***
  – Modern systems use ***mmap()***
  – ***malloc()*** uses one or the other
    - Implementations rarely release brk() memory back to the OS

Terminal (1/3)

- Not the same as ***console***
  - Although ***console*** is usually connected to a ***terminal***
- Terminals have two ends
  - One connects to an input/output device
    - Teletype, serial port, console/screen and keyboard
  - Other end attached to software (e.g., bash)
  - Anything written into one end comes out the other
    - Extremely convenient for such a simple interface
- Provide input discipline
  - Buffers input until newline
- Handles necessities like ***local echo***

Terminal (2/3)

- Formatting done via ***escape sequences***
  - Sequences of characters control output behavior
  - Example: vt100 family sets red color with: ESC[31m
- Input also has a level of processing
  - Printable characters pass as-is
  - Modifiers (e.g., Control) used for additional control
    - "H" is ASCII 40, "Ctrl+H" is ASCII 8
      - Backspace key is typically just ASCII 8
    - "C" is ASCII 35, "Ctrl+C" is ASCII 3
      - Terminal sends SIGINT to ***foreground*** process when receiving char 3
  - Implements ***type-ahead***, buffers chars until they are read

Terminal (3/3)

- Most systems today use ***pseudo terminals***
  - No physical hardware attached for I/O
    - Simulated with network (e.g., ssh) or graphical widow (e.g., xterm)
  - Arranged as pair of devices in OS
    - Traditional software end is ***slave***
    - Traditional teletype is ***master***
    - Things written to slave end come out of master and vice versa
- Terminals are a fundamental part of the OS
  - Sadly, many people consider them archaic and legacy
  - In truth, a necessary and major modern component

File System

- Provides access to data
  - open/creat, read, write, seek, close
  - opendir, readdir, closedir, mkdir, unlink
  - mmap (interface combines memory and files)
- Organized as mount points
  - Each mount point is a directory in the parent system
  - “root” mount point always at the top
- OS maintains a ***descriptor table*** for each open file
  - Returned by open()
  - Used by all subsequent operations

Everyone here ***must*** already be familiar with this

Pipes

- Primitive (but helpful) inter-process communication
  - Enables one process to communicate to another
  - Uni-directional (has distinct writer and reader ends)
- Uses same interface as files
  - OS maintains descriptors in same table as regular files
  - created with pipe() call (or open() on “fifo” file type)
  - write/read done just like on files
- Kernel maintains a small buffer as temp storage
  - Avoids switching between processes after every byte

Everyone here *must* already be familiar with this

Network

- Enables communication between processes
  - Can be even on same machine (e.g., “localhost”)
- Dominated by IPv4 today
- Common operations
  - Assign address to an interface
  - Manipulate routing table
  - Make outgoing connections, receive incoming connections
  - Send data, receive data
- Uses same descriptor table as files
  - Called ***socket descriptors*** for network

Everyone should already be familiar with this

Limits

- OS protects processes from each other
  - And processes from themselves
- Parent process limits are inherited by child process
- Process can set its own limits
  - Can set ***hard*** limits lower or equal to existing ones
    - Can only reduce, can never increase
  - Can set ***soft*** limits lower or equal to hard ones
    - These are the actual limits enforced by the OS
- Examples:
  - Max memory, max stack size, max open files
  - Some limits are ***per user*** – e.g., number of processes

What Software Expects of the OS

- Shell
- Memory Address Space for Process
- System Calls
- System Services
- Launching “Program” Executables

Launching Program Executables (1/2)

- Roughly a 3-step process
  - Load initial contents into memory
  - Find starting point (usually function called ***_start()***)
  - Set initial registers (stack pointer, program counter)
- How to load program into memory?
  - Dictated by ***binary format***
    - Most systems today use ELF or PE
  - Defines parts of the file to load and where to load tem
    - Broken up into sections
      - Offset (in the file), length, destination address, and size
      - Length can be smaller than size – indicates zero pad

Launching Program Executables (2/2)

- Programs are launched using ***execve*** syscall
- First bytes determine binary format
  - 0x7F E L F : ELF binary
  - #!command : (***shebang***) interpret with (command)
- An “interpreter” may run instead of the program
  - Program becomes first argument to interpreter
  - Interpreter path supported in ELF binaries
    - Critical for ***shared libraries***
    - Interpreter is set to /lib/ld.so for instance
      - Running “/bin/ls /” is equivalent to “/libexec/ld-elf.so /bin/ls /”