Linux overview

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Summary

- Hypervisors
  - Type 1 vs. type 2
  - Advantages
- OS-level / process-level virtualization
  - Container abstraction
  - Light virtualization
  - Performance, isolation, security
- Trusted execution environments

Reading material

- File system: Ch. 37, 39-42 of OSTEP
  - http://pages.cs.wisc.edu/~remzi/OSTEP/
- VMMs: Ch. “VMM intro” of OSTEP
  - http://pages.cs.wisc.edu/~remzi/OSTEP/vmm-intro.pdf
  - “Hardware and Software Support for Virtualization”
    Edouard Bugnion, Jason Nieh, Dan Tsafrir. 2017
- Containers:
  - Understanding and Hardening Linux Containers (mainly Ch. 1, 2, 9 and 11)
    https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-1-1.pdf
- SGX:
  - https://blog.quarkslab.com/overview-of-intel-sgx-part-1-sgx-internals.html

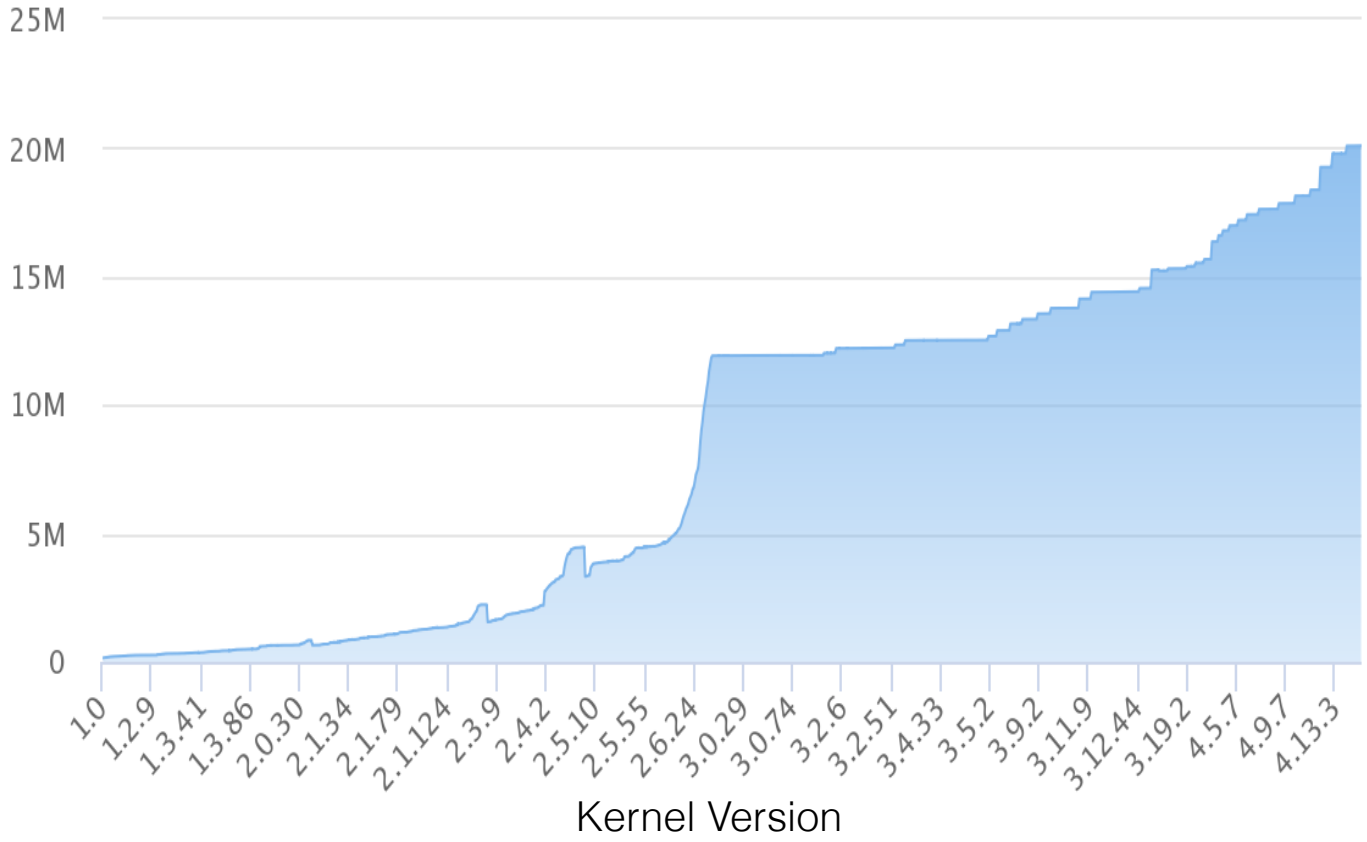
History

- Version 0.01 released in May 1991
  - Initially only supported PCs
  - Added support for architectures other than the PC in 1995
  - Currently supports **25** architectures
- Latest stable version: 5.0.7
  - https://www.kernel.org/
- Linux distributions: basic Linux system with system installation and management utilities, and ready-to-install packages of common UNIX tools
- GPL license

Design principles

- Linux is multiuser and multitasking
- Designed to be compatible with POSIX standards
- Main design goals are speed, efficiency, standardization

Scale



+15M
lines of code in the last 15 years

+67k files, +900 MB of source code data

More drivers, more architectures, more functions, more optimizations

(Xinu: ~70k lines)

Linux system

- Kernel:
  - Full access to all physical resources of the computer
    - Any exceptions?
  - Uses a single address space for both kernel code and kernel data structures
- System libraries:
  - Define a standard set of functions through which applications interact with the kernel
- System utilities:
  - Performance management tasks

Kernel code and kernel modules

- Kernel modules: part of the kernel code can be compiled, loaded and unloaded during system execution (i.e., without restarting)
- Q: What is the use of modules? How to implement this?
- Device drivers are often configured as a modules
- Support for registering new drivers

Module management

- Load and unload modules
- Loading requires mechanism:
  - Put module code in kernel memory
  - Link kernel symbols with module code
- Unloads modules requires checking whether it is in use
- Modules have special routines that are called when loading and unloading
- Uses a driver registration mechanism (e.g., file system, device driver, etc.)

Other aspects

- Window manager is outside of the kernel
- Extensive use of macros
- Dependent on the compiler
  - Even requires compiler optimizations to build

Does this code build with gcc?

```
void f();

int main() {
    int x = 0, *y;
    y = &x;

    if (*y)
        f();
    return 0;
}
```

Does this code build with gcc?

```
$ gcc -o deadcode  deadcode.c
Undefined symbols for architecture x86_64:
  "_f", referenced from:
      _main in deadcode-7cb7db.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Without optimizations it is not able to link
(f function is not defined)

```
$ gcc -o deadcode -O1 deadcode.c
```

With optimization it compiles and links successfully
because because compiler does a dead code analysis

```
void f();

int main() {
    int x = 0, *y;
    y = &x;

    if (*y)
        f();
    return 0;
}
```

Process management

- Fork() and execve()
- Process properties:
  - Process identity: pid, credentials (user id, group id)
  - Environment: argument vector and environment vector
  - Context: scheduling context, accounting, file table

Environment variables

- Environment vector/variables can be used to customize system on a per-process basis
- They are inherited by processes

```
$ export | grep -i editor                    (checking from bash shell)
declare -x EDITOR=“vim"

$ export EDITOR=emacs                        (setting from bash shell)
```

Processes and threads

- Clone() creates a new process with its own identity but that is allowed to share the data structures of its parent
- pthread library:
  - Management of threads
  - Coordination mechanisms

Useful tools

- gdb
- strace
- Info in /proc/ directory

strace output example

```
$ strace ls -l /boot | head
execve("/bin/ls", ["ls", "-l", "/boot"], [/* 65 vars */]) = 0
brk(NULL)                               = 0x620000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=186081, ...}) = 0
mmap(NULL, 186081, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ffff7fca000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260Z\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=130224, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ffff7fc9000
mmap(NULL, 2234080, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ffff7bb5000
```

strace

- Runs another application and records the system calls invoked
  - Also records arguments and return values
- Can be attached to a running process as well
- Q: How to implement such mechanism in Xinu?
- In Linux implemented with **ptrace** system call
  - http://man7.org/linux/man-pages/man2/ptrace.2.html

Gdb

- Basic mechanism:
  - Catch the exec syscall and block the start of the execution
  - Query the CPU registers to get the process's current instruction and stack location
  - Catch for clone/fork events to detect new threads
  - Peek and poke data addresses to read and alter memory variables
- Breakpoints can be created by modifying the code (e.g., writing an invalid instruction) and catching the trap
- How gdb works:

Gdb breakpoints

- *The debugger reads (ptrace peek) the binary instruction stored at this address, and saves it in its data structures.*
- *It writes an invalid instruction at this location. What ever this instruction, it just has to be invalid.*
- *When the debuggee reaches this invalid instruction (or, put more correctly, the processor, setup with the debuggee memory context), the it won't be able to execute it (because it's invalid).*
- *In modern multitask OSes, an invalid instruction doesn't crash the whole system, but it gives the control back to the OS kernel, by raising an interruption (or a fault).*
- *This interruption is translated by Linux into a SIGTRAP signal, and transmitted to the process ... or to it's parent, as the debugger asked for.*
- *The debugger gets the information about the signal, and checks the value of the debuggee's instruction pointer (i.e., where the trap occurred). If the IP address is in its breakpoint list, that means it's a debugger breakpoint (otherwise, it's a fault in the process, just pass the signal and let it crash).*
- *Now that the debuggee is stopped at the breakpoint, the debugger can let its user do what ever s/he wants, until it's time to continue the execution.*
- *To continue, the debugger needs to 1/ write the correct instruction back in the debuggee's memory, 2/ single-step it (continue the execution for one CPU instruction, with ptrace single-step) and 3/ write the invalid instruction back (so that the execution can stop again next time). And 4/, let the execution flow normally.*

https://blog.0x972.info/?d=2014/11/13/10/40/50-how-does-a-debugger-work

/proc/<pid>

/proc/PID/cmdline : Command line arguments.
/proc/PID/cpu : Current and last cpu in which it was executed.
/proc/PID/cwd : Link to the current working directory.
/proc/PID/environ : Values of environment variables.
/proc/PID/exe : Link to the executable of this process.
/proc/PID/fd : Directory, which contains all file descriptors.
/proc/PID/maps : Memory maps to executables and library files.
/proc/PID/mem : Memory held by this process.
/proc/PID/root : Link to the root directory of this process.
/proc/PID/stat : Process status.
/proc/PID/statm : Process memory status information.
/proc/PID/status : Process status in human readable form.

… and more

More questions

- How to build defense mechanisms against kernel bugs?

Summary

- History of Linux
- Scale
- Kernel structure
- Process management
- Debugging tools:
  - Debugger, strace, /proc/