The LINUX Operating System

A Brief on it's Architecture

Important Notice : Courseware - Legal

This courseware is both the product of the author and of freely available opensource and/or public domain materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of the **courseware PDFs are considered proprietary** and thus cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - **source code** and binaries (where applicable) - that form part of this courseware, and that are made available to the participant, are released under the terms of the permissive **MIT license** [1].
Under the terms of the MIT License, you can certainly use the source code provided here; you must just attribute the original source (author of this courseware and/or other copyright/trademark holders).

VERY IMPORTANT :: Before using this source(s) in your project(s), you *MUST* check with your organization's legal staff that it is appropriate to do so.

The courseware PDFs are *not* under the MIT License, they are to be kept confidential, non-distributable without consent, for your private internal use only.

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant VM are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind, and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

2000-2024 Kaiwan N Billimoria
kaiwanTECH, Bangalore, India.

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: https://bit.ly/ktcorp

Table of Contents

A few ‘golden rules’

1.

empirical

/ɛmˈpɪrɪk(ə)l,ɪmˈpɪrɪk(ə)l/

based on, concerned with, or verifiable by observation or experience rather than theory or pure logic.

Be empirical

2.

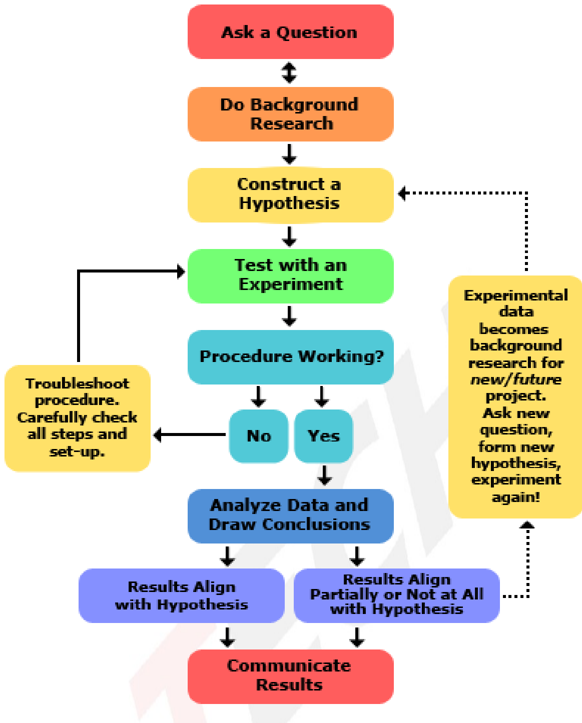
Don’t Assume

To ASSUME == to make an ASS out of U and ME :-)

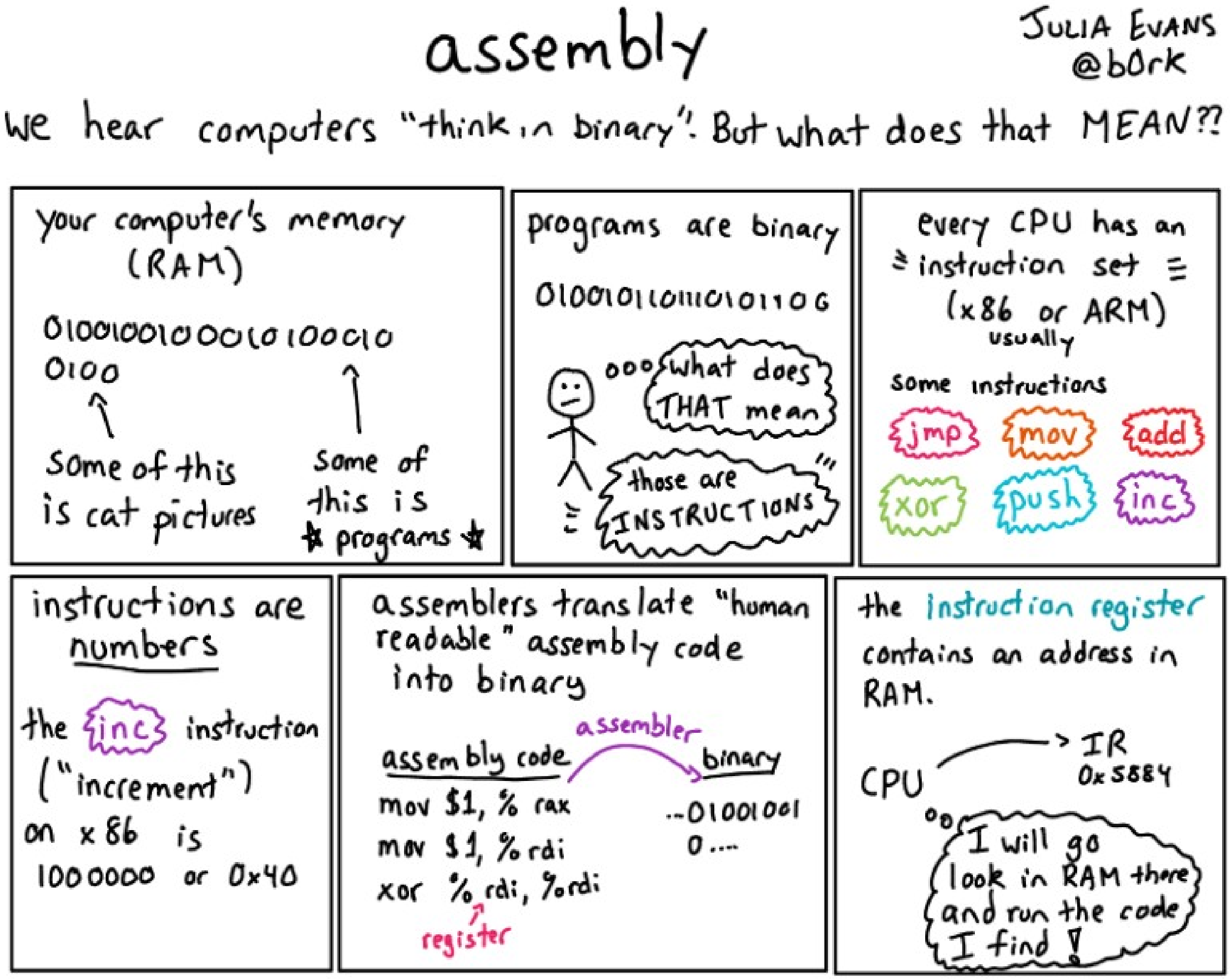
3.

The steps in the Scientific Method

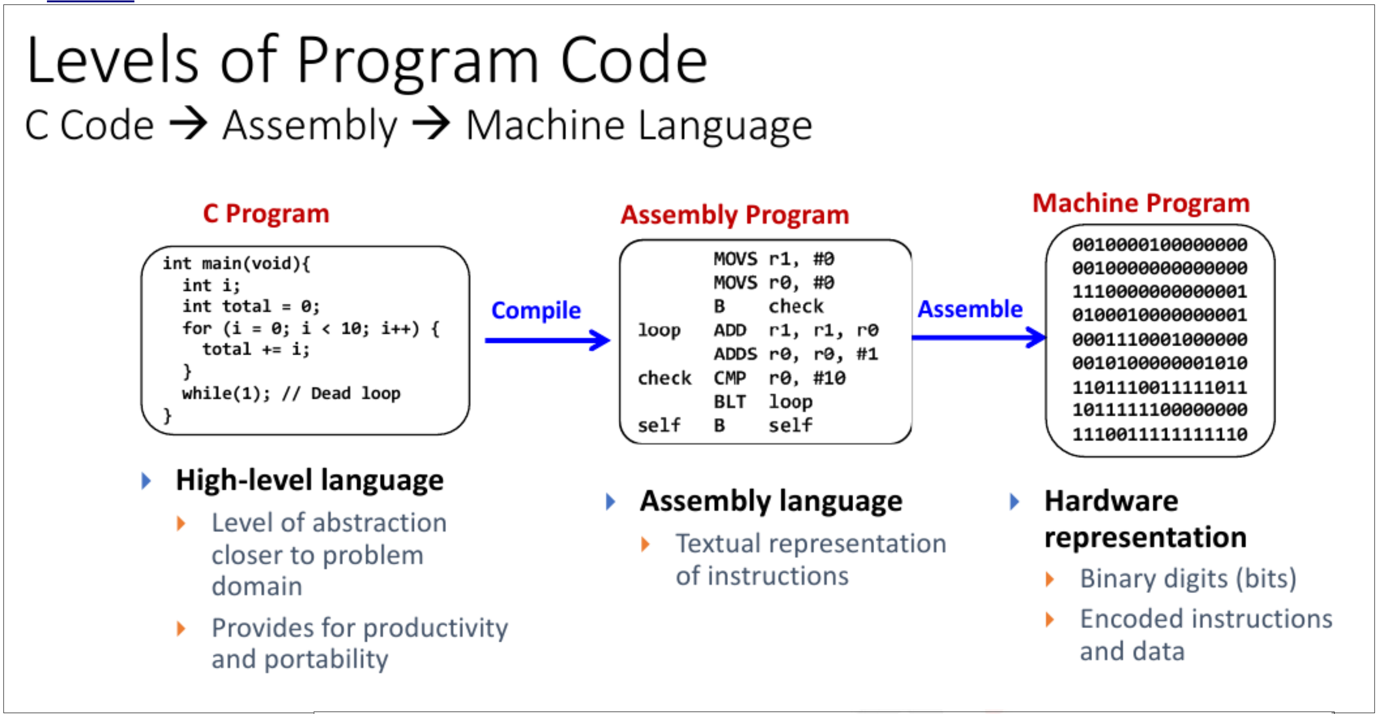
Preliminaries

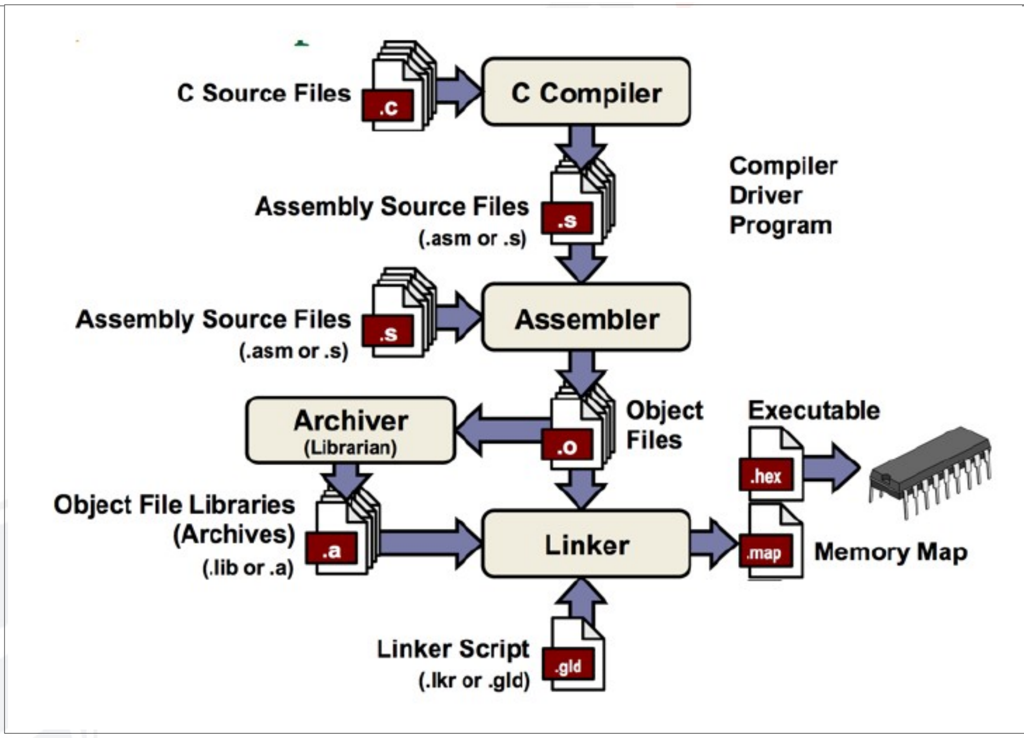


Ref: https://drawings.jvns.ca/assembly/



[Source](#)





You can see the ‘toolchain’ above…

Experiment:
See what exactly runs via the system native toolchain – on x86_64 Linux (Ubuntu 23.10) – when we compile the classic ‘hello, world’ C program:

```
$ cat helloworld.c
#include <stdio.h>
int main()
{
        printf("hello, world\n");
}
$ gcc helloworld.c -o helloworld
```

I ‘traced’ it via the powerful **eBPF execsnoop** utility; here’s the result:

```
Timestamp  UID   Command-name        PID    PPID   RET  ARGS
16:15:01 1000  grep             65699  23702    0 /usr/bin/grep -q GCC
16:15:01 1000  cc               65698  23702    0 /usr/bin/cc --version
16:15:01 1000  realpath         65702  65700    0 /usr/bin/realpath /usr/bin/cc
16:15:01 1000  grep             65704  23702    0 /usr/bin/grep -q GCC
16:15:01 1000  c++              65703  23702    0 /usr/bin/c++ --version
16:15:01 1000  realpath         65707  65705    0 /usr/bin/realpath /usr/bin/c++
16:15:01 1000  grep             65709  23702    0 /usr/bin/grep -q GCC
16:15:01 1000  command-not-fou  65710  65708    0 /usr/lib/command-not-found -- f77
16:15:01 1000  snap             65711  65710    0 /usr/bin/snap advise-snap --
format=json --command f77
16:15:01 1000  snap             65711  65710    0 /snap/snapd/current/usr/bin/snap
advise-snap --format=json --command f77
16:15:01 1000  grep             65724  23702    0 /usr/bin/grep -q GCC
16:15:01 1000  command-not-fou  65725  65723    0 /usr/lib/command-not-found -- f95
16:15:01 1000  snap             65726  65725    0 /usr/bin/snap advise-snap --
format=json --command f95
16:15:01 1000  snap             65726  65725    0 /snap/snapd/current/usr/bin/snap
advise-snap --format=json --command f95
16:15:01 1000  gcc              65742  23702    0 /usr/bin/gcc --completion=
16:15:02 1000  gcc              65745  23702    0 /usr/bin/gcc --completion=
16:15:05 1000  gcc              65748  23702    0 /usr/bin/gcc --completion=
16:15:06 1000  gcc              65752  23702    0 /usr/bin/g
helloworld
16:15:06 1000  cc1              65753  65752    0
/usr/libexec/gcc/x86_64-linux-gnu/13/cc1 -quiet -imultiarch x86_64-linux-gnu
helloworld.c -quiet -dumpbase helloworld.c -dumpbase-ext .c -mtune=generic -march=x86-64
-fasynchronous-unwind-tables -fstack-protector-strong -Wformat -Wformat-security -
fstack-clash-protection -fcf-protection -o /tmp/ccCF3E1F.s
16:15:06 1000  as               65754  65752    0 /usr/bin/as        /ccAgU2aK.o
/tmp/ccCF3E1F.s
16:15:06 1000  collect2         65755  65752    0
/usr/libexec/gcc/x86_64-linux-gnu/13/collect2 -plugin /usr/libexec/gcc/x86_64-linux-
gnu/13/liblto_plugin.so -plugin-opt=/usr/lib
...
```

cc1: compiler (C src to assembly)

as: assembler (assembly to machine language)

collect2: the linker (link)

<<

FYI:

[*Linux x86 Program Start Up or - How the heck do we get to main()? by Patrick Horgan*]

[*crt0 – C runtime 0*]

>>

Linux / Unix Architecture

Source
"In the old days, you had a processor and it executed instructions. When an interrupt occured, the processor would save its current state and then branch to a specific place in order to service the interrupt. Thus, essentially, the processor had two 'modes' - dealing with an interrupt, and not.

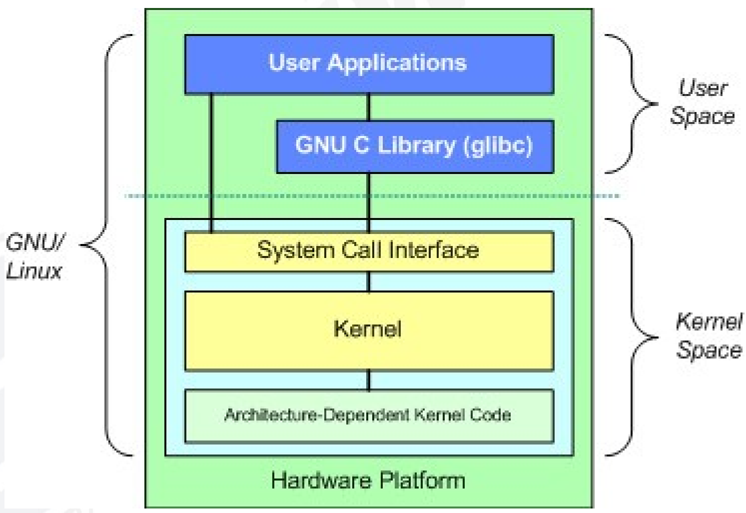
Fast forward a few decades, processors are much more capable and something of great importance is the ability to multitask, to run numerous programs at the same time. Technically this isn't possible, the processor has one data bus and one address bus, and however many cores are inside it, each core can only do one thing at a time. However by breaking a program's execution into tiny chunks, the processor can switch between them many times a second, providing the illusion that they are all running at the same time.

This brings with it a number of additional requirements, namely that

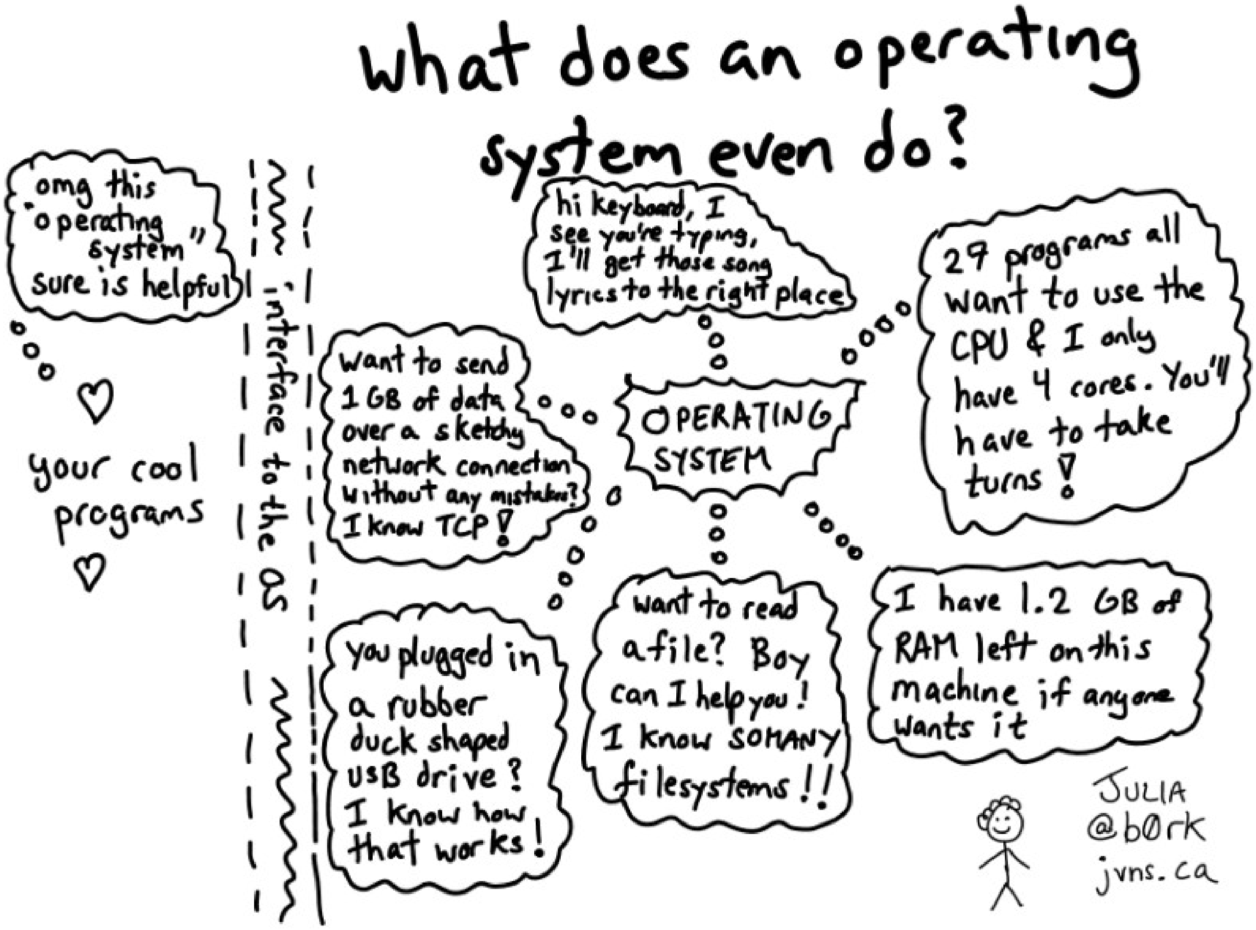
- one program should not be able to mess around with the memory used by another program
- furthermore, *none* of the mere programs should be able to mess around with the operating system or the machine's hardware

This is managed, in part, by the use of an MMU or other memory management system, and in part by the use of *privilege*. It is the processor mode that provides the desired level of privilege. ..."

Simplified System Architecture



Ref



Src

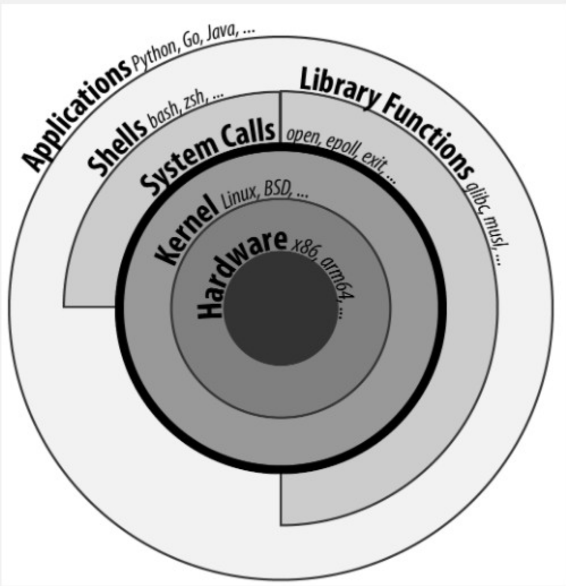
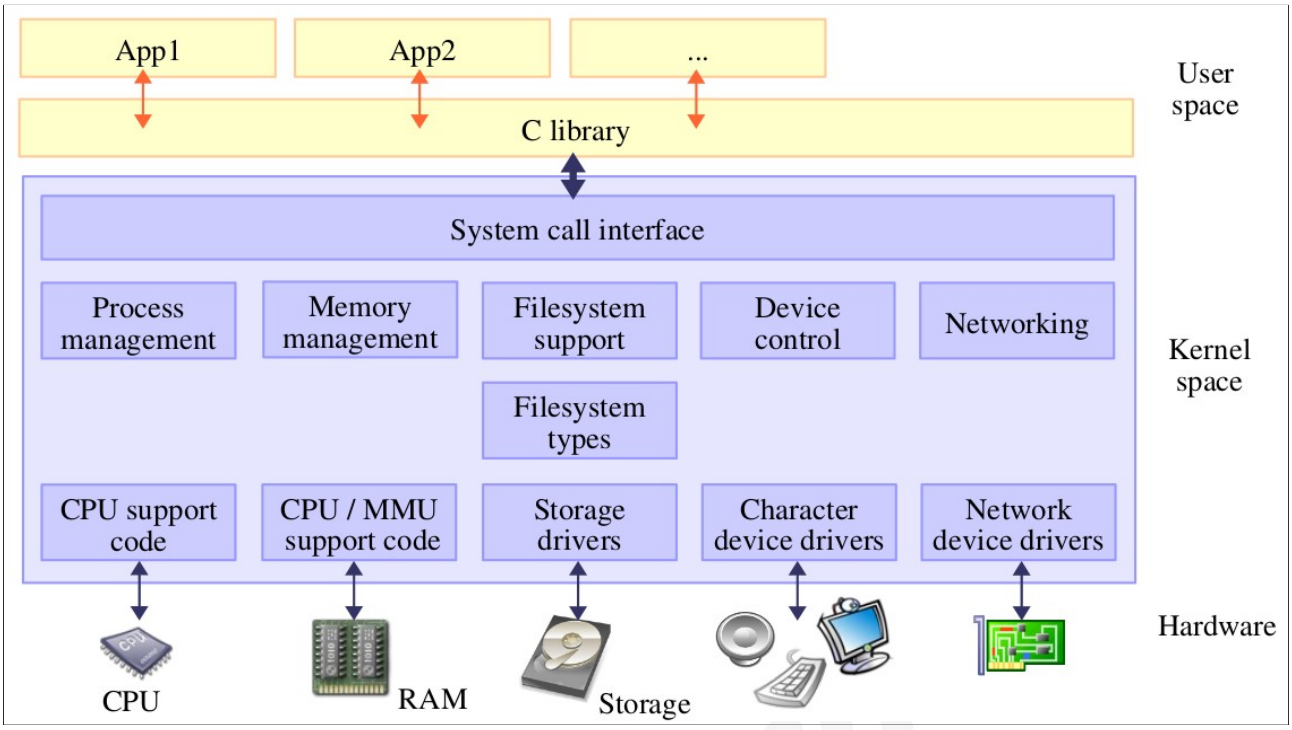


Figure 1. System calls are the border between your application and the Linux kernel

More Detailed System Architecture



<< Above Pic: © Copyright 2006 2004, Michael Opdenacker , © Copyright 2004 2008 Codefidence Ltd. >>

<<
Src: [*https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/KernelMechanics.html#mode-switches-and-privileged-instructions*](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/KernelMechanics.html#mode-switches-and-privileged-instructions)
<this portion is x86-biased>

A **mode switch** refers to a change in the CPL (CPU Privilege Level). System calls and interrupts both trigger a mode switch from ring 3 to ring 0. At the same time that the CPL changes, the %rip register is updated to begin reading from the kernel's code segment. The address loaded into the %rip is determined by a data structure that the kernel sets up during the boot process. In addition to updating the CPL and the %rip, the CPU makes a copy of the user-mode program status (such as its %rip value).

One important aspect to note about a mode switch is how quickly it occurs. Specifically, mode switches occur within a single execution of the von Neumann instruction cycle. Once the %rip has been updated at the end of one instruction's cycle, the CPU checks if an interrupt needs processing. If there is a pending interrupt, the CPU triggers a mode switch before fetching the next instruction; if not, then the next instruction is fetched.

After the system call or interrupt has been processed, the kernel forces a mode switch by executing the iret instruction. Just as ret updated the %rip to return to the portion of code that called a function, iret acts as a return from an interrupt to get back to the appropriate location in the user-mode program. The iret instruction restores the user-mode program's status that it had stored previously and lowers the CPL back to ring 3.

[1] More recent x86 processors have also added another bit to the CPL that is used by certain types of virtualization technologies. This additional bit is used to distinguish between "guest mode" and "host mode." In these types of systems, multiple guest *virtual machines* may be running as "guests" while a single *hypervisor* manages them as the "host." In these types of environments, ring 0 refers to kernel mode within a guest, whereas the hypervisor operates in "ring -1," which is kernel mode within the host.

>>

- A modern CPU has several **levels of privilege at which code is executed**
- Minimally, **two levels** of privilege
  - privileged mode : the kernel / OS + most drivers
  - unprivileged mode : apps (user processes/threads)

If not, a user app can do this:

```
__asm__("HLT");  // <--- DoS attack !
```

As an example, lookup the ISA – Instruction Set Architecture – manuals for the Intel 64 and IA-32 processors!
https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

Intel 64 and IA-32 ISA – Vol 2 PDF:
https://cdrdv2.intel.com/v1/dl/getContent/671110

Want to see all system calls available on Linux?

There are several ways to:

- man pages, section 2 ; *link*
- *type:* man 2 syscalls
- via code (headers)
- using software like the *auditd* package (below):

```
$ ausyscall --dump
Using x86_64 syscall table:
0     read
1     write
2     open
3     close
4     stat
...
...
446   landlock_restrict_self
447   memfd_secret
448   process_mrelease
449   futex_waitv
```

```
     $
```

<<

Using the K&R C ‘Hello, world’ program to understand the Linux architecture

Having seen the essential system architecture, let’s check out the K&R Hello, world C program:

```
#include <stdio.h>
int main()
{
     printf("hello, world\n");
}
```

Right; compile and run; it’s fine:

```
$ gcc -g helloworld.c
$ ./a.out
hello, world
$
```

(Why compile with debug enabled (-g)? It’s for the objdump below…):
But under the hood?

Viewing the source, assembly and machine code via objdump

```
$ objdump -dS --source-comment="< C code, assembly follows below
>" ./a.out

./a.out:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000: f3 0f 1e fa              endbr64

...
...

0000000000001169 <main>:
< C code, assembly follows below >#include <stdio.h>
< C code, assembly follows below >#include <unistd.h>
< C code, assembly follows below >int main()
```

C code

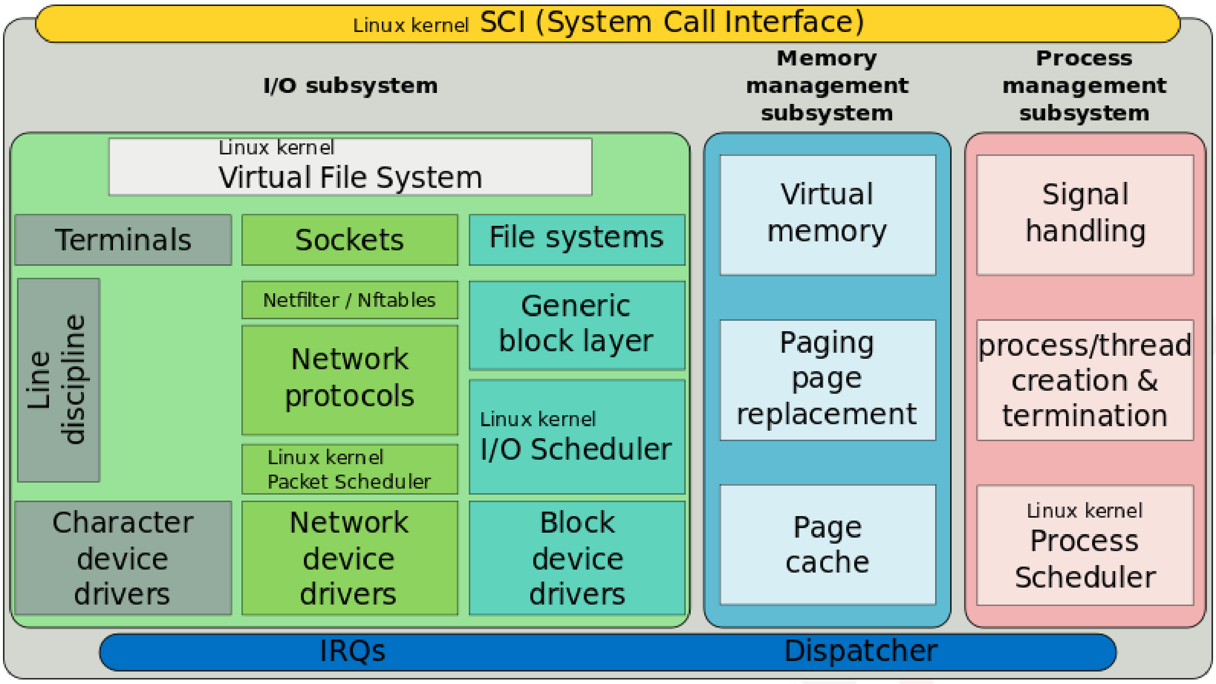
```
< C code, assembly follows below >{
    1169: f3 0f 1e fa                endbr64
    116d: 55                         push   %rbp
    116e: 48 89 e5                   mov    %rsp,%rbp
< C code, assembly follows below >printf("hello, world\n");
    1171: 48 8d 05 8c 0e 00 00       lea    0xe8c(%rip),%rax        #
2004 <_IO_stdin_used+0x4>
    1178: 48 89 c7                   mov    %rax,%rdi
    117b: e8 e0 fe ff ff             call   1060 <puts@plt>
< C code, assembly follows below >}
    118a: 5d                         pop    %rbp
    118b: c3                         ret
...

>>
```

Machine code

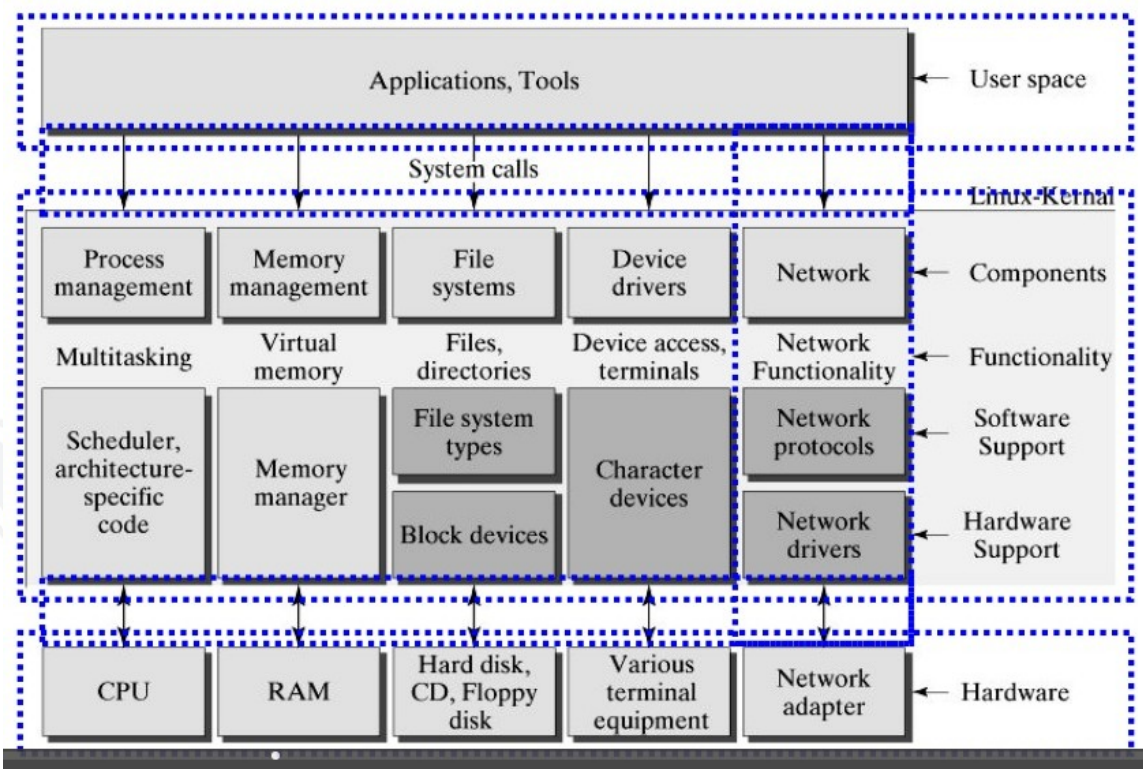
Assembly code

<<

[Simplified Structure of the Linux Kernel : Wikimedia](#)



Yet another architecture diagram [[Source](#)]:



>>

<<

1. *Lab exercise:* Write a C 'Hello, world' program without using any (g)libc routines (like printf() or puts())
2. *Lab exercise:* Write a 'Hello, world' program without using any libc routines nor any system calls (tip: it need not be written in C)

>>

System calls – major impact on performance !

System calls should be sparingly used; they're *much* slower to execute than user mode APIs. Why? As we have to cross the boundary from user → kernel mode and back again; it's a switch of context, it has definite overheads!

(A part) of *this superb article* - ***LINUX SYSTEM CALLS UNDER THE HOOD, Julien Sobczak, Aug 2021*** - nicely illustrates it:

…

```
 start_time = clock();
    for (int i=0; i<10000000; i++) {
        pid = getdummyid();
    }
    elapsed_time = (double)(clock() - start_time) / CLOCKS_PER_SEC;
    printf("Done getdummyid in %f seconds\n", elapsed_time);

    start_time = clock();
    for (int i=0; i<10000000; i++) {
        pid = getpid();
    }
    elapsed_time = (double)(clock() - start_time) / CLOCKS_PER_SEC;
    printf("Done getpid     in %f seconds\n", elapsed_time);
```

…

Results:

```
# ./benchmark
Done getdummyid in 0.022424 seconds
Done getpid     in 4.141334 seconds
```

Whoa!

"Calling a system call is, on this example, 200 times slower than calling a simple function. Indeed, a system call is not a simple function call. When you call the function `getpid()`, you use a wrapper implemented by glibc hiding the actual logic to execute a system call. Under the hood, this wrapper function does a lot of work …

[...]

System calls are essential for developers. They define the capabilities of your system. For example, the epoll system call helped Nginx to solve the C10k problem by offering a event-driven I/O model, the inotify_* system calls allows react-scripts to automatically rerun your tests when you are making a code change, the sendfile system call supports the Zero-Copy optimization used by Kafka, which is one of the main reasons explaining its performance, the ptrace system call is used by debugguers like gdb to inspect your program using breakpoints, and so on. …"

Execution Privilege Levels in Different CPU Architectures

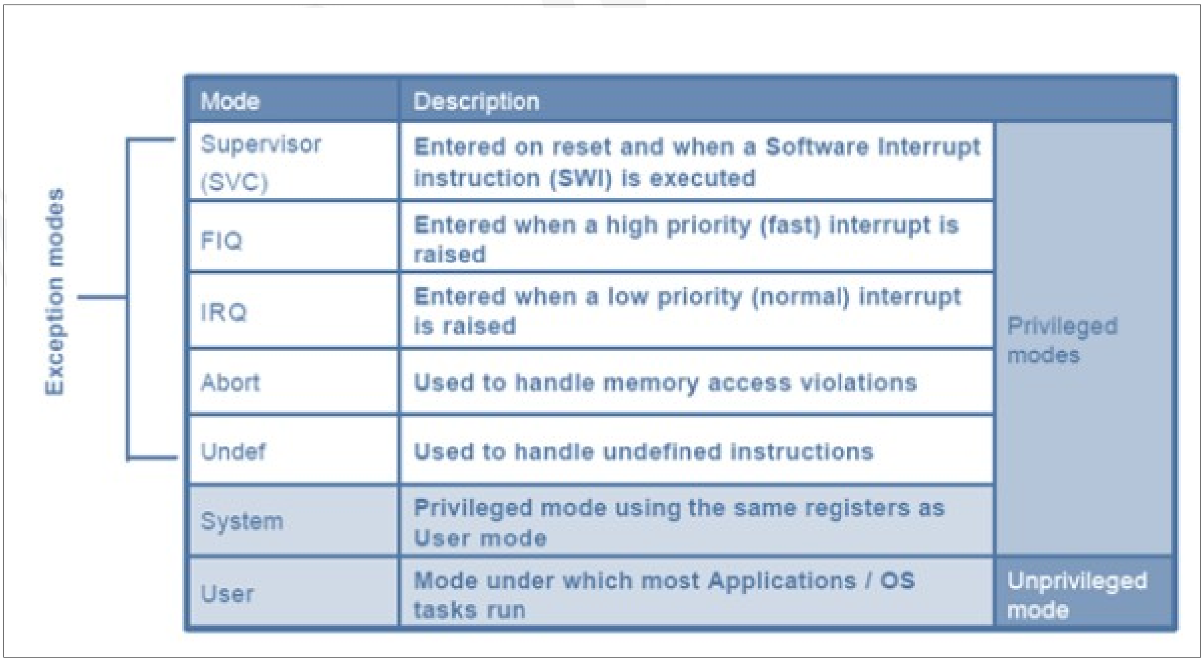
x86

Discussion

- The SCI – System Call Interface – Layer
- CPU Privilege Levels
  - User Mode
  - Kernel (Supervisor) Mode
- *Intel/AMD: Ring 3 => User, Ring 0 => OS*

ARM (AArch32) CPU Modes

7 modes, 6 of which are privileged



Mode	Description	
Supervisor (SVC)	Entered on reset and when a Software Interrupt instruction (SWI) is executed	Privileged modes
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a low priority (normal) interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode	
User	Mode under which most Applications / OS tasks run	Unprivileged mode

AArch64 (ARM-64 / ARMv8)

- Modes replaced by processor ***Exception Levels (ELs)***

- Four ELs *(lowest to highest privilege)*
  - **EL0** **:** Normal user applications
  - **EL1** **:** Operating System (OS) kernel typically described as *privileged*
  - **EL2** **:** Hypervisor [optional]
  - **EL3** **:** Low-level firmware, including the Secure Monitor

- Here ELs determine the CPU privilege level (just as Modes do in Aarch32)

- Exception: in-kernel hypervisors (such as KVM) – operate across both EL1 and EL2

- The Aarch64 will also of course perform **exception handling** (just as with the Aarch32) – via interrupts, aborts, and synchronous exception-raising instructions (SVC, HVC, etc.); refer AArch64 Exception Handling on the ARM developer site.

Arch-specific - issuing of system calls

Architecture	*Machine Instruction(s)*	*Syscall # Register*
Intel x86[_64]	int $0x80 | syscall	EAX / RAX
ARM	SWI	R7
ARM64	SVC	X8
MIPS	syscall	$v0

Details ***[FYI/Optional]***

From man syscall(2):

```
--snip--

 Architecture calling conventions
       Every architecture has its own way of invoking and passing arguments to
the kernel.  The details  for various architectures are listed in the two tables
below.
```

The first table lists the instruction used to transition to kernel mode (which might not be the fastest or best way to transition to the kernel, so you might have to refer to vdso(7)), the register used to indicate the system call number, the register used to return the system call result, and the register used to signal an error.

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error	Notes
alpha	callsys	v0	v0	a4	a3	1, 6
arc	trap0	r8	r0	-	-	
arm/OABI	swi NR	-	a1	-	-	2
arm/EABI	swi 0x0	r7	r0	r1	-	
arm64	svc #0	x8	x0	x1	-	
blackfin	excpt 0x0	P0	R0	-	-	
i386	int $0x80	eax	eax	edx	-	
ia64	break 0x100000	r15	r8	r9	r10	1, 6
m68k	trap #0	d0	d0	-	-	
microblaze	brki r14,8	r12	r3	-	-	
mips	syscall	v0	v0	v1	a3	1, 6
nios2	trap	r2	r2	-	r7	
parisc	ble 0x100(%sr2, %r0)	r20	r28	-	-	
powerpc	sc	r0	r3	-	r0	1
powerpc64	sc	r0	r3	-	cr0.SO	1
riscv	ecall	a7	a0	a1	-	
s390	svc 0	r1	r2	r3	-	3
s390x	svc 0	r1	r2	r3	-	3
superh	trap #0x17	r3	r0	r1	-	4, 6
sparc/32	t 0x10	g1	o0	o1	psr/csr	1, 6
sparc/64	t 0x6d	g1	o0	o1	psr/csr	1, 6
tile	swint1	R10	R00	-	R01	1
x86-64	syscall	rax	rax	rdx	-	5
x32	syscall	rax	rax	rdx	-	5
xtensa	syscall	a2	a2	-	-	

--snip--

The second table shows the registers used to pass the system call arguments.

Arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
alpha	a0	a1	a2	a3	a4	a5	-	
arc	r0	r1	r2	r3	r4	r5	-	
arm/OABI	a1	a2	a3	a4	v1	v2	v3	
arm/EABI	r0	r1	r2	r3	r4	r5	r6	
arm64	x0	x1	x2	x3	x4	x5	-	
blackfin	R0	R1	R2	R3	R4	R5	-	
i386	ebx	ecx	edx	esi	edi	ebp	-	
ia64	out0	out1	out2	out3	out4	out5	-	
m68k	d1	d2	d3	d4	d5	a0	-	
microblaze	r5	r6	r7	r8	r9	r10	-	
mips/o32	a0	a1	a2	a3	-	-	-	1
mips/n32,64	a0	a1	a2	a3	a4	a5	-	
nios2	r4	r5	r6	r7	r8	r9	-	
parisc	r26	r25	r24	r23	r22	r21	-	
powerpc	r3	r4	r5	r6	r7	r8	r9	
powerpc64	r3	r4	r5	r6	r7	r8	-	
riscv	a0	a1	a2	a3	a4	a5	-	

```
       s390          r2     r3     r4     r5     r6     r7     -
       s390x         r2     r3     r4     r5     r6     r7     -
       superh        r4     r5     r6     r7     r0     r1     r2
       sparc/32      o0     o1     o2     o3     o4     o5     -
       sparc/64      o0     o1     o2     o3     o4     o5     -
       tile          R00    R01    R02    R03    R04    R05    -
       x86-64        rdi    rsi    rdx    r10rcx    r8     r9     -
       x32           rdi    rsi    rdx    r10    r8     r9     -
       xtensa        a6     a3     a4     a5     a8     a9     -
--snip--
Note that these tables don't cover the entire calling convention— some
architectures may indiscriminately clobber other registers not listed here.
```

<<
See the relevant CPU ABI document for the full details. An intro to processor ABI - simpler and easier perhaps to assimilate quickly - can be found here:
https://kaiwantech.wordpress.com/2018/05/07/application-binary-interface-abi-docs-and-their-meaning/
>>

Implementation of System Calls within Android's (AOSP) Bionic (it's libc replacement)

Lets take the common getpid(2) system call as an example:

x86_64
<AOSP>/bionic/libc/arch-x86_64/syscalls/__getpid.S
<<
(A part) of this superb article - [***LINUX SYSTEM CALLS UNDER THE HOOD, Julien Sobczak, Aug 2021***]() - illustrates a simplified view to how system calls are implemented on Linux!

Notice the **`syscall`** *(within the library getpid() routine!):*

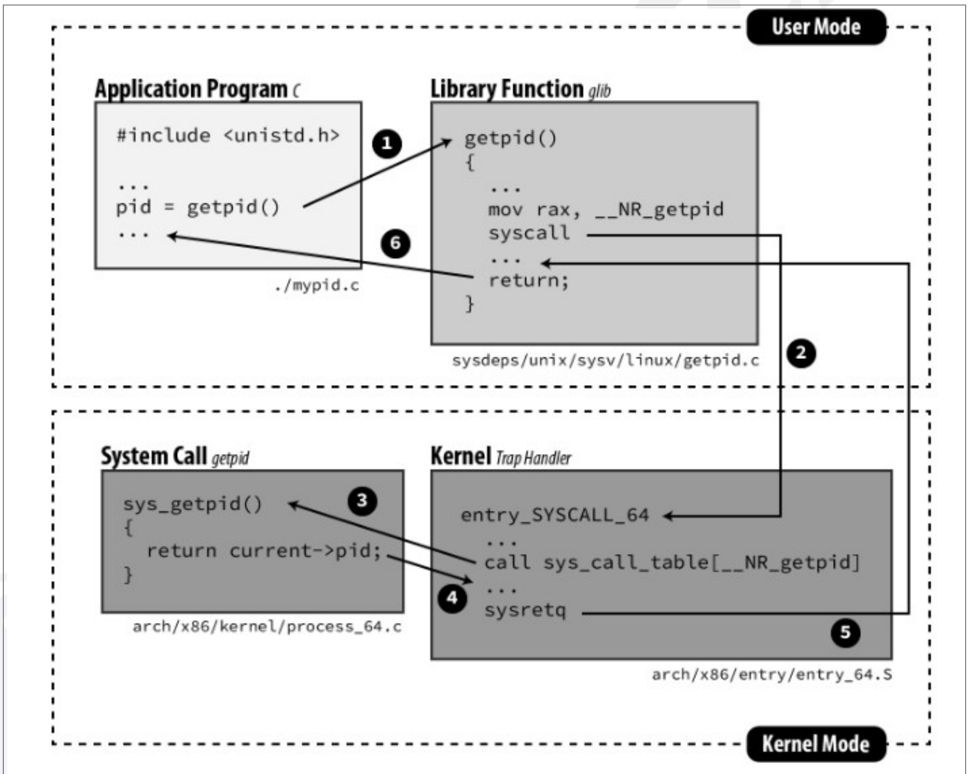


Figure 2. Steps when executing a system call

>>
```
/* Generated by gensyscalls.py. Do not edit. */
#include <private/bionic_asm.h>

ENTRY(__getpid)
    movl    $__NR_getpid, %eax
    syscall
    cmpq    $-MAX_ERRNO, %rax
    jb      1f
    negl    %eax
    movl    %eax, %edi
```

```
    call    __set_errno_internal
1:
    ret
END(__getpid)
.hidden __getpid
```

ARM (Aarch32)

<AOSP>/bionic/libc/arch-arm/syscalls/__getpid.S

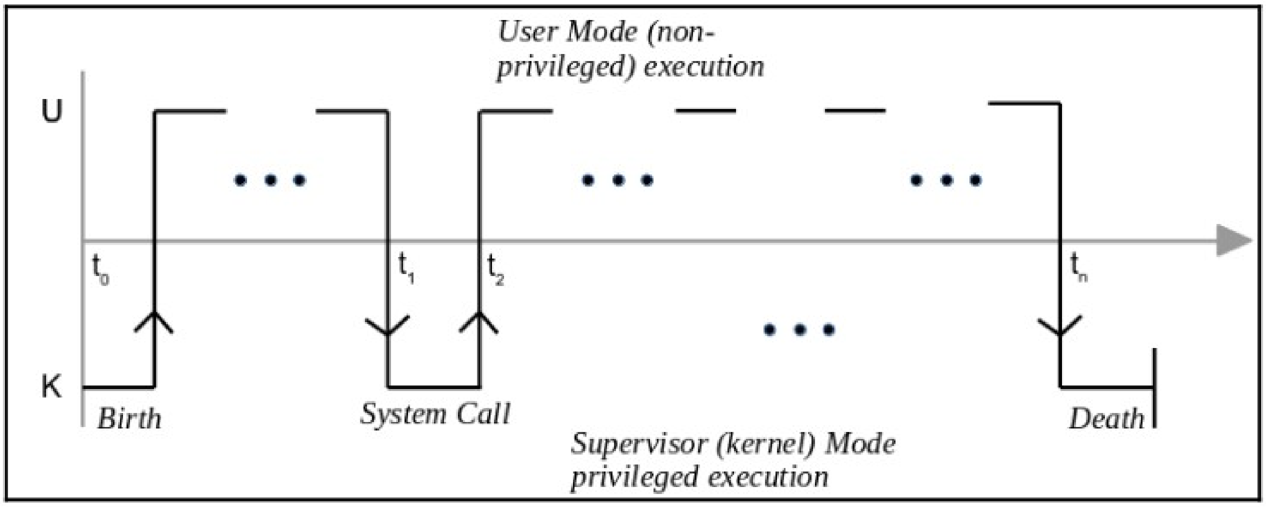
```
/* Generated by gensyscalls.py. Do not edit. */

#include <private/bionic_asm.h>

ENTRY(__getpid)
    mov     ip, r7
    .cfi_register r7, ip
    ldr     r7, =__NR_getpid
    swi     #0
    mov     r7, ip
    .cfi_restore r7
    cmn     r0, #(MAX_ERRNO + 1)
    bxls    lr
    neg     r0, r0
    b       __set_errno_internal
END(__getpid)
```

ARM-64 (Aarch64)

<AOSP-Pie_9.0.0-r3>/bionic/libc/arch-arm64/syscalls/__getpid.S

```
/* Generated by gensyscalls.py. Do not edit. */

#include <private/bionic_asm.h>

ENTRY(__getpid)
    mov     x8, __NR_getpid
    svc     #0

    cmn     x0, #(MAX_ERRNO + 1)
    cneg    x0, x0, hi
    b.hi    __set_errno_internal

    ret
END(__getpid)
.hidden __getpid
```

Flow of a Process – Birth to Death – between privilege levels



From: *[‘Hands-On System Programming with Linux’, Kaiwan N Billimoria, Packt]*.

…

Various layers within Linux, also showing separation between the userland and kernel space

User mode	**User applications**	For example, bash, LibreOffice, Apache OpenOffice, Blender, 0 A.D., Mozilla Firefox, etc.				
	Low-level system components:	**System daemons**: *systemd*, *runit*, *logind*, *networkd*, *soundd*, ...	**Windowing system**: *X11*, *Wayland*, *Mir*, *SurfaceFlinger* (Android)	**Other libraries:** *GTK+*, *Qt*, *EFL*, *SDL*, *SFML*, *FLTK*, *GNUstep*, etc.		**Graphics**: *Mesa*, *AMD Catalyst*, ...
	C standard library	`open()`, `exec()`, `sbrk()`, `socket()`, `fopen()`, `calloc()`, ... (up to 2000 subroutines) *glibc* aims to be POSIX/SUS-compatible, *uClibc* targets embedded systems, *bionic* written for Android, etc.				
Kernel mode	**Linux kernel**	`stat`, `splice`, `dup`, `read`, `open`, `ioctl`, `write`, `mmap`, `close`, `exit`, etc. (about 380 system calls) The Linux kernel System Call Interface (SCI, aims to be POSIX/SUS-compatible)				
		Process scheduling subsystem	IPC subsystem	Memory management subsystem	Virtual files subsystem	Network subsystem
		Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: *SELinux*, *TOMOYO*, *AppArmor*, *Smack*				
Hardware (CPU, main memory, data storage devices, etc.)						

OS's architecturally are broadly of two types:

```
                 OS
                  |
      |---------------------------------|
  Monolithic                      Microkernel
```

Monolithic Kernel

Googling "monolithic meaning" gets one this:

Stonehenge – a mystical site in England

- In India, the world-famous Ajanta and Ellora caves: … *The Ajanta and Ellora Caves are a group of 64 monolithic rock-cut caves that were carved over 1200 years. …*

Linux being Monolithic – a “single piece of stone”; this is ‘seen’ in two ways:

- memory view
  - the entire kernel code runs in a separate address space called “kernel-space”
  - this kernel-space is shared by all usermode processes
- how kernel is invoked
  - from user-mode : synchronous
  - via interrupts : async

Source [below]
[…]

Linux is a monolithic kernel. Device drivers and kernel extensions run in kernel space (ring 0 in many CPU architectures), with full access to the hardware, although some exceptions run in user space, for example filesystems based on FUSE.
...

Source
Monolithic Kernel

A monolithic kernel is an operating system architecture where the entire operating system is working in kernel space and is alone in supervisor mode. The monolithic model differs from other operating system architectures (such as the microkernel architecture)[1][2] in that it alone defines a high-level virtual interface over computer hardware. A set of primitives or system calls implement all operating system services such as process management, concurrency, and memory management. Device drivers can be added to the kernel as modules.

Monolithic architecture examples

Unix kernels

- BSD
- FreeBSD
- NetBSD
- OpenBSD
- Solaris 1 / SunOS 1.x-4.x

UNIX System V

- AIX
- HP-UX

Unix-like kernels

- Linux

DOS

- DR-DOS
- MS-DOS
  - Microsoft Windows 9x series (95, 98, Windows 98SE, Me)

OpenVMS
XTS-400

An obelisk – a monolith

Microkernel (in brief)

In computer science, a microkernel (also known as μ-kernel) is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system (OS). These mechanisms include low-level address space management, thread management, and inter-process communication (IPC). If the hardware provides multiple rings or CPU modes, the microkernel is the only software executing at the most privileged level (generally referred to as supervisor or kernel mode).

[citation needed] Traditional operating system functions, such as device drivers, protocol stacks and file systems, are removed from the microkernel to run in user space.[citation needed] In source code size, microkernels tend to be under 10,000 lines of code, as a general rule. MINIX's kernel, for example has fewer than 6,000 lines of code.[1]
Example : Minix, QNX, VxWorks. << OLDer >>

See the Wikipedia page on Category:Microkernels; there are over 70 microkernel OS's here.

<<
[OPTIONAL / FYI]

Source: 5 Major Software Architecture Patterns
“**Microkernel Pattern**

The microkernel architectural pattern is also referred to as a plug-in architectural pattern. It is typically used when software teams create systems with interchangeable components.
It applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. It'll also serves as a socket for plugging in these extensions and coordinating their collaboration.

--*snip*--

The microkernel architecture pattern consists of two types of architecture components: a core system and plug-in modules. Application logic is divided between independent plug-in modules and the basic core system, providing extensibility, flexibility, and isolation of application features and custom processing logic. And the core system of the microkernel architecture pattern traditionally contains only the minimal functionality required to make the system operational.

Perhaps the best example of the microkernel architecture is the *Eclipse IDE*. Downloading the basic Eclipse product provides you little more than an editor. However, once you start adding plug-ins, it becomes a highly customizable and useful product. ...”
>>

Hybrid OS

A hybrid kernel is a kernel architecture based on combining aspects of microkernel and monolithic kernel architectures used in computer operating systems. The traditional kernel categories are monolithic kernelsand microkernels (with nanokernels and exokernels seen as more extreme versions of microkernels). The category is controversial due to the similarity to monolithic kernel; the term has been dismissed by Linus Torvalds as simple marketing.[1]

The idea behind this category is to have a kernel structure similar to a microkernel, but implemented in terms of a monolithic kernel. In contrast to a microkernel, all (or nearly all) operating system services are in kernel space. While there is no performance overhead for message passing and context switching between kernel and user mode, as in monolithic kernels, there are no reliability benefits of having services in user space, as in microkernels.

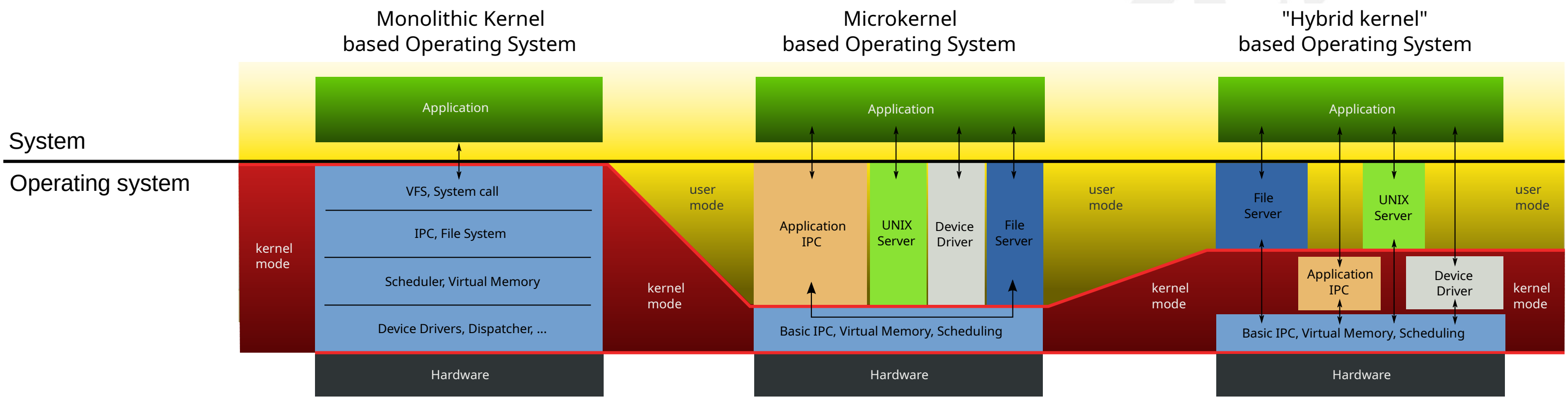
Implementations
BeOS kernel
  Haiku kernel
Syllable
BSD-based
  DragonFly BSD (first non-Mach BSD OS to use a hybrid kernel)

XNU kernel (core of Darwin, used in Mac OS X and iOS)
NetWare kernel[7]
Inferno kernel
NT kernel (used in Windows NT 3.1, Windows NT 3.5, Windows NT 4.0, Windows 2000, Windows Server 2003, Windows XP, Windows Vista, Windows Server 2008, Windows 7, Windows Server 2008 R2,Windows 8, and Windows Server 2012)

... One prominent example of a hybrid kernel is the Microsoft NT kernel that powers all operating systems in theWindows NT family, up to and including Windows 10 and Windows Server 2012, and powers Windows Phone 8,Windows Phone 8.1, and Xbox One. ...

<< Find more overview information on the Windows NT design here, *section "NT kernel"* >>

ReactOS kernel



Ref:

How does Linux kernel compare to microkernel architectures?

Why is Linux called a monolithic kernel?

Instructor : mention what Process and Interrupt Contexts are.

Miscellaneous

Using Ftrace / trace-cmd to see 'Hello, world' in the kernel

Tracing can reveal how the 'hello, world' app really works… here, I used my convenience script **trccmd** (a front-end to **trace-cmd(1)** which is itself a front-end to **Ftrace**!) to trace the classic 'Hello, world' process (here, with function parameter values turned on):

```
[ ... ]
hello-21528   0.... 12154.366192: sys_enter:            NR 1 (1,
562f902342a0, d, 7f3d63d53be0, 0, 7c)
hello-21528   0.... 12154.366193: sys_enter_write:      fd: 0x00000001,
buf: 0x562f902342a0, count: 0x0000000d
[ ... ]
```

Check out the parameters to *sys_enter_write()* (part of the entry to the write(2) invoked via the glibc printf(3) or puts(3)): you can clearly see all 3 parameters:

1. the file descriptor (1, which is *stdout*)
2. buffer pointer (`0x562f902342a0 -` a usermode virtual address)
3. # of bytes to write (0xd = 13 = strlen(Hello, world)).

More importantly, the above is kernel code that runs in the context of the *hello, world* process; iow, *in process context*; it's a monolithic kernel!

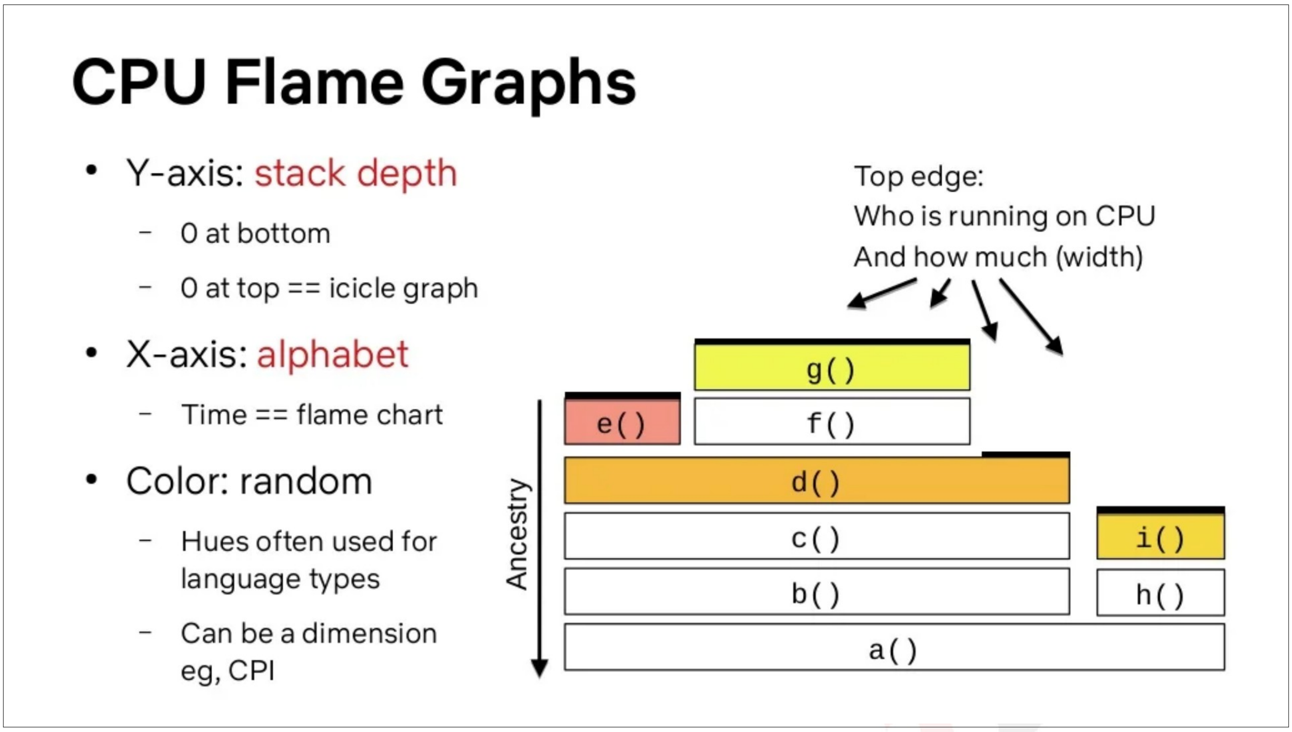
FYI / OPTIONAL

CPU Flame Graphs

Even better, use Brendan Gregg's fantastic ***FlameGraph*** set of scripts to actually visualize the stack – and thus what happened when we do `printf("Hello, world\n");` !

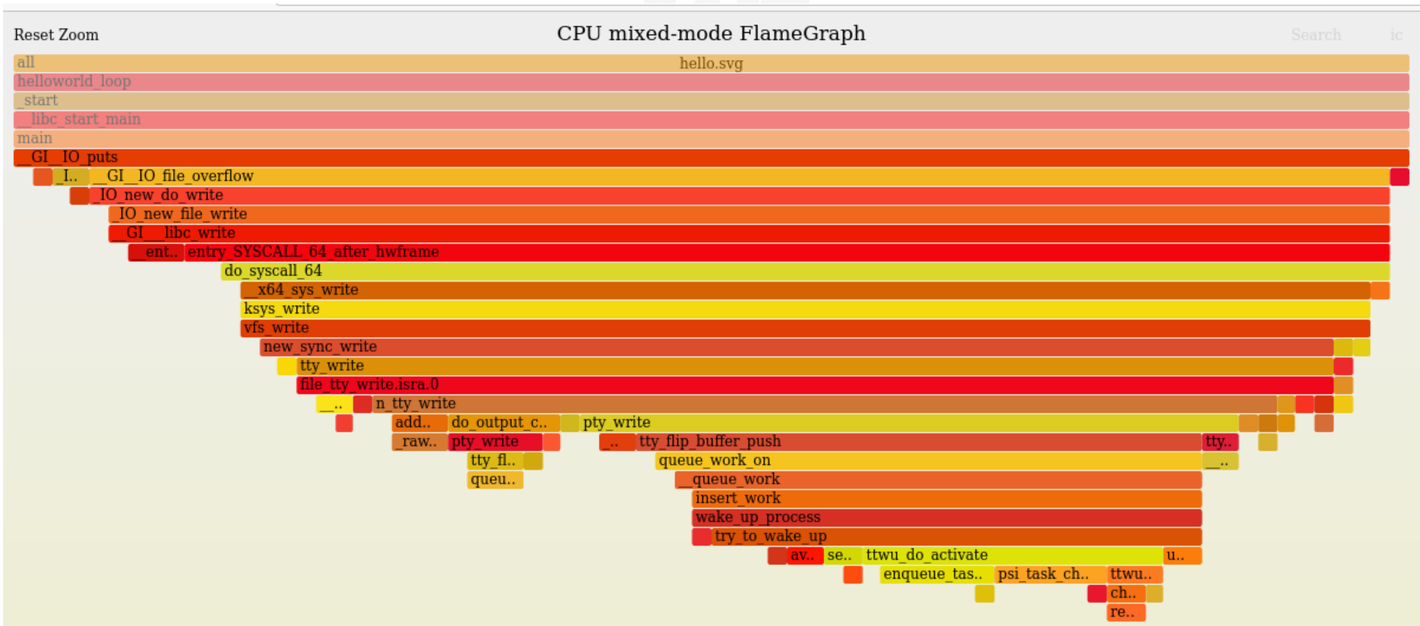
Interpretation of a FlameGraph

X-axis: The width of the box shows the total time it was on-CPU or part of an ancestry that was on-CPU (based on sample count). Functions with wide boxes may consume more CPU per execution than those with narrow boxes, or, they may simply be called more often.



Here’s the – inverted / icicle style – FlameGraph for the K&R C ‘Hello, world’ program on x86_64 Linux!

The relevant portion is shown – notice, the printf() gets optimized to a puts() by the compiler! It ultimately, of course, becomes the write() system call …

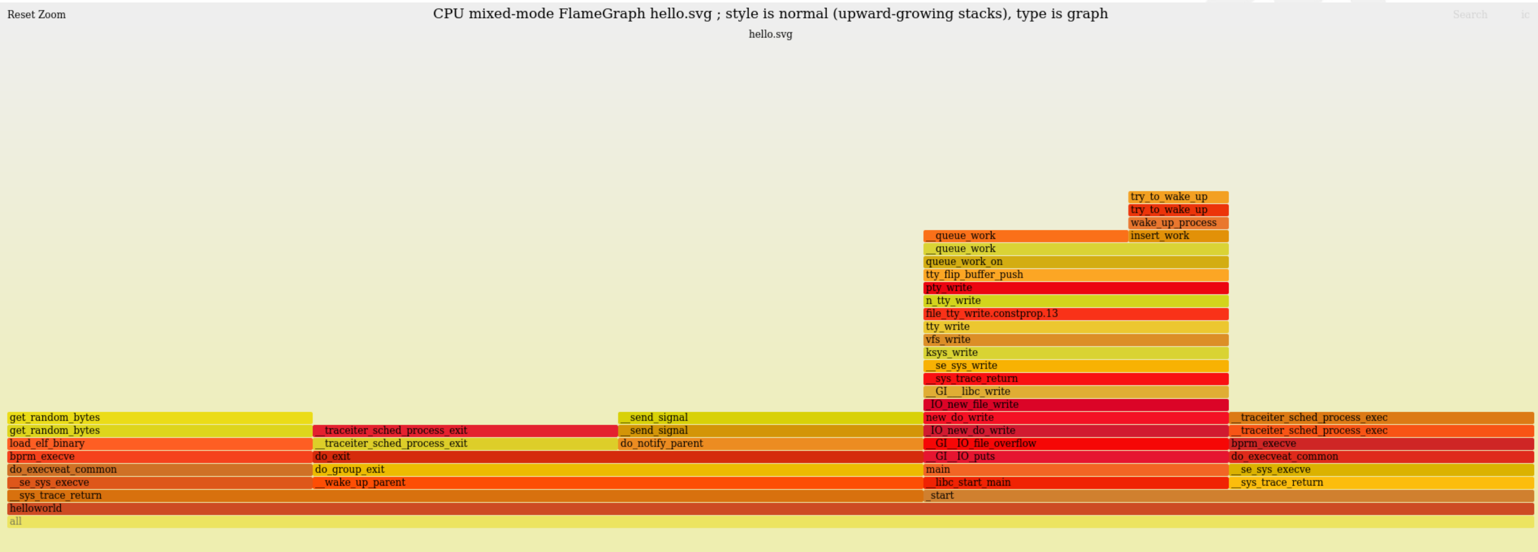


I did this by:

1. running a 'Hello, world' in a loop (so that we can capture printf()'s later too)
2. generate the FlameChart – a FlameGraph with the --flamechart option to set graph style to 'icicle' – downward-growing! - with my wrapper script here: https://github.com/kaiwan/L5_debug_trg/tree/master/flamegraph .

Try it!

Here's a flamegraph when running an unoptimized 'Hello, world' on an ARMv8 (Raspberry Pi 4, BCM2837 SoC):

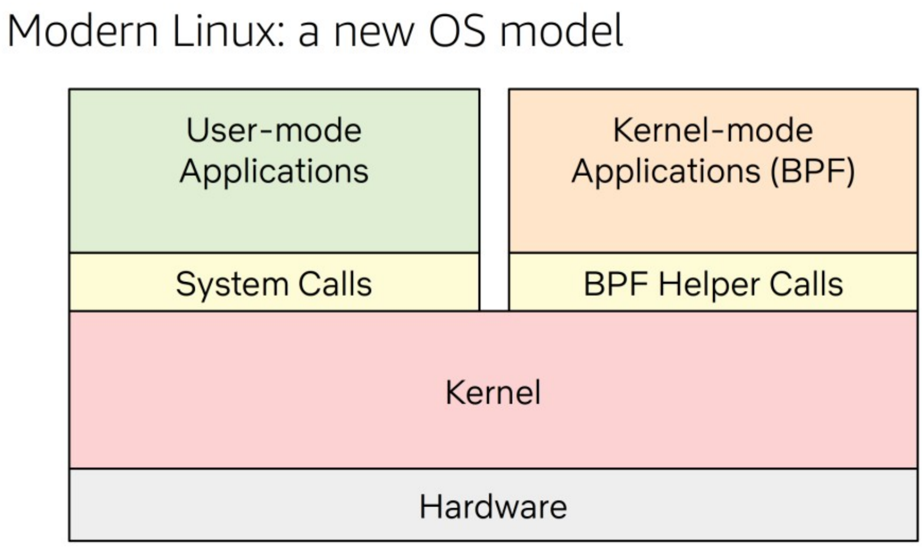


The eBPF revolution

eBPF (or simply, BPF) is a virtual machine technology allowing one to write a (small) program in userspace and run it within the kernel!

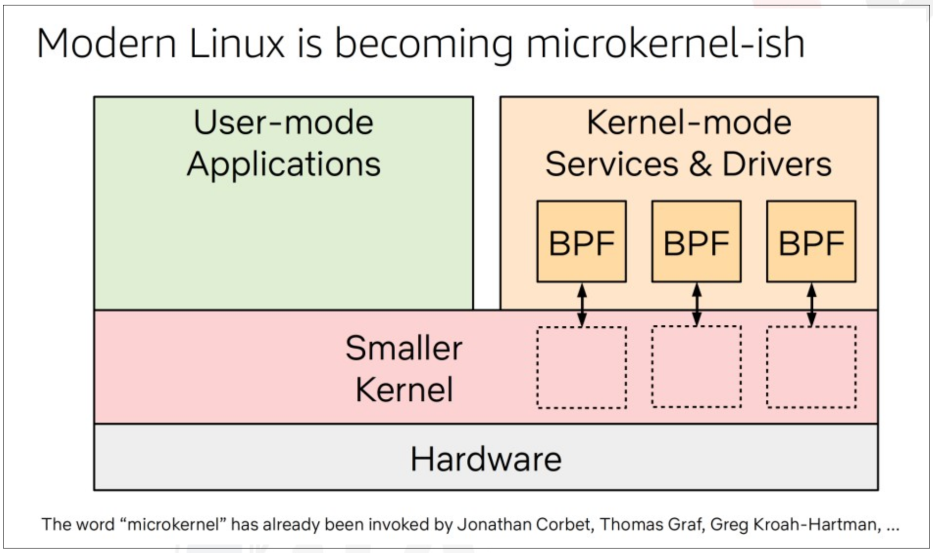
"BPF is an in-kernel virtual machine" !
"A safer form of C"

eBPF is changing the landscape…



It's like having a superpower !

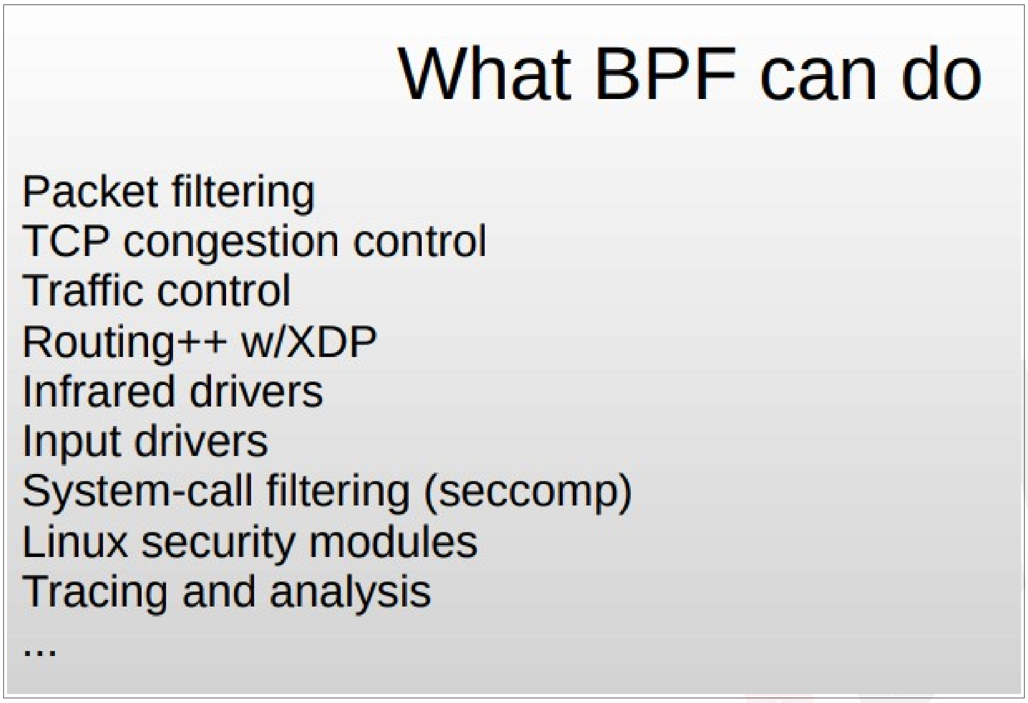
[Src]



See

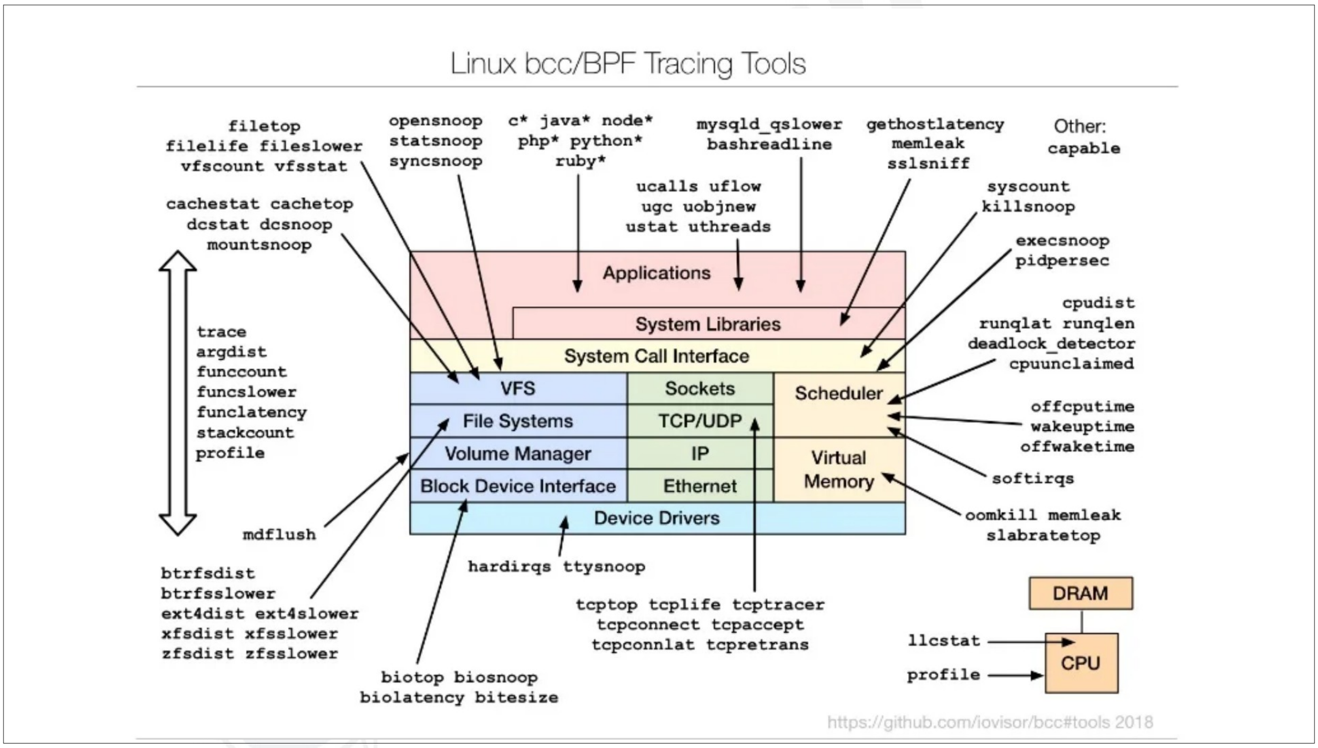
- *Velocity 2017: Performance Analysis Superpowers with Linux eBPF, YouTube, Sept 2017*
- *'Linux Extended BPF (eBPF) Tracing Tools' by Brendan Gregg)*

https://lwn.net/talks/2023/kr-osseu.pdf



+ Observability tooling !

Install the package bpfcc-tools; (on Ubuntu: `sudo apt install bpfcc-tools`).
run: `sudo <toolname>-bpfcc [-h]`



<< ***In-depth article on eBPF*:**
~~[every Boring Problem Found in eBPF] by @FridayOrtiz~~ ,, Feb 2022 >>

Installation of eBPF tooling

https://github.com/iovisor/bcc/blob/master/INSTALL.md
 	
Ubuntu:
`sudo apt-get install bpfcc-tools linux-headers-$(uname -r)`
Tools get installed under /usr/sbin/*-bpfcc
 	
Fedora 30 and higher:
`sudo dnf install bcc`
Tools get installed under /usr/share/bcc/tools .

Example – using eBPF to figure out which config file exactly is being parsed when, for configuring the Linux kernel, the 'make menuconfig' command is run

FYI, this experiment is carried out on the upstream raspberrypi Linux kernel (6.1.21; that I forked from here: https://github.com/kaiwan/rpi-linux-kernel).

Procedure:

1. Install the bpfcc-tools package
2. In one terminal window, setup opensnoop-bpfcc to run:
   `sudo opensnoop-bpfcc |tee of1.txt`
3. Run the menuconfig program:
   `make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- menuconfig`
4. Examine the saved file of all open's ! (of1.txt here):

...

```
11944 198327 mconf              5   0 Documentation/Kconfig
11945 198327 mconf              4   0 Documentation/Kconfig
11946 198327 mconf             -1   2 .config
11947 198327 mconf             -1   2 ./.config
11948 198327 mconf              3   0 arch/arm/configs/multi_v7_defconfig
11949 198327 mconf              3   0 /lib/terminfo/x/xterm-256color
11950 911    gpm                4   0 /dev/tty0
11951 661    systemd-oomd       7   0 /proc/meminfo
11952 661    systemd-oomd       7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/memory.pressure
11953 661    systemd-oomd       7   0 /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/memory.current
```

…

I find that the config file parsed is arch/arm/configs/multi_v7_defconfig .

Licensing

The Linux kernel is always intended to be released under the **GPL v2** license.

'Pillars" or 'Freedoms', courtesy ChatGPT:

k

the 4 freedoms of the GPL license?

The GPL (General Public License) is a widely used free software license that was created by the Free Software Foundation (FSF). It is designed to protect the rights of users and ensure that software remains free and open for everyone to use and modify. The GPL has four key freedoms, which are:

1. Freedom to run the program for any purpose: The user is allowed to run the software for any purpose they wish, without any restrictions.
2. Freedom to study how the program works: The user has access to the source code of the software and can study how it works and modify it as needed.
3. Freedom to redistribute copies: The user is free to distribute copies of the software to others, whether for free or for a fee.
4. Freedom to improve the program and release improvements to the public: The user has the freedom to modify the software and release their modifications to the public, as long as they also distribute the source code and make it available under the same license.

These four freedoms ensure that users of GPL-licensed software have the ability to use, modify, and share the software freely, without any restrictions or limitations.

From ChatGPT:

k

what does 'free as in free beer' mean?

"Free as in free beer" is a phrase used to describe something that is available at no cost, like a free drink. The phrase contrasts with "free as in freedom," which refers to the concept of open-source software or other intellectual property that is free to use, modify, and distribute.

In other words, "free as in free beer" means that something is being given away for free without any expectation of payment, whereas "free as in freedom" means that the user has the freedom to use, modify, and distribute the product, even if it may come with a cost.

[OPTIONAL / FYI]

An FAQ regarding keeping track of Linux kernel Changes

The Linux kernel is a very fast moving target: things change, quite rapidly at times, new enhancements and features get merged, **kernel internal APIs / ABIs change, get deprecated**, etc etc.

Why do kernel APIs change? To keep them fresh and working! For security reasons as well…

Src: The Soul of the Movement: 30 Years of Linux (Part 1), deMasi, Sept 2021:
"Concerning Linux's longevity, Rostedt adds, "From the beginning, Torvalds held fast to one rule—you don't break user space. Meaning, if you have an application running on one kernel, it should run on all kernels after that and that's another reason why Linux is so successful. You don't have to worry about porting your applications to the next version of the kernel. They may prove to be buggy and you should update them, but they will always work as they did in the past." …"

From Greg Hartmann's presentation (article):

"When he turned his attention to library developers, his first slide simply read: "I pity you."

> "You never know if an API is really useful, until you have too many people using it to ever be able to change it."—Greg Kroah-Hartman
>
> "This is the hardest job ever," he said with a laugh. "I really, really feel sorry for you… It's one of the hardest things to do." …"

An example: a kernel module which built and worked perfectly on an earlier kernel fails to even compile on a recent (4.10) kernel:

```
[...]
$ make
make -Wall -C /lib/modules/4.10.0-33-generic/build  M=<...>  modules
make[1]: Entering directory '/usr/src/linux-headers-4.10.0-33-generic'
Building with KERNELRELEASE = 4.10.0-33-generic
  CC [M]  <...>/2nf.o
<...>/2nf.c: In function 'reg_nf':
<...>/2nf.c:162:10: error: 'struct nf_hook_ops' has no member named 'owner'
  psNFHook->owner = THIS_MODULE;
          ^~
<...>/2nf.c: In function 'nf_init':
<...>/2nf.c:230:44: error: passing argument 3 of 'reg_nf' from incompatible
pointer type [-Werror=incompatible-pointer-types]
```

```
  reg_nf(&nfhk_in_pre, NF_INET_PRE_ROUTING, nfhook_in_pre, PF_INET,
NF_IP_PRI_FIRST);
                                                       ^~~~~~~~~~~~~
<...>/2nf.c:159:20: note: expected ‘unsigned int (*)(void *, struct sk_buff
*, const struct nf_hook_state *)’ but argument is of type ‘unsigned int (*)
(const struct nf_hook_ops *, struct sk_buff *, const struct net_device *,
const struct net_device *, int (*)(struct sk_buff *))’
 static inline void reg_nf(struct nf_hook_ops *psNFHook, int hooknum,
                    ^~~~~~
<...>/2nf.c:231:43: error: passing argument 3 of ‘reg_nf’ from incompatible
pointer type [-Werror=incompatible-pointer-types]
  reg_nf(&nfhk_local_in, NF_INET_LOCAL_IN, nfhook_local_in, PF_INET,
NF_IP_PRI_FIRST);
                                                      ^~~~~~~~~~~~~~~
<...>/2nf.c:159:20: note: expected ‘unsigned int (*)(void *, struct sk_buff
*, const struct nf_hook_state *)’ but argument is of type ‘unsigned int (*)
(const struct nf_hook_ops *, struct sk_buff *, const struct net_device *,
const struct net_device *, int (*)(struct sk_buff *))’
 static inline void reg_nf(struct nf_hook_ops *psNFHook, int hooknum,
                    ^~~~~~
[...]

cc1: some warnings being treated as errors
scripts/Makefile.build:301: recipe for target '<...>/2nf.o' failed
make[2]: *** [<...>/2nf.o] Error 1
Makefile:1524: recipe for target '_module_<...>' failed
make[1]: *** [_module_<...>] Error 2
make[1]: Leaving directory '/usr/src/linux-headers-4.10.0-33-generic'
Makefile:16: recipe for target 'build' failed
make: *** [build] Error 2
$
```

2.6.36 : the ioctl() signature changes

[below: code from a device driver taking this into account]

```
#include <linux/version.h>
[…]
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,36)
static long rwmem_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
#else
static int rwmem_ioctl(struct inode *ino, struct file *filp, unsigned int cmd, unsigned long arg)
#endif
[…]
```

Why are the kernel APIs “unstable”?

From: https://www.kernel.org/doc/html/latest/process/1.Intro.html#the-importance-of-getting-code-into-the-mainline

… While kernel developers strive to maintain a stable interface to user space, the internal kernel API is in constant flux. The lack of a stable internal interface is a deliberate design decision; it allows fundamental improvements to be made at any time and results in higher-quality code. But one result of that policy is that any out-of-tree code requires constant upkeep if it is to work with new kernels. Maintaining out-of-tree code requires significant amounts of work just to keep that code working.

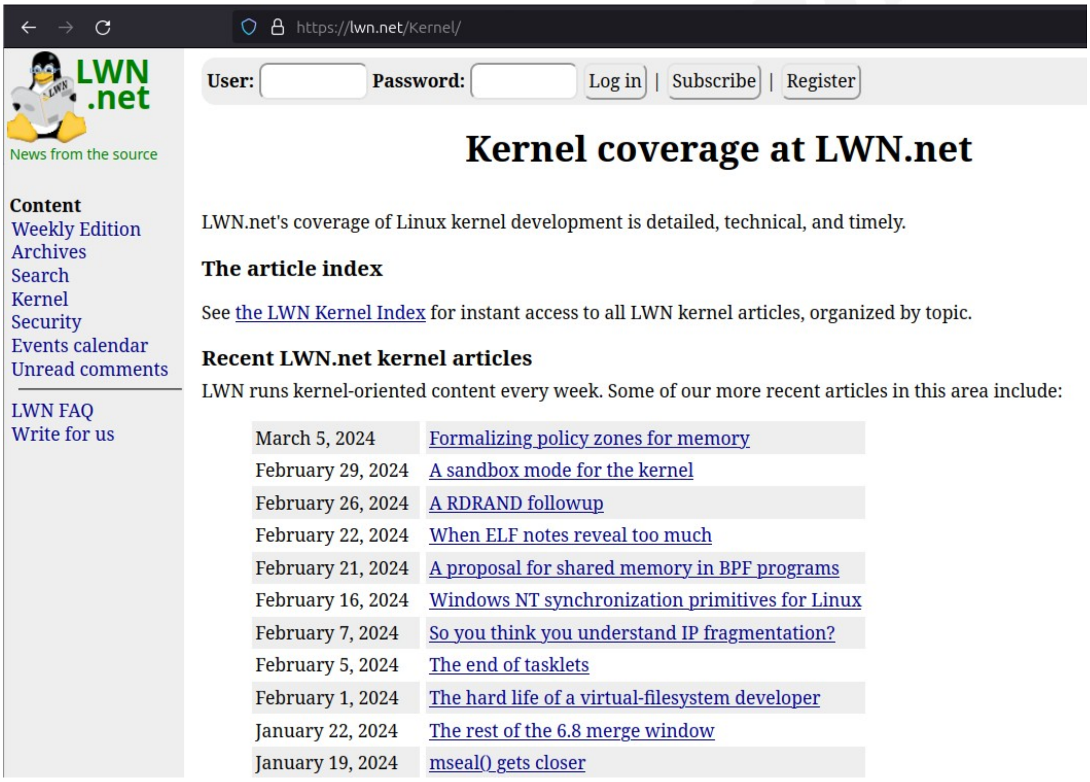
Code which is in the mainline, instead, does not require this work as the result of a simple rule requiring any developer who makes an API change to also fix any code that breaks as the result of that change. So code which has been merged into the mainline has significantly lower maintenance costs. …

< many more points follow as well >

How can one sanely keep track of all kernel changes?

The knee-jerk answer: follow the LKML (Linux Kernel Mailing List).
But "sanely"?
:-)

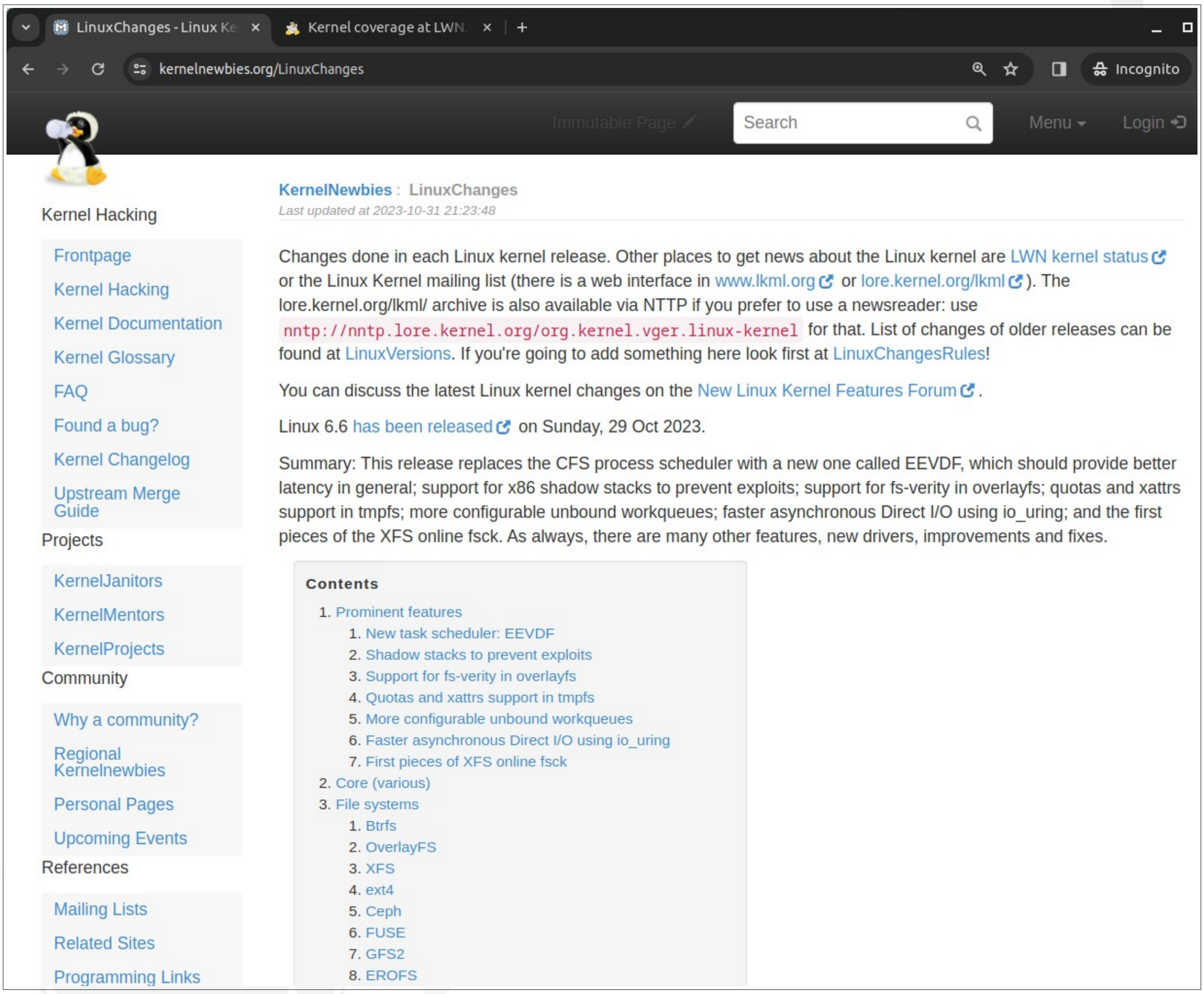
Read the
https://lwn.net/Kernel/



and the
kernelnewbies “Linux Changes” website !

1. The page
http://kernelnewbies.org/LinuxChanges

will have the *latest mainline kernel* changes information:
<< 6.6 at the time of this insertion (Jan 2024) >>



[SIDEBAR :: Get the Linux kernel ‘finger banner’]

`curl -L https://www.kernel.org/finger_banner`

As of 06 Mar 2024:

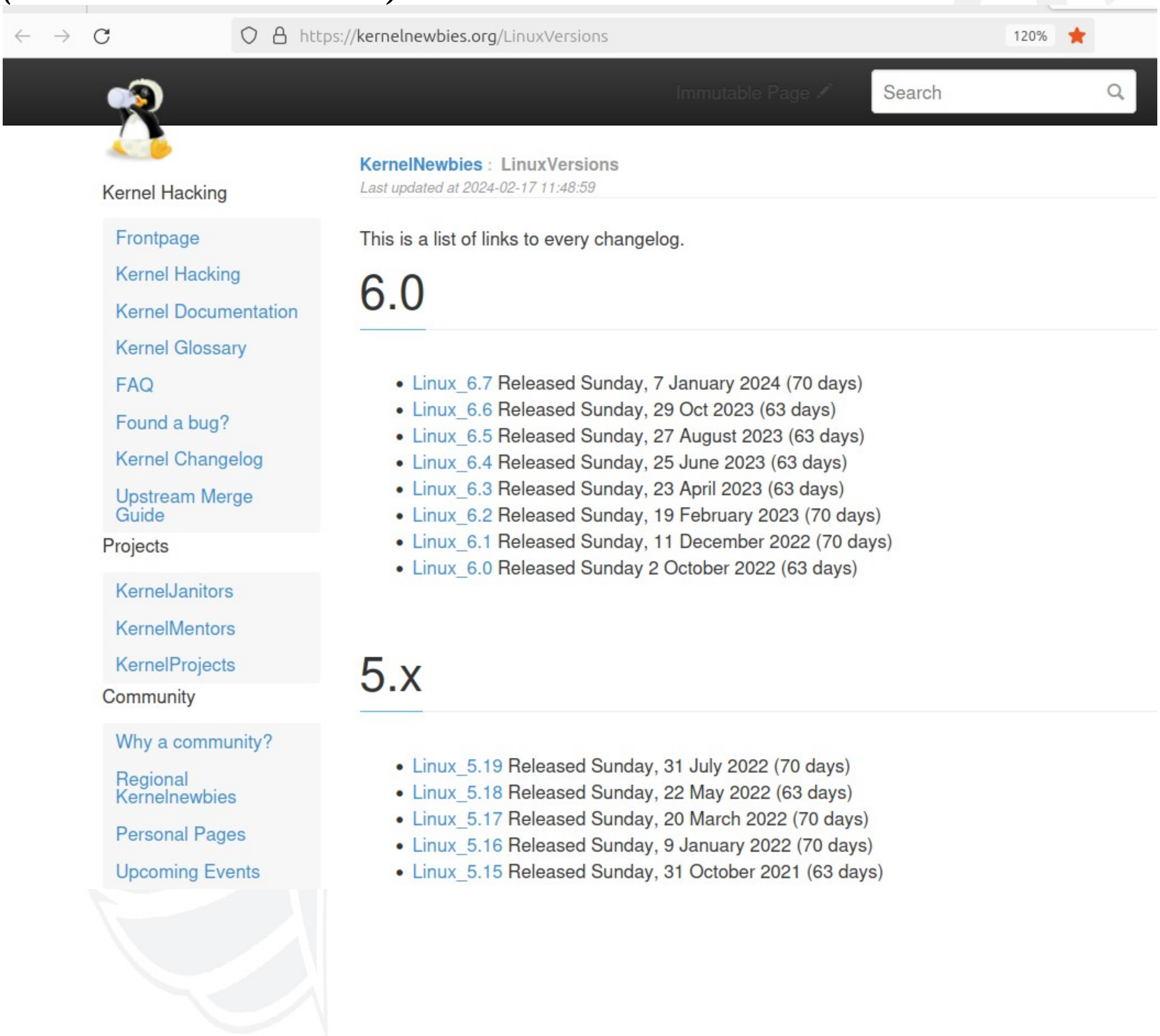
```
$ curl -L https://www.kernel.org/finger_banner
The latest stable version of the Linux kernel is:             6.7.8
The latest mainline version of the Linux kernel is:           6.8-rc7
The latest stable 6.7 version of the Linux kernel is:         6.7.8
```

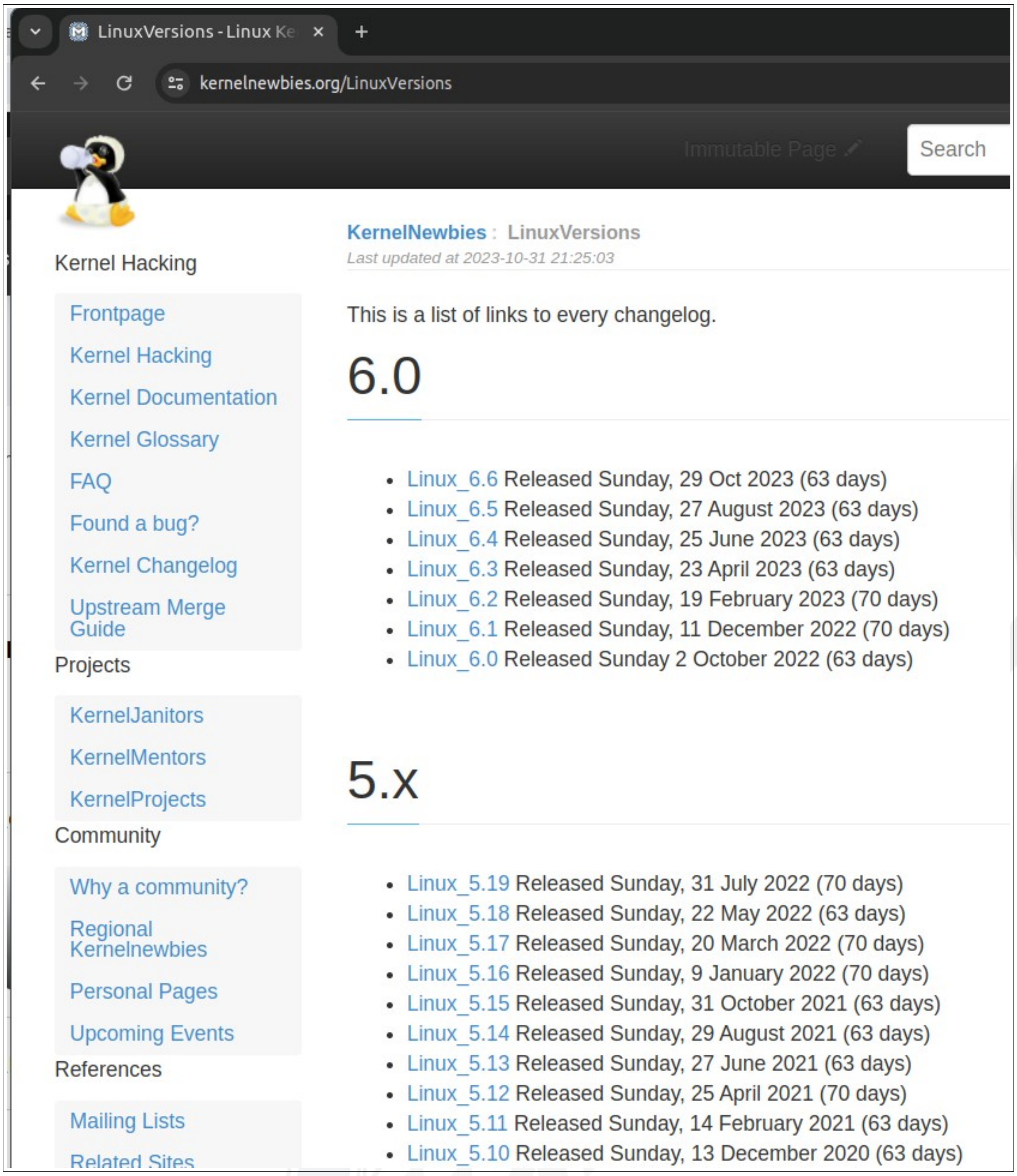
```
The latest longterm 6.6 version of the Linux kernel is:       6.6.20
The latest longterm 6.1 version of the Linux kernel is:       6.1.80
The latest longterm 5.15 version of the Linux kernel is:      5.15.150
The latest longterm 5.10 version of the Linux kernel is:      5.10.211
The latest longterm 5.4 version of the Linux kernel is:       5.4.270
The latest longterm 4.19 version of the Linux kernel is:      4.19.308
The latest linux-next version of the Linux kernel is:         next-20240306
$
```

---

2. To see links to all kernel versions, go to
http://kernelnewbies.org/LinuxVersions

(Partial screenshot : 06Mar2024)





How would a professional Linux product company select a kernel version and what would the product life cycle be like? See this Wikipedia content on RedHat's product life ccyle and kernel backporting.

---

https://kaiwantech.com

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp