Linux Haters Club

"Two of the most famous products of Berkeley are LSD and Unix. I don't think that is a coincidence" ~Jeremy S. Anderson

"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'." ~ Linus Benedict Torvalds

1. A Fork in the Road

Fork is an anachronism: a relic from another era that is out of place. [1]

It isn't thread safe[2], inefficient and unscalable, and introduces security concerns, and doesn't compose. [3]

Alternatives include using spawn, e.g. `posix_spawn`[4] or `CreateProcess`[5] from NT.

```
pid_t pid = fork(); // Create a child process

if (pid == 0) {
    // child
    const char* args[] = {"echo", "Hello from the child process!", NULL};
    execvpe("echo", args, environ);
} else {
    // parent
    waitpid(pid, &status, 0);
}
```

[1] A Fork in the road
[2] fork man page
[3] A Fork in the road
[4] spawn man page
[5] CreateProcess Docs

Too many exec types

```c
int execl(const char *path, const char *arg, .../*, (char *) NULL */);
int execlp(const char *file, const char *arg, ... /*, (char *) NULL */);
int execle(const char *path, const char *arg, ... /*, (char *) NULL, char *const envp[] */
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

`exec` replaces the calling process image rather than creating a new process.[6]

[6] Exec man page

Compared to spawn:

```c
const char *argv[] = {"echo", "Hello from the child process!", NULL};

printf("Spawning child process...\n");

// 1. &pid           stores the child's process ID
// 2. "echo"         path to the executable
// 3. NULL           file actions struct NULL = keep parent's FDs
// 4. NULL           spawn attributes struct NULL uses defaults
// 5. argv           arguments array
// 6. environ        environment variables array
int result = posix_spawnp(&pid, "echo", NULL, NULL, argv, environ);

waitpid(pid, &status, 0); // wait for child
```

2. The monolithic kernel makes one bug everybody's problem

Linux puts core services and many device drivers in privileged kernel space.[7]

A bug in the kernel path can crash the whole machine, or do worse.[8]

One example is "SACK Panic": specially crafted TCP SACK (selective acknowledgement) packets could trigger a kernel integer overflow and panic.[9]

[7] Linux kernel documentation: Device Drivers
[8] Linux kernel documentation: ramfs, rootfs and initramfs; Linux kernel documentation: kdump
[9] The TCP SACK panic

3. POSIX abstractions are rubbish

The POSIX process is a CPU-centric abstraction because CPUs were the central computation resource for many decades of Unix evolution...

However, offloading computation to domain-specific coprocessors and accelerators such as GPUs for graphics and NICs for packet processing has become mainstream (AF_XDP)

However, POSIX does not have the machinery to address coprocessors or accelerators.[10]

Most people bypass the OS for high-speed I/O, with eBPF[11]

You can dodge the OS cache with `O_DIRECT`.

`io_uring` as an attempt to provide submission and completion queues for faster speed.[12]

[10] USENIX: Transcending POSIX: The End of an Era?
[11] EBPF
[12] Linux kernel documentation: io_uring

4. Linux Sockets

Socket, bind, listen, accept, epoll, poll, select: a readiness-based API and its per-connection bookkeeping can become a scalability tax. Is it the abstraction, the syscall and wakeup path, or the application architecture that becomes the bottleneck?

This needs workload-specific evidence: `epoll` is explicitly designed to scale to large numbers of watched descriptors, so "huge bottlenecks" is too broad as a universal claim. The stronger critique is that readiness APIs make applications manage state around repeated readiness notifications, while completion APIs deliver finished operations.[13][14][15]

[13] Linux man-pages: epoll(7)
[14] Linux kernel documentation: io_uring
[15] Microsoft: I/O Completion Ports

`Select`

```c
// the highest fd you want to monitor
// then the readfds to monitor, writefds, "exceptional" and then a timeout
int select(int nfds, fd_set *_Nullable restrict readfds,
                  fd_set *_Nullable restrict writefds,
                  fd_set *_Nullable restrict exceptfds,
                  struct timeval *_Nullable restrict timeout);
```

After `select` has returned, `readfs` will be cleared of all file descriptors except those that are ready for reading. . . [16]

So you need to keep a copy of the fd_set and then keep resetting it in a loop.

`select()` can only monitor FD_SETSIZE fds, 1024. This will not change, use `poll` or `epoll`.[17]

The kernel must scan every bit up to the highest file descriptor value, and so does user space. For 1 ready FD, that means scanning up to 2096 FDs, and requiring a copy since this is destructive.

[16] Linux man pages: select(2)
[17] Linux man pages: select(2)

`poll` doesn't really solve the problem

```
struct pollfd {
     int   fd;         /* file descriptor */
     short events;     /* requested events */
     short revents;    /* returned events */
 };

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Still requires scanning all monitored descriptors.

5. Everything is a file

File system anarchy; devices, kernel state, normal files, config, sockets and directories are all files. That allows them to use the same `open`, `read`, `write`, `close` apis, but they lack type safety.[18][19]

There's perf overhead for reading "virtual directories", including for networking.

There's `ioctl` as a catchall for adding "more" functionality to files than was originally intended, a grab bag of flags to apply to files.

You can also delete device data, and brick devices, like your motherboard.

[18] Linux man-pages: proc(5)
[19] Linux kernel documentation: Linux ABI descriptions

6. Unix has no object namespace (or its IPC APIs are rubbish)

Unix offers many distinct IPC mechanisms—pipes, FIFOs, signals, System V and POSIX message queues, shared memory, and Unix-domain sockets with different creation, naming, lifetime, and permission rules. They are rarely used too compared to just internet sockets.

NT's Object Manager gives kernel objects a hierarchical namespace and lets processes obtain handles to named objects. [20][21][22]

[20] USENIX: Transcending POSIX: The End of an Era?
[21] Linux man-pages: unix(7)
[22] Microsoft: Managing Kernel Objects

7. C is not a low-level language

Spectre and Meltdown vulnerabilities occurred because C programmers believe their language is close to the underlying hardware.[23]

C's abstract machine is serial, even though computers have not been serial for 50 years.

Flat memory and cache are both invisible to the user.

Languages like erlang with actors or clojure with software transactional memory allow you to write concurrent code fairly easily.

[23] C is not a low-level language

8. The compiler is allowed to break your security invariants

An optimizer may remove a secret wipe as a dead store, or turn constant-time source into secret-dependent machine code. The security property you intended is not necessarily part of the language contract.[24][25]

The compile can remove or modify integer-overflow checks.[26]

[24] MITRE CWE-14: Compiler Removal of Code to Clear Buffers
[25] Breaking Bad: How Compilers Break Constant-Time Implementations
[26] CVE-2008-1685; CVE-2019-1010006

9. Unix permissions are too simple

The classic Unix model gives an inode one owner, one group, and three sets of read/write/execute bits. POSIX ACLs add that flexibility, along with mask and inheritance rules that make the effective permissions harder to read at a glance.[27]

Isolation is done with `chroot`, which just changes the root directory of a process, but otherwise isn't a security boundary. Most jailbreaks use mount to escape.

Linux then added cgroupsv1 and cgroupsv2 for isolation to limit processes

Jails are an alternative design: [28]

NT objects can carry access-control lists with entries granting or denying specific rights to users and groups. [29]

[27] Linux man-pages: acl(5); Microsoft: Access Control Lists
[28] FreeBSD Jails
[29] Microsoft: How DACLs Control Access to an Object

10. Configuration is scattered

System configuration generally lives under `/etc`, but user configuration may live in dotfiles directly in `$HOME`, under `~/.config`, or in application- specific locations. Configuration discovery, precedence, and backup therefore depend on conventions layered on top of ordinary files. XDG tried to regularize user paths, but defines multiple base directories and search paths rather than one authoritative configuration database.[30][31][32]

Windows NT introduced the Windows Registry, which allows you to edit configuration in one place, with access control, and better querying.

[30] Filesystem Hierarchy Standard: `/etc`
[31] Filesystem Hierarchy Standard: Home directories
[32] XDG Base Directory Specification

11. Microkernels

Most older operating systems are monolithic, that is the whole operating system is a single a.out file that runs in "kernel mode". This contains process management, memory management, file system, and the rest. Examples are Unix, MS-DOS, VMS, OS/360, Multics...[33]

The alternative is a microkernel-based system, in which most of the OS runs as separate processes, mostly outside the kernel. They communicate by message passing. Examples of this design include Amoeba, Chorus, Mach, and the not-yet-released Windows/NT.[34]

[33] Linux is Obsolete
[34] Linux is Obsolete