10 Things Every Linux Programmer Should Know

Linux Misconceptions in 30 Minutes

Muli Ben-Yehuda

mulix@mulix.org

IBM Haifa Research Labs

TOC

- What this talk is about
- User space vs. kernel space
- Memory allocation
- Processes vs. threads
- NPTL - Native POSIX Threads Library
- Optimization
- Use the right abstraction layer for the job
- Why coding style is important
- Why you should always check for errors
- The real cost of software development
- Portability

What this talk is about

- This talk is about common misconceptions, and things I think every Linux programmer should know.
- This talk is about learning from other people's experience and mistakes.
- This talk is about how to write better software.
- It's about knowing how things work; Knowing how things work lets us embrace and extend Linux.

This talk represents my opinions, and mine only. Feel free to disagree. . .

User space vs. kernel space

- A process is executing either in user space, or in kernel space. Depending on which priviledges, address space and address (EIP) a process is executing in, we say that it is either in user space, or kernel space.
- When executing in user space, a process has normal priviledges and can and can't do certain things. When executing in kernel space, a process has every priviledge, and can do anything.
- Processes switch between user space and kernel space using system calls.
- Note that there are exceptions and mixed cases, such as processes using iopl(). The biggest difference between user space and kernel space is conceptual - the programmer's state of mind.

Memory Allocation

- *malloc* is a library function, implemented in the standard C library. It is **not** a system call.
- *malloc* is implemented by two means in Linux. The first is the *brk()* system call, which grows a process's data segment. The second is mapping /dev/zero to get anonymous memory. In both cases, the end result is the same.
- The kernel allocates a virtual memory area for the application, but does *not* allocate physical memory for it. Only when the area is accessed physical memory is allocated. This is known as "memory overcommit", and there are ways to disable it.
- Do you really need your own malloc implementation?
- Is *malloc()* really too slow for you?

Processes vs. threads

- In Linux, processes and threads are almost the same. The major difference is that threads share the same virtual memory address space.
- The low level interface to create threads is the *clone()* system call. The higher level interface is *pthread_create()*.
- Context switches between processes in Linux are fast. Context switches between threads are even faster.
- Which is better, multi process or multi threaded? **depends**.
- Linux threading is “1-1”, not “1-N” or “M-N”.
- Linux threading libraries evolution: Linux Threads, NGPT, NPTL. The library is part of gilbc, programmer interface is POSIX pthreads.

NPTL - Native POSIX Threads Library

The New POSIX Threads Library, written by Ulrich Drepper and Ingo Molnar of Red Hat, Inc.

- Addresses shortcomings in the design and implementation of Linux Threads.
- Makes use of improved Linux kernel code and facilities. In places, the changes were written specifically for NPTL.
- Uses the newly added Futexes (Fast User space Mutexes).
- Standard in kernel 2.5 and up, and back ported to 2.4 by Red Hat.
- For more information, http://people.redhat.com/drepper/nptl-design.pdf

Optimization

- Knuth: "Premature optimization is the root of all evil".
- Avoid the desire to optimize prematurely; concentrate on getting it working *first*.
- When it's time to optimize, profile, profile, profile. The desire to "search for the coin under the streetlight" is strong, and must be resisted.
- Optimize algorithms first, implementation second.
- There's (almost) no excuse for assembly.
- Decide how fast is fast enough.
- Optimize the system before components.
- Never forget you are not running alone. There are libraries, background processes, the kernel, the CPU and the peripherals.

Use the right abstraction layer

- Use the right language for the job.
- Use the right tools for the job.
- Should you be writing in a high level language, low level language, object oriented, functional, logic programming or procedural language? which can express the problem domain best?
- Balance between the desire to learn new tools and techniques and the need to deliver on time and on budget.
- Make sure you have several hammers, but don’t disdain a hammer just because you’ve already used it.
- perl::open(), fopen(), open() or int 0x80?

Why coding style is important

- Your code does not belong to you. Someone will be maintaining and extending it long after you are gone.
- Many eyes make all bugs shallow, but if no one can understand your code, no one will look at it.
- There's plenty of room for personal expression even when using someone else's coding style.
- Avoid Hungarian notation. Leave the compiler's job to the compiler.
- Read Documentation/CodingStyle. Burn it if you want, but follow it.

Why you should always check for errors

- Anything that can go wrong, will. It's your responsibility to do the best job you can, and that includes handling errors. It's the thing you don't expect to fail that will.
- If there's nothing to be done when an error occures, your design is probably wrong. Nevertheless, leave a comment for the next programmer to read the code.
- The distance between an unchecked error and a security hole is very short.
- Always have sanity checks - you want to catch the error as close as possible to where it occures.
- Check printf() and close(). Be very wary of any string function that doesn't take an explicit length parameter.

The real cost of software development

- Maintenance costs far more than development.
- How often have people go on using that ten liner script you wrote year ago? how often did it become “production”? “mission critical”?
- There’s nothing more permanent than the temporary. Throw away code never is.
- Always write code with an eye to the people who will read it after you. A few years later it might be you, and you’ll appreciate every comment you left.
- Write for people, not compilers or processors.

Portability

- Portability doesn’t cost you, within reason, and can save you time, money and effort down the line.
- Today’s platform is tomorrow’s garbage. Platforms and operating environments change, and your code must change with them.
- Prefer cross platform toolkits and environments.
- If you must do something platform-specific, abstract it well in its own interchangeable component.
- Avoid multiple portability layers - a good one is enough.
- Don’t try to abstract that which is inherently different.

Questions, Answers, Comments?

What do **you** think every Linux programmer should know?