2024-04-30-Free Range Programming

That was then and now is now.

That was then and now is now. The computer-hardware world has gone through a pole shift. N is S and S is N. Today's computer hardware is 180 degrees out of phase with what was computer hardware in the 1950s. Standing on the shoulders of giants makes advances on the problems that the giants were trying to solve - in 1950, not in 2024.

It used to be thought to be too expensive to slap CPUs into every device. Yet, Apple+Adobe did it, with postscript. IBM had the right idea with channels, but abandoned the idea when it came to PCs. UNIX took a major step in the right direction (normalization) by making all devices look the same but blew it by implementing processes in C instead of using closures. Actors took a major step in the right direction, but, blew it by allowing step-wise implementations of Actors.

It used to be that industrial secrets were vitally necessary and open source was unheard of.

Today, though, CPUs are dirt cheap. We can afford to put one in each printer, one in each keyboard, one in each mouse, one in each display, etc. *Aside: even my Logitech MX Master 3S mouse has some sort of voodoo magic in it, I can feel it engaging and disengaging the physical ratchet system on the scroll wheel when certain conditions are met*.

We can finally use CPUs the way they were meant to be used - single-threaded, single-purpose - instead of band-aiding ad-hoc solutions onto the notion that we need to need to conserve CPUs and memory. *Aside: It's only been 50+ years, and, we ain't finished band-aiding yet. What's the rush, anyway?*

Continuing to research existing programming methodology based on 50-year old biases no longer addresses the realities of the capabilities of modern computers. At best, FoPoC (Future of the Past of Coding) addresses programming issues for single nodes in modern-day internet, robotics, etc.

This mindset is a disservice to non-programmers and to imagination.

Computers are a new medium, but, we have been forcing a paper-based, 2D, mentality onto them. And then, computers in 2024 are doubly-new: a new medium based on a new medium.

FoPoC-style programming (Future of the Past of Coding) isn't *the* way to program computers. It is but *one* way to program them. Deep-diving into only this one technique for 50+ years is too narrow a view, IMO.

What’s Wrong With Preemptive Operating Systems?

Preemption was invented to support function-based programming.

Programmers who wish to use function-based programming to develop assembler code, need preemption.

Non-programmers (aka “end-users”) want to solve their own kinds of problems and don’t *want* to know about function-based programming. Non-programmers don’t *need* preemption, they just want machines that augment their abilities to solve their own kinds of problems, and, they expect code sold to them by programmers to be bullet-proof[1].

Functions create ad-hoc blocking - a caller must suspend until the callee returns a value, regardless of how many other blocking functions the callee invokes.

Preemption

Code[2] might “run away” or loop endlessly or recur infinitely. That constitutes one kind of bug that is frequently encountered during development. Preemption gives developers a “big red button” that they can use to kill buggy programs and to regain control of their development computers.

Preemption, also, allows for time-sharing - a software version of changing game cartridges. Time-sharing was a serious concern in the 1950s when CPUs cost lots of money. Today, though, CPUs are dirt cheap, and, we no longer need to waste time and effort dealing with time-sharing[3]. Time-sharing, especially when conflated with memory sharing, comes with a lot of baggage and *gotchas*. A lot of so-called “computer science” would simply dissolve if we simply stopped using time-sharing muddied up with memory sharing.

Worse, preemption-based operating systems have been used as a crutch to allow programmers to deliver low-quality products to non-programmers. For example, CD (Continuous Delivery) enables programmers to ship broken designs (and broken code) to non-programmers, knowing that the non-programmers will *pay*

[1] Aka, “tested”, “Q/A-ed”, bug-free, etc.

[2] Any code, not just function-based code.

[3] But, we continue to use time-sharing, even though we can afford not to.

for the privilege of suffering with low-quality products, and, with reporting failures which can then be fixed and updated on-the-fly on a quarterly/monthly/daily basis.

Libraries and Device Drivers and LEGO®

Operating systems started out life as libraries of common, useful lumps of code that helped programmers produce products more quickly. Non-programmers, don't really need most of this extra code, but, they are forced to pay for it anyway[4]. Non-programmers would be just as happy to buy machines that did only what they needed, then to plug in new LEGO®-like blocks of code into their machines when they upgraded their hardware environments. Function-based code libraries are *not* LEGO®-like blocks, regardless of what words are used to describe them[5]. So, non-programmers have to put up with huge updates which contain stuff that they mostly don't need and/or that mostly hasn't changed.

A glaring example of the failure of this kind of approach was the Mars Pathfinder fiasco. Programmers believed in the religion of function-based programming and the overwhelming superiority of their arc of the covenant created from scripture by monks at the monastery of Green Hills. They shipped their code to Mars. The hardware arrived in fine shape, but the software promptly crashed. After programmers figured out what was wrong, they couldn't just send one LEGO® block to Mars, they had to send all of the software as one big lump containing only few tiny fixes embedded in it. Because the whole thing was so huge, they couldn't just send it to Mars via radio in a single short burst. They had to wait for windows of opportunity that were dictated by the orbit and rotation of Mars within our solar system.

This fiasco spawned the invention of a new band-aid in the world of function-based programming called "priority inheritance". The real problem was memory sharing and too much synchronization of things that should have been truly

[4] by paying for extra memory and disk space to store all of this extra stuff. (How much disk space do you really need to play, say, Tetris?)

[5] Function-based code contains too much hidden synchronization and unnecessary coupling, due to function-based thinking, to allow code units (functions) to be used as loosely-coupled LEGO® blocks.

asynchronous. The new band-aid, though, was the Advil® that dulled the pain without curing the cancer.

What’s Wrong With Mutual Multitasking Operating Systems?

Buggy apps can damage other apps[6].

Buggy apps can destroy the whole operating system and render the host computer useless.

Programmers insist on shipping buggy apps. Anyone who runs such buggy apps needs “protection”. They need red-button preemption and further complications - hardware and software - such as MMUs and extra software needed to support MMUs.

During development, programmers expect bugs, but, when applications are shipped, bugs are not expected.

Microsoft tried to sell version of MSDOS that used mutual multitasking, i.e. with no forced preemption. Some applications were buggy and caused computer crashes for end-users. This led to the generalization that mutual multitasking was untenable.

In fact, mutual multitasking is only untenable for protecting apps from other buggy apps, i.e. on end-user machines that run many apps on a time-sharing and memory-sharing basis.

Within a single app, a bug is just a bug, regardless of how it is caused. During development, it is not considered a surprise when one function clobbers another function, or when one function loops endlessly - this kind of thing is just considered to be a bug. During end-user use, though, this kind of problem is considered to be a disaster.

Game cartridges provide “enough” memory protection for end-user machines.

Conclusion

The baby got thrown out with the bathwater. Mutual multitasking ain’t necessarily bad *within* a single program, but, isn’t good enough to protect apps from other buggy apps.

[6] regardless whether the other apps are buggy or bug-free

Developers want big-red-button preemption.

End-users don't want complicated big-red-button preemption. They want bug-free apps, or, at worst, simple big-red-button preemption, like that provided by gaming systems and game cartridges.

Operating systems for end-users have different requirements from operating systems for developers. Forcing end-users to use developer-level operating systems just increases end-user costs and creates frustration through addition of complexity.

I believe that we will make faster progress when we recognize the difference and decouple development systems from end-user systems.

My argument boils down to:

1. Bring mutual multitasking back within single apps

2. Segregate operating systems for developers from operating systems for end-users, with the goal of not needing preemption at all in end-user systems. We're not there yet, end-user systems still need protection, because programmers are not up to the task of selling bug-free applications. *Aside: note that other disciplines can and do put guarantees on their products, e.g. chip manufacturers, automobile manufacturers[7], etc.*

[7] Relatively recently, automobile manufacturers have begun issuing recalls, but, they eat the cost of rework, and, do not issue recalls nearly as often as updates are sent out for software that uses Continuous Deployment techniques.

Appendix - See Also

See Also

References *https://guitarvydas.github.io/2024/01/06/References.html*
Blog *https://guitarvydas.github.io/*
Blog *https://publish.obsidian.md/programmingsimplicity*
Videos *https://www.youtube.com/@programmingsimplicity2980*
[see playlist “programming simplicity”]
Discord *https://discord.gg/Jjx62ypR (Everyone welcome to join)*
X *(Twitter) @paul_tarvydas*
More writing *(WIP): https://leanpub.com/u/paul-tarvydas*