Rebootless Kernel Updates

Srivatsa S. Bhat
VMware
srivatsa@csail.mit.edu

University of Washington
3 Dec 2018
Why are reboots undesirable?
Why are reboots undesirable?

Remember this? 😊
Why are reboots undesirable?
Why are reboots undesirable?

• Downtime:
  • Shutdown + Boot + App startup
  • Loss of state (eg: network connections)
  • Loss of results from long running processes
  • Unexpected complications
Why do kernel updates need rebooting?
Why do kernel updates need rebooting?

• Kernel manages hardware
  • Driver updates may require re-init of hardware
• Userspace programs need kernel services
  • System calls, signals, IPC etc
Why would you want live kernel updates?

- Minimal service disruption
- Apply security (CVE) fixes ASAP without scheduled maintenance windows
- Avoid application start-up times following OS updates
Adding kernel code on the fly

• Loadable kernel modules
Live kernel updates wishlist

• Ability to fix bugs/vulnerabilities in any part of the kernel (both core + module code)
• Small update latency (say, < 10 seconds)
• Ability to rollback on update failure
• Minimal programmer effort to tailor fixes to live update scenarios
Live update approaches for Linux

- Ksplice (MIT/Oracle)
- kGraft (SUSE)
- Kpatch (RedHat)
- Livepatch (Upstream) [ inspired by kGraft + kpatch ]
Ksplice

- Works at the level of object-code
- Function-level code replacement
- Latency: 0.7 milliseconds

Workflow:
- Generate binary replacement code using **pre-post differencing**
- Resolve symbols and verify safety using **run-pre matching**
- Use stop-machine for quiescence and perform code update
Ksplice: Generating repl code using pre-post differencing

Source patch

Post obj files

Extract functions that differed

Post code functions that differed

Processed post obj file

Kernel’s source code

Pre obj files

binary diff

List of functions that differ

Generic kernel module

Linker

Primary module
stop-machine framework in Linux

• Mechanism to run a given function on a given CPU with the rest of the machine stopped!
stop-machine framework in Linux

• Mechanism to run a given function on a given CPU with the rest of the machine stopped!

• “Stopper threads” created for each CPU during boot
  • Have highest priority in the system
  • Execute only in kernel mode
  • Typically in non-runnable state
stop-machine framework in Linux

• Mechanism to run a given function on a given CPU with the rest of the machine stopped!

• “Stopper threads” created for each CPU during boot
  • Have highest priority in the system
  • Execute only in kernel mode
  • Typically in non-runnable state

• stop-machine flow:
  • Mark all per-CPU stopper threads as runnable
  • Each stopper thread preempts userspace and hogs the CPU
    • Interrupts disabled on each CPU
  • Runs the requested function on the specified CPU
kGraft

- Replaces entire functions
- Uses ftrace to perform code patching
- Process by process transition to new kernel code:
  - Old vs New Universe
    - Band-Aid functions that understand both old and new layouts of data-structures
    - Uses fake signals to force “slow” processes to transition

- Needs special care to deal with:
  - Kernel threads
  - Interrupt handlers
kpatch

- Similar to kGraft for the most part
- A fundamental difference from kGraft:
  - Uses stop-machine for quiescence:
    - Examine kernel stacks of all processes with machine stopped.
    - If function not on any stack, proceed to patch.
    - Can’t patch functions always found on the stack
    - Eg: schedule()
Livepatch

- Best of both kGraft and kpatch
- Consistency model:
  - Supports both stop-machine and process-by-process transition
  - Stack traces used to be unreliable
    - Assembly routines may not setup stack frames
    - Fixed by ORC unwinder + objtool (stack validation)
Challenges for Livepatch / similar mechanisms

- Data-structure / semantic changes
  - (Partially) solved using shadow data-structures
- Changes to initialization routines
- Changes to static variables
- Dealing with compiler optimizations
- Patching hand-written assembly
- Handling changes in locking rules
- Patching modules that are not yet loaded
- Patching patched kernels
- Reverting live patches in case of failures
- ...
- Undecidability: In the general case, can’t prove that patch + state transition leads to valid state.
“Seamless” kernel updates

• Achieved via a combination of:
  • Kexec – exec a new kernel image from a running kernel
  • CRIU – Checkpoint Restore In Userspace

• Approach:
  • Similar to hibernation, but more generic
  • Checkpoint all userspace state using CRIU to disk
    • Kernel-version agnostic checkpointed state/format
  • Kexec into new kernel
  • Restore all userspace from checkpointed image
Kernel updates via kexec + CRIU

- Latency improvements
  - Incremental checkpoints
  - On-demand restore
  - Persistent Physical Pages
Kernel updates via kexec + CRIU

- Demo
  - https://gts3.org/pages/kup.html
PROTEOS

• Assumes microkernel design (eg: Minix)
• Performs process-level updates (unlike function-level updates)
• State quiescence (unlike function quiescence)
• State-transfer between old/new process versions
• Uses LLVM link-time pass for instrumentation
• Per-update state filters and interface filters
• Strictly event-driven process loops
• Structured design to handle many live update complications
• Supports a wider range of OS updates automatically than Livepatch-like approaches.
• Updating the microkernel itself might be challenging.
Revisiting the wishlist – Are we there yet?

• Ability to fix bugs/vulnerabilities in any part of the kernel (both core + module code)
• Minimal update latency (say, < 10 seconds)
• Ability to rollback on update failure
• Minimal programmer effort to tailor fixes to live update scenarios
References

• Ksplice: Automatic Rebootless Kernel Updates

• kGraft, kpatch and Livepatch:
  • https://lwn.net/Articles/596854/
  • https://lwn.net/Articles/597407/
  • https://lwn.net/Articles/734765/

• Kexec + CRIU: Instant OS Updates via Userspace Checkpoint-and-Restart

• PROTEOS: Safe and Automatic Live Update for Operating Systems