berrange.com

Daniel P. Berrangé » Blog Archive

Daniel Berrange

9-11 minutes

As a general rule, people using virtual machines have only one requirement when it comes to keyboard handling: any key they press should generate the same output in the host OS and the guest OS. Unfortunately, this is a surprisingly difficult requirement to satisfy. For a long time when using either Xen or KVM, to get workable keyboard handling it was necessary to configure the keymap in three places, 1. the VNC client OS, 2. the guest OS, 3. QEMU itself. The third item was a particular pain because it meant that a regular guest OS user would need administrative access to the host to change the keymap of their guest. Not good for delegation of control. In Fedora 11 we introduced a special VNC extension which allowed us to remove the need to configure keymaps in QEMU, so now it is merely necessary to configure the guest OS to match the client OS. One day when we get a general purpose guest OS agent, we might be able to automatically set the guest OS keymap to match client OS, whenever connecting via VNC, removing the last manual step. This post aims to give background on how keyboards work, what we done in VNC to improve the guest keyboard handling and what problems we still have.

Keyboard hardware scan codes

Between the time of pressing the physical key and text appearing on the screen, there are several steps in processing the input with data conversions along the way. The keyboard generates what are known as scan codes, a sequence of one or more bytes, which uniquely identifies the physical key and whether it is a make or break (press or release) event. Scan codes are invariant across all keyboards of the same type, regardless of what label is printed on the key. In the PC world, IBM defined the first scan code set with their IBM XT, followed later by AT scan codes and PS/2 scan codes. Other manufacturers adopted the IBM scan codes for their own products to ensure they worked out of the box. In the USB world, the HID specification defines the standard scan codes that manufacturers must use.

Operating system key codes

For operating systems wanted to support more than one different type of keyboard, scan codes are not a particularly good representation. They are also often unwieldly as a result of encoding both the key & its make/break state into the same byte(s). Thus operating systems typically define their own standard set of key codes, which is able to represent any possible keys on all known keyboards. They will also track the make/break state separately, now using press/release or up/down as terminology. Thus the first task of the keyboard driver is to convert from the hardware specific scan codes to the operating system specific key code. This is an **easily reverseable, lossless** mapping.

Display / toolkit key symbols & modifiers

Key codes still aren't a concept that is particularly useful for (most) applications, which would rather known what user's intended symbol was, rather than the physical key. Thus the display service (X11, Win32 GUI, etc) or application toolkit (GTK) define what are known as key symbols. To convert from key codes to key symbols, a key map is required for the language specific keyboard layout. The key map declares modifier keys (shift, alt, control, etc) and provides a list of key symbols that are associated with each key code. This mapping is **only reverseable** if you know the original key map. This is also **a lossy mapping,** because it is possible for several different key codes to map to the same key symbol.

Considering an end-to-end example, starting with the user pressing the key in row 4, column 2 which is labelled 'A' in a US layout, XT compatible keyboard. The operating system keyboard driver receives XT scan code 0x1e, which it converts to Linux key code 30 (KEY_A), Xorg server keyboard driver further converts to X11 key symbol 0x0061 (XK_a), GTK toolkit converts this to GDK key symbol 0x0061 (GDK_a), and finally the application displays the character 'a' in the text entry field on screen. There are actually a couple of conversions I've left out here, specifically how X11 gets the key codes from the OS and how X11 handles key codes internally, which I'll come back to another time.

The problem with virtualization

For 99.99% of applications all these different steps / conversions are no problem at all, because they are only interested in text entry. Virtualization, as ever, introduces fun new problems where ever it goes. The problem occurs at the interface between the host virtual desktop client and the hardware emulation. The virtual desktop may be a local fat client using a toolkit like SDL, or it may be a remote network client using a protocol like VNC, RFB or SPICE. In both cases, a naive virtual desktop client will be getting key symbols with their key events. The hardware emulation layer will usually want to provide something like a virtualizated PS/2 or USB keyboard. This implies that there needs to be a conversion from key symbols back to hardware specific scan codes. Remember a couple of paragraphs ago where it was noted that the key code -> key symbol conversion is **lossy**. That is a now a big problem. In the case of a network client it is even worse, because the virtualization host does not even know what language specific keymap was used.

Faced with these obstacles, the initial approach QEMU took was to just add a command line parameter ‘-k $KEYMAP’. Without this parameter set, it will assume the virtuall desktop client is using a US layout, otherwise it will use the specified keymap. There is still the problem that many key codes can map to the same key symbol. It is impossible to get around this problem – QEMU just has to pick one of the many possible reverse mappings & use it. This means hat, even if the user configures matching keymaps on their client, QEMU and the guest OS, there may be certain keys that will never work in the guest OS. There is also a burden for QEMU to maintain a set of keymaps for every language layout. This set is inevitably incomplete.

The solution with virtualization

Fortunately, there is a get out of jail free card available. When passing key events to an application, all graphical windowing systems will provide both the key symbol and layout independent key code. Remember from many paragraphs earlier that scan code -> key code conversion is **lossless and easily reverseable**. For local fat clients, the only stumbling block is knowing what key code set is in use. Windows and OS-X both define a standard virtual key set, but sadly X11 does not :-(Instead the key codes are specific to the X11 keyboard driver that is in use, with a Linux based Xorg this is typically ‘kbd’ or ‘evdev’. QEMU has to use heuristics on X11 to decide which key codes it is receiving, but at least once this is identified, the conversion back to scan codes is trivial.

For the VNC network client though, there was one additional stumbling block. The RFB protocol encoding of key events only includes the key symbol, not the key code.

So even though the VNC client has all the information the VNC server needs, there is no way to send it. Leading upto the development of Fedora 11, the upstream GTK-VNC and QEMU communities collaborated to define an official extension to the RFB protocol for an extended key event, that includes both key symbol and key code. Since every windowing system has its own set of key codes & the RFB needs to be platform independent, the protocol extension defined that the keycode set on the wire will be a special 32-bit encoding of the traditional XT scancodes. It is not a coincidence that QEMU already uses a 32-bit encoding of traditional XT scan codes internally :-) With this in place the RFB client, merely has to identify what operating system specific key codes it is receiving and then apply the suitable lossless mapping back to XT scancodes.

Considering an end-to-end example, starting with the user pressing the key in row 4, column 2 which is labelled 'A' in a US layout, XT compatible keyboard. The operating system keyboard driver receives XT scan code 0x1e, which it converts to Linux key code 30 (KEY_A), Xorg server keyboard driver further converts to X11 key symbol 0x0061 (XK_a) and an evdev key code, GTK toolkit converts the key symbol to GDK key symbol 0x0061 (GDK_a) but passes the evdev key code unchanged. The VNC client converts the evdev key code to the RFB key code and sends it to the QEMU VNC server along with the key symbol. The QEMU VNC server totally ignores the key symbol and does the no-op conversion from the RFB key code to the QEMU key code. The QEMU PS/2 emulation converts the QEMU keycode to either the XT scan code, AT scan code or PS/2 scan code, depending on how the guest OS has configured the keyboard controller. Finally the conversion mentioned much earlier takes place in the guest OS and the letter 'a' appears in a text field in the guest. The important bit to note is that although the key symbol was present, it was never used in the host OS or remote network client at any point. The only conversions performed were between scan codes and key codes all of which were lossless. The user is able to generate all the same key sequences in the guest OS as they can in the host OS. The user is happy their keyboard works as expected; the virtualization developer is happy at lack of bug reports about broken keyboard handling.