OSDEV.ORG
The Place to Start for Operating System Developers

BEST WAY TO DO I/O PORT PROTECTION

Re: Best way to do I/O port protection

by **Brendan** » Fri May 29, 2009 4:07 am

Brendan
Member
Posts: 8561
Joined: Sat Jan 15, 2005 12:00 am
Location: At his keyboard!
Contact:

Hi,

> **NickJohnson wrote:**
> Would it also be possible to just set the IOPL to 3 for driver processes exclusively, and stay out of the whole segmentation thing altogether? Would that be safe?

Imagine a "device driver protection" scale that goes from 0 (least protection) to 10 (most protection).

I'd give older versions of Windows and DOS a rating of 0. I'm not too sure about newer 64-bit versions of Windows that require signed drivers (I'm not convinced it makes a significant difference).

I'd give Linux, FreeBSD, OpenBSD, etc a rating of 1 (because people can see the code they're meant to be running), but then I'd subtract half a point (because it's a lot easier for other people to change the code that people thought they were running); which leaves a rating of 0.5 out of 10. 😉

If an OS sets IOPL to 3 for device drivers then this doesn't really do anything to improve the OS's "device driver protection" rating - a buggy or malicious device driver can still trash anything it likes (although running device drivers at CPL=3 is probably worth a point or 2, depending on how that's done).

To get a rating of 10 you'd need to follow the principle of least privilege. This includes:

- using system calls and/or emulation for I/O port access (to prevent the device driver from accessing I/O ports that have nothing to do with the device, including certain bits in some I/O ports)
- prevent the device driver from accessing any memory mapped I/O areas that have nothing to do with the device
- prevent the device driver from messing up IRQ priorities and IRQ delivery
- preventing the driver from accessing any RAM that it shouldn't have permission to access
- using the IOMMU built into modern hardware to protect the OS from dodgy DMA and bus mastering
- making sure a device driver can't hog CPU time
- preventing device drivers from accessing anything else that shouldn't be necessary (including files, network services, the GUI, etc). For example, a keyboard device driver shouldn't be able to open a TCP/IP connection to a remote server; and an ethernet card driver shouldn't be able to use the GUI to create a window that just happens to look like the login screen.

Cheers,

Brendan

For all things; perfection is, and will always remain, impossible to achieve in practice. However; by striving for perfection we create things that are as perfect as practically possible. Let the pursuit of perfection be our guide.

Re: Best way to do I/O port protection

by **uglyoldbob** » Mon Jun 01, 2009 8:36 am

> **Brendan wrote:**
> Hi,
>
> There's at least 3 methods:
>
> - ...
>   - emulate the I/O instructions in the general protection fault handler (e.g. when any "CPL != 0" code tries to access an I/O port it causes a GPF and the general protection fault hander checks if the cause was an I/O port access and determines if access should be allowed, and then does the I/O port access itself)
>   ...
>
>   Emulating the I/O port instructions in the general protection fault handler is the most powerful way, because it lets you protect individual bits in some I/O ports (for e.g. giving a CMOS driver access I/O port 0x70 without letting it change bit 7 and enable/disable NMI; or letting a keyboard driver access I/O port 0x64 without letting it change the A20 and reset bits). Also, it allows you to use virtual I/O ports - for example, tell the device driver that it's device uses I/O ports from 0x0000 to 0x0008, and then when the driver accesses I/O port 0x0001 you actually access I/O port 0x0123 instead (which could make it easier to write devices drivers because you could hard-code I/O ports instead of doing "mov dx,[IOportBase]" everywhere). It can be used for device virtualization (e.g. have an entirely fictional floppy drive controller, where a completely normal device driver can't tell if it's a virtual device or a real device). Lastly it's useful for debugging purposes (e.g. create a log of all I/O port accesses make by a device driver).
>
>   ...
>
>   Cheers,
>
>   Brendan

I haven't done enough research to know if this is possible, but would using this allow DOS programs to be more compatible with a 32-bit kernel (and with the proper precautions taken in the I/O port redirection code of course). You would still need to offer/emulate the same services and whatnot that dos provided. Of course this wouldn't be as good as running a computer emulator (say qemu) to provide this support (right?).

I have an 80386SX 20MHz 2MB RAM.
It is my testbed platform. Only has the 3.5" and 5.25" floppy drives.

uglyoldbob
Member

Posts: 62
Joined: Tue Feb 13, 2007 10:46 am

Re: Best way to do I/O port protection

by **Brendan** » Mon Jun 01, 2009 10:34 am

Hi,

> **uglyoldbob wrote:**
>
> > **Brendan wrote:**
> > emulate the I/O instructions in the general protection fault handler
>
> I haven't done enough research to know if this is possible, but would using this allow DOS programs to be more compatible with a 32-bit kernel (and with the proper precautions taken in the I/O port redirection code of course).

Brendan
Member

Posts: 8561
Joined: Sat Jan 15, 2005 12:00 am
Location: At his keyboard!
Contact:

It's probably the only way to run DOS software under a 32-bit OS, as you'd want to emulate all I/O ports (the DOS software never accesses any real I/O port), even if you implement full virtual8086 mode.

> **uglyoldbob wrote:**
> You would still need to offer/emulate the same services and whatnot that dos provided. Of course this wouldn't be as good as running a computer emulator (say qemu) to provide this support (right?).

In either case you'd need to emulate all the hardware (as DOS software has a tendency to bypass DOS and program the devices itself). Emulating a full computer would be more flexible, but more hassle (e.g. the end-user would need a licensed copy of DOS to run within the emulator). If you only wanted to run DOS software you could implement DOS services with host code (which would execute faster, and would mean more free virtual RAM below 0x000100000 left for applications), and you could skip half the BIOS (initialization/POST) and skip boot code too.

Cheers,

Brendan

For all things; perfection is, and will always remain, impossible to achieve in practice. However; by striving for perfection we create things that are as perfect as practically possible. Let the pursuit of perfection be our guide.

Re: Best way to do I/O port protection

by **NickJohnson** » Wed Jun 03, 2009 7:48 pm

NickJohnson
Member
Posts: 1249
Joined: Tue Mar 24, 2009 8:11 pm
Location: Sunnyvale, California

> **Brendan wrote:**
> Hi,
>
> > **NickJohnson wrote:**
> > Would it also be possible to just set the IOPL to 3 for driver processes exclusively, and stay out of the whole segmentation thing altogether? Would that be safe?
>
> Imagine a "device driver protection" scale that goes from 0 (least protection) to 10 (most protection).
>
> I'd give older versions of Windows and DOS a rating of 0. I'm not too sure about newer 64-bit versions of Windows that require signed drivers (I'm not convinced it makes a significant difference).
>
> I'd give Linux, FreeBSD, OpenBSD, etc a rating of 1 (because people can see the code they're meant to be running), but then I'd subtract half a point (because it's a lot easier for other people to change the code that people thought they were running); which leaves a rating of 0.5 out of 10. 😉
>
> If an OS sets IOPL to 3 for device drivers then this doesn't really do anything to improve the OS's "device driver protection" rating - a buggy or malicious device driver can still trash anything it likes (although running device drivers at CPL=3 is probably worth a point or 2, depending on how that's done).
>
> To get a rating of 10 you'd need to follow the principle of least privilege. This includes:
>
> - using system calls and/or emulation for I/O port access (to prevent the device driver from accessing I/O ports that have nothing to do with the device, including certain bits in some I/O ports)
> - prevent the device driver from accessing any memory mapped I/O areas that have nothing to do with the device
> - prevent the device driver from messing up IRQ priorities and IRQ delivery
> - preventing the driver from accessing any RAM that it shouldn't have permission to access

> - using the IOMMU built into modern hardware to protect the OS from dodgy DMA and bus mastering
> - making sure a device driver can't hog CPU time
> - preventing device drivers from accessing anything else that shouldn't be necessary (including files, network services, the GUI, etc). For example, a keyboard device driver shouldn't be able to open a TCP/IP connection to a remote server; and an ethernet card driver shouldn't be able to use the GUI to create a window that just happens to look like the login screen.
>
> Cheers,
>
> Brendan

I implemented the whole IOPL = 3 idea, and it works fine so far. But I do see your point.

So, how's this plan: Do all I/O access via system calls, but have each process structure have a base and limit for which ports the process can access. Ports are accessed relative to the base so you get the benefit of not requiring the driver know which ports it's using. Any process can strictly reduce its I/O range, so you could use sort of "shell-style" techniques to protect things. E.g. you fork(), reduce I/O range properly, then exec() the driver code.

Using my existing ring-based permission system, the driver processes are only able to access (i.e. map to their address space manually) the memory used by processes in strictly lower privilege levels than them. At least this amount of control is required for the way I do IPC, so it violates at least one of your rules. A similar base and limit value pair is used for the physical memory mappable by each driver, which would usually be used for mmapped I/O and DMA. The processes could, again, only make that range smaller at any time.

I would also make some changes to IRQ handling to make sure drivers don't mess each other up (which is quite possible in the current architecture). I'd say it would get pretty close to the optimum in your scale.

Re: Best way to do I/O port protection

by **Brendan** » Thu Jun 04, 2009 5:34 am

Brendan
Member
Posts: 8561
Joined: Sat Jan 15, 2005 12:00 am
Location: At his keyboard!
Contact:

Hi,

> **NickJohnson wrote:**
> I implemented the whole IOPL = 3 idea, and it works fine so far. But I do see your point.
>
> So, how's this plan: Do all I/O access via system calls, but have each process structure have a base and limit for which ports the process can access. Ports are accessed relative to the base so you get the benefit of not requiring the driver know which ports it's using.

It's a little more complex than that because some devices use several ranges of I/O ports. For example, the primary hard disk controller might use I/O ports 0x01F0h to 0x01F7 and I/O port 0x03F6, and would therefore need two I/O port ranges rather than just one. For the sake of completeness I'd support up to six I/O port ranges per process (or per device) as PCI configuration space has 6 BARs, where all BARs could (in theory) refer to different ranges of I/O ports.

> **NickJohnson wrote:**
> Any process can strictly reduce its I/O range, so you could use sort of "shell-style" techniques to protect things. E.g. you fork(), reduce I/O range properly, then exec() the driver code.

Can you think of any situation where it makes sense for a device driver to let some other piece of software have direct access to the device's I/O ports?

The way I see it is that you'll have something that scans the PCI bus (e.g. a "device manager"); and for each device found during this scan the device manager starts a device driver, tells the kernel to allow the device driver to access certain I/O ports, and tells the device driver which I/O ports it's allowed to access. In this case the device manager is the only piece of code that controls access to I/O ports, and just because a device driver is granted access to some I/O ports doesn't mean it can let any other process have access to any of those I/O ports.

> **NickJohnson wrote:**
> Using my existing ring-based permission system, the driver processes are only able to access (i.e. map to their address space manually) the memory used by processes in strictly lower privilege levels than them. At least this amount of control is required for the way I do IPC, so it violates at least one of your rules. A similar base and limit value pair is used for the physical memory mappable by each driver, which would usually be used for mmapped I/O and DMA. The processes could, again, only make that range smaller at any time.

Can you think of any situation where it makes sense for a device driver to let some other piece of software have direct access to any of the device's memory mapped areas?

The way I see it is that you'll have something that scans the PCI bus (e.g. a "device manager"); and for each device found during this scan the device manager starts a device driver, tells the kernel to allow the device driver to access certain physical memory areas, and tells the device driver which physical memory areas it's allowed to access. In this case the device manager is the only piece of code that controls access to these physical memory areas, and just because a device driver is granted access to some physical memory areas doesn't mean it can let any other process have access to any of those physical memory areas.

Note: You may be thinking that for video it might be a good idea to let some processes (e.g. a GUI) have direct access to display memory. IMHO it's not a good idea (and it only seems like a good idea if your video driver interface is a lame piece of crap, where things like page flipping and 2D/3D acceleration can never be used).

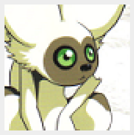
Cheers,

Brendan

For all things; perfection is, and will always remain, impossible to achieve in practice. However; by striving for perfection we create things that are as perfect as practically possible. Let the pursuit of perfection be our guide.

Re: Best way to do I/O port protection

by **Combuster** » Thu Jun 04, 2009 10:04 am



Combuster
Member
★★★★★

Posts: 9301
Joined: Wed Oct 18, 2006 3:45 am
Libera.chat IRC: [com]buster
Location: On the balcony, where I can actually keep 1½m distance
Contact: 💬

> **Brendan wrote:**
> It's a little more complex than that because some devices use several ranges of I/O ports. For example, the primary hard disk controller might use I/O ports 0x01F0h to 0x01F7 and I/O port 0x03F6, and would therefore need two I/O port ranges rather than just one. For the sake of completeness I'd support up to six I/O port ranges per process (or per device) as PCI configuration space has 6 BARs, where all BARs could (in theory) refer to different ranges of I/O ports.

I'd add sparse port sets to the list. Many ISA devices use ports in the style of 0x??nn, where n is constant, and only the top x bits change. To make matters worse, some PCI devices do the exact same thing, and do not report that in the BARs. (and can therefore thoroughly screw up your system when something locates a set of ports and they start overlapping. (I even know of a well known VM that emulates one such card)

> Can you think of any situation where it makes sense for a device driver to let some other piece of software have direct access to the device's I/O ports?

To prevent repeated context switches just to push some polygons through a hardware FIFO? (especially when the device doesn't have DMA access to command buffers and you can only have two commands queued at any one time)

Edit: and before you come with the security argument - consider the case where the app has exclusive use of the screen device, and you give it only access to the command fifo (and not the CRTC regs to blow up the monitor, or DMA regs to screw with your memory, and so on)

> Can you think of any situation where it makes sense for a device driver to let some other piece of software have direct access to any of the device's memory mapped areas?

cough linux nvidia opengl *cough*

> Note: You may be thinking that for video it might be a good idea to let some processes (e.g. a GUI) have direct access to display memory. IMHO it's not a good idea (and it only seems like a good idea if your video driver interface is a lame piece of crap, where things like page flipping and 2D/3D acceleration can never be used).

Giving access to FB memory does not imply an absence of hardware acceleration. In fact I'd expect it to be more useful to pop in a page so that you can write your textures to video memory, after which you unpage it again. Worst case its as slow as making a copy, giving it to the app to deal with, then copying it back to video memory.

"Certainly avoid yourself. He is a newbie and might not realize it. You'll hate his code deeply a few years down the road." - Sortie
[My OS] [VDisk/SFS]

Re: Best way to do I/O port protection

by **Brendan** » Thu Jun 04, 2009 12:16 pm

Brendan
Member
Posts: 8561
Joined: Sat Jan 15, 2005 12:00 am
Location: At his keyboard!
Contact:

Hi,

> **Combuster wrote:**
>
> > **Brendan wrote:**
> > It's a little more complex than that because some devices use several ranges of I/O ports. For example, the primary hard disk controller might use I/O ports 0x01F0h to 0x01F7 and I/O port 0x03F6, and would therefore need two I/O port ranges rather than just one. For the sake of completeness I'd support up to six I/O port ranges per process (or per device) as PCI configuration space has 6 BARs, where all BARs could (in theory) refer to different ranges of I/O ports.
>
> I'd add sparse port sets to the list. Many ISA devices use ports in the style of 0x??nn, where n is constant, and only the top x bits change. To make matters worse, some PCI devices do the exact same thing, and do not report that in the BARs. (and can therefore thoroughly screw up your system when something locates a set of ports and they start overlapping. (I even know of a well known VM that emulates one such card)

Some old ISA devices only decode the least significant 10-bits of an I/O port address, so that (for e.g.) if the device uses I/O ports from 0x0200 to 0x0208 you get annoying shadows (I/O ports 0x0600 to 0x0608, from 0x0A00 to 0x0A08, from 0x0E00 to 0x0E08, etc). In this case the device driver itself doesn't need access to the shadow ranges (e.g. it would only need to use I/O ports from 0x0200 to 0x0208).

Also, for PCI devices that support a legacy/ISA interface and a native interface, you'd expect that a native device driver written specifically for the device would use the device's native interface rather than worrying about the legacy/ISA interface.

> **Combuster wrote:**
>
> > Can you think of any situation where it makes sense for a device driver to let some other piece of software have direct access to the device's I/O ports?

> To prevent repeated context switches just to push some polygons through a hardware FIFO? (especially when the device doesn't have DMA access to command buffers and you can only have two commands queued at any one time)
>
> Edit: and before you come with the security argument - consider the case where the app has exclusive use of the screen device, and you give it only access to the command fifo (and not the CRTC regs to blow up the monitor, or DMA regs to screw with your memory, and so on)

One of the main goals of having device drivers is to prevent the need for applications to have lots and lots of code to support lots and lots of devices itself. Allowing a process to access a video card's command queue tends to defeat the purpose of having a device driver.

To avoid repeated context switches (e.g. for polygon drawing), a process can give the video driver an entire list of polygons that need to be drawn, then do one context switch to the video driver (which draws all polygons in the list).

> **Combuster wrote:**
>
> > Can you think of any situation where it makes sense for a device driver to let some other piece of software have direct access to any of the device's memory mapped areas?
>
> *cough* linux nvidia opengl *cough*

Hehe - Nvidia's Linux driver is a workaround rather than something I'd consider ideal.

> **Combuster wrote:**
>
> > Note: You may be thinking that for video it might be a good idea to let some processes (e.g. a GUI) have direct access to display memory. IMHO it's not a good idea (and it only seems like a good idea if your video driver interface is a lame piece of crap, where things like page flipping and 2D/3D acceleration can never be used).
>
> Giving access to FB memory does not imply an absence of hardware acceleration. In fact I'd expect it to be more useful to pop in a page so that you can write your textures to video memory, after which you unpage it again. Worst case its as slow as making a copy, giving it to the app to deal with, then copying it back to video memory.

The main reason for giving a process access to a FB memory is to allow that process to write directly to the FB memory; which is probably the slowest way to do things in all modern video cards. It'd be better to have no device specific code in the application, where the application asks the video driver to do any device specific operations (like using bus mastering to copy texture data from RAM into display memory, or mapping that page into the video card's address space, or whatever other method might be more appropriate for the specific device). Then there's the problem of page flipping - if you let a process map the FB into it's address space, then how do you synchronize the page flipping so that the application knows which area is the front-buffer and which area is the back-buffer (without causing race conditions)?

Of course I'd go a step further - I'm planning to use resolution independence and colour depth independence, so that normal processes don't need to care about these device specific (video mode specific?) details either.

Cheers,

Brendan

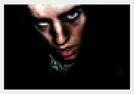
For all things; perfection is, and will always remain, impossible to achieve in practice. However; by striving for perfection we create things that are as perfect as practically possible. Let the pursuit of perfection be our guide.

Re: Best way to do I/O port protection

by **01000101** » Thu Jun 04, 2009 3:28 pm

Using VMX you can specify two 4k memory regions that act as a I/O bitmap (for 0-0xFFFF). Even if you didn't want to set up guest virtual machines, you could just enter the host state and continue executing instructions just as before (but now it would be monitored) and every time an I/O port is addressed that is masked it would exit the virtual state temporarily to allow the true host to modify the I/O request or use internal functions to ensure security. Just a thought.

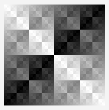
Website: https://joscor.com


01000101
Member
☆☆☆☆☆

Posts: 1599
Joined: Fri Jun 22, 2007 12:47 pm
Contact:

Re: Best way to do I/O port protection

by **NickJohnson** » Thu Jun 04, 2009 3:38 pm


NickJohnson
Member
☆☆☆☆☆

Posts: 1249
Joined: Tue Mar 24, 2009 8:11 pm
Location: Sunnyvale, California

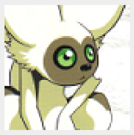
> **Brendan wrote:**
> Hi,
>
> > **Combuster wrote:**
> >
> > > Can you think of any situation where it makes sense for a device driver to let some other piece of software have direct access to the device's I/O ports?
> >
> > To prevent repeated context switches just to push some polygons through a hardware FIFO? (especially when the device doesn't have DMA access to command buffers and you can only have two commands queued at any one time)
> >
> > Edit: and before you come with the security argument - consider the case where the app has exclusive use of the screen device, and you give it only access to the command fifo (and not the CRTC regs to blow up the monitor, or DMA regs to screw with your memory, and so on)
>
> One of the main goals of having device drivers is to prevent the need for applications to have lots and lots of code to support lots and lots of devices itself. Allowing a process to access a video card's command queue tends to defeat the purpose of having a device driver.
>
> To avoid repeated context switches (e.g. for polygon drawing), a process can give the video driver an entire list of polygons that need to be drawn, then do one context switch to the video driver (which draws all polygons in the list).
>
> > **Combuster wrote:**
> >
> > > Can you think of any situation where it makes sense for a device driver to let some other piece of software have direct access to any of the device's memory mapped areas?
> >
> > *cough* linux nvidia opengl *cough*
>
> Hehe - Nvidia's Linux driver is a workaround rather than something I'd consider ideal.
>
> > **Combuster wrote:**
> >
> > > Note: You may be thinking that for video it might be a good idea to let some processes (e.g. a GUI) have direct access to display memory. IMHO it's not a good idea (and it only seems like a good idea if your video driver interface is a lame piece of crap, where things like page flipping and 2D/3D acceleration can never be used).
> >
> > Giving access to FB memory does not imply an absence of hardware acceleration. In fact I'd expect it to be more useful to pop in a page so that you can write your textures to video

> > memory, after which you unpage it again. Worst case its as slow as making a copy, giving it to the app to deal with, then copying it back to video memory.
>
> The main reason for giving a process access to a FB memory is to allow that process to write directly to the FB memory; which is probably the slowest way to do things in all modern video cards. It'd be better to have no device specific code in the application, where the application asks the video driver to do any device specific operations (like using bus mastering to copy texture data from RAM into display memory, or mapping that page into the video card's address space, or whatever other method might be more appropriate for the specific device). Then there's the problem of page flipping - if you let a process map the FB into it's address space, then how do you synchronize the page flipping so that the application knows which area is the front-buffer and which area is the back-buffer (without causing race conditions)?
>
> Of course I'd go a step further - I'm planning to use resolution independence and colour depth independence, so that normal processes don't need to care about these device specific (video mode specific?) details either.

From this it seems like you don't realize that I'm making a microkernel - the programs that I'm giving memory and port access to *are* the device drivers. They're supposed to have full I/O access to the devices they govern; I was just trying to figure out a way to make sure the device drivers have less chance of breaking things. I wasn't planning on having the GUI engine be part of the video driver either.

Re: Best way to do I/O port protection

by **Combuster** » Thu Jun 04, 2009 5:24 pm



Combuster
Member

Posts: 9301
Joined: Wed Oct 18, 2006 3:45 am
Libera.chat IRC: [com]buster
Location: On the balcony, where I can actually keep 1½m distance
Contact:

> **Brendan wrote:**
> Hi,
>
> > **Combuster wrote:**
> >
> > > **Brendan wrote:**
> > > It's a little more complex than that because some devices use several ranges of I/O ports. For example, the primary hard disk controller might use I/O ports 0x01F0h to 0x01F7 and I/O port 0x03F6, and would therefore need two I/O port ranges rather than just one. For the sake of completeness I'd support up to six I/O port ranges per process (or per device) as PCI configuration space has 6 BARs, where all BARs could (in theory) refer to different ranges of I/O ports.
> >
> > I'd add sparse port sets to the list. Many ISA devices use ports in the style of 0x??nn, where n is constant, and only the top x bits change. To make matters worse, some PCI devices do the exact same thing, and do not report that in the BARs. (and can therefore thoroughly screw up your system when something locates a set of ports and they start overlapping. (I even know of a well known VM that emulates one such card)
>
> Some old ISA devices only decode the least significant 10-bits of an I/O port address, so that (for e.g.) if the device uses I/O ports from 0x0200 to 0x0208 you get annoying shadows (I/O ports 0x0600 to 0x0608, from 0x0A00 to 0x0A08, from 0x0E00 to 0x0E08, etc). In this case the device driver itself doesn't need access to the shadow ranges (e.g. it would only need to use I/O ports from 0x0200 to 0x0208).
>
> Also, for PCI devices that support a legacy/ISA interface and a native interface, you'd expect that a native device driver written specifically for the device would use the device's native interface rather than worrying about the legacy/ISA interface.

I know those, and they do not form a problem with your system. However there is the set of ISA (compatible) cards that decode the full address, and use the top bits as offset while the bottom bits are constant (most likely a compensation for having register files that cooperate with the mirrored register files you mentioned)
As an example: S3 Trio/Virge series of cards. MMIO is barely covered, and in fact not enough to

make it work. And even the X.org drivers use the sparse register file for some cards that have PCI versions (including the one that VirtualPC emulates)

> > **Combuster wrote:**
> >
> > > Can you think of any situation where it makes sense for a device driver to let some other piece of software have direct access to the device's I/O ports?
> >
> > To prevent repeated context switches just to push some polygons through a hardware FIFO? (especially when the device doesn't have DMA access to command buffers and you can only have two commands queued at any one time)
> >
> > Edit: and before you come with the security argument - consider the case where the app has exclusive use of the screen device, and you give it only access to the command fifo (and not the CRTC regs to blow up the monitor, or DMA regs to screw with your memory, and so on)
>
> One of the main goals of having device drivers is to prevent the need for applications to have lots and lots of code to support lots and lots of devices itself. Allowing a process to access a video card's command queue tends to defeat the purpose of having a device driver.
>
> To avoid repeated context switches (e.g. for polygon drawing), a process can give the video driver an entire list of polygons that need to be drawn, then do one context switch to the video driver (which draws all polygons in the list).

In my design, the driver will be able to download itself into a process, so that those functions can be called directly, as an optimisation sacrificing security over speed. (and if you take care selecting the proper subset of controls, security may not even be an issue)

And there's still the FIFO problem you have yet to address: (for the demonstration, the fifo can hold one command, IRL the fifo holds ~2 while a third is executed):

CODE: SELECT ALL

```
Naive approach:     Batch processing:       Direct access:
create               create                  create
   -switch-          create                  render
        render       create                  create
   -switch-             -switch-             render
create                       render          create
   -switch-                  -wait-          render
        render               render          create
   -switch-                  -wait-          render
create                       render          create
       .                -switch-                .
       .                     .                  .
```

True, that doesn't hold with the recent cards that have comparatively huge FIFOs and/or DMA commands.

> Of course I'd go a step further - I'm planning to use resolution independence and colour depth independence, so that normal processes don't need to care about these device specific (video mode specific?) details either.

Exokernel here.
feeds output to /dev/blasphemy 😈
ducks and runs

"Certainly avoid yourself. He is a newbie and might not realize it. You'll hate his code deeply a few years down the road." - Sortie
[My OS] [VDisk/SFS]

Re: Best way to do I/O port protection

by **Brendan** » Fri Jun 05, 2009 9:21 am

Brendan
Member

Posts: 8561
Joined: Sat Jan 15, 2005 12:00 am
Location: At his keyboard!
Contact:

Hi,

> **NickJohnson wrote:**
> From this it seems like you don't realize that I'm making a microkernel - the programs that I'm giving memory and port access to *are* the device drivers. They're supposed to have full I/O access to the devices they govern; I was just trying to figure out a way to make sure the device drivers have less chance of breaking things. I wasn't planning on having the GUI engine be part of the video driver either.

Yes, but your previous comments game me the impression that any process that has been granted access to any I/O ports would be able to give permission to access any of it's I/O ports to a new child process (e.g. a video driver could give some other process access to some of the video card's I/O ports).

What I'm suggesting is that device drivers themselves shouldn't be able to "fork(), reduce I/O range, then exec() some other arbitrary process".

Cheers,

Brendan

For all things; perfection is, and will always remain, impossible to achieve in practice. However; by striving for perfection we create things that are as perfect as practically possible. Let the pursuit of perfection be our guide.

Re: Best way to do I/O port protection

by **Brendan** » Fri Jun 05, 2009 11:07 am

Brendan
Member

Posts: 8561
Joined: Sat Jan 15, 2005 12:00 am
Location: At his keyboard!
Contact:

Hi,

> **Combuster wrote:**
>
> > **Brendan wrote:**
> >
> > > **Combuster wrote:**
> > > I'd add sparse port sets to the list. Many ISA devices use ports in the style of 0x??nn, where n is constant, and only the top x bits change. To make matters worse, some PCI devices do the exact same thing, and do not report that in the BARs. (and can therefore thoroughly screw up your system when something locates a set of ports and they start overlapping. (I even know of a well known VM that emulates one such card)
> >
> > Some old ISA devices only decode the least significant 10-bits of an I/O port address, so that (for e.g.) if the device uses I/O ports from 0x0200 to 0x0208 you get annoying shadows (I/O ports 0x0600 to 0x0608, from 0x0A00 to 0x0A08, from 0x0E00 to 0x0E08, etc). In this case the device driver itself doesn't need access to the shadow ranges (e.g. it would only need to use I/O ports from 0x0200 to 0x0208).
> >
> > Also, for PCI devices that support a legacy/ISA interface and a native interface, you'd expect that a native device driver written specifically for the device would use the device's native interface rather than worrying about the legacy/ISA interface.
>
> I know those, and they do not form a problem with your system. However there is the set of ISA (compatible) cards that decode the full address, and use the top bits as offset while the bottom bits are constant (most likely a compensation for having register files that cooperate with the mirrored register files you mentioned)
> As an example: S3 Trio/Virge series of cards. MMIO is barely covered, and in fact not enough to make it work. And even the X.org drivers use the sparse register file for some cards that have PCI versions (including the one that VirtualPC emulates)

Oh my - unfortunately you're right.

I've got a book here ("Programmer's Guide to the EGA, VGA, and Super VGA Cards (3rd Edition)") which has an (unfortunately partial) description of S3 video card registers. This book says that setting bit 4 in CR53 (I/O port 0x03D4, index 0x53) enables memory mapped I/O mode, where the extended registers become memory mapped like this:

- I/O port 0x8?E8 accessable via. memory address 0x000A8?E8
  I/O port 0x9?E8 accessable via. memory address 0x000A9?E8
  I/O port 0xA?E8 accessable via. memory address 0x000AA?E8
  I/O port 0xB?E8 accessable via. memory address 0x000AB?E8

In addition, pixel data can be written to the memory area from 0x0000A0000 to 0x0000A7FFF (instead of using the "Pixel Data Transfer Registers" - I/O ports 0xE2E8 and 0xE2EA)

It also says that "I/O port 0xBEE8 index 0x0F, Read Register Select Register" is the only extended register that can't be memory mapped, and that memory mapped registers are "write-only". This shouldn't be a problem though (keep a copy of video card's registers in RAM so that you can read from RAM instead of reading from the device's registers directly).

It seems to me that the problem is the poor quality of available documentation for these card/s, rather than the card's actual abilities.

A more significant problem with these cards (and similar cards) is that it's impossible for the BIOS to know which I/O ports the device is using, and therefore impossible for the BIOS to avoid assigning these I/O ports to a different PCI card (e.g. a sound card); and likewise it's impossible for the BIOS to correctly configure any PCI to PCI bridges to forward I/O port access to the correct PCI bus segment (to ensure I/O port accesses that are meant to be forwarded to the video card actually are forwarded to the video card). Basically it breaks all of the "plug and play" principles of the PCI bus.

Because of this, IMHO memory mapping the extended registers is the only way this video card can be used safely (and even then you'd need to make sure that the legacy VGA I/O ports and legacy VGA "display memory window" addresses are forwarded to the video card).

Of course you already knew this; however, I still don't think that such an appauling mess justifies sparse port sets - if sparse I/O ports are necessary for a specific device, then that device should go on the OS's "currently not supported by this OS, and never will be supported this OS" hardware compatability blacklist (where if the device is detected during boot, the OS panics with a "*Rip that scanky piece of crud out of your computer NOW!*" error message).. 😉

> **Combuster wrote:**
>
> > > **Combuster wrote:**
> > > To prevent repeated context switches just to push some polygons through a hardware FIFO? (especially when the device doesn't have DMA access to command buffers and you can only have two commands queued at any one time)
> > >
> > > Edit: and before you come with the security argument - consider the case where the app has exclusive use of the screen device, and you give it only access to the command fifo (and not the CRTC regs to blow up the monitor, or DMA regs to screw with your memory, and so on)
> >
> > One of the main goals of having device drivers is to prevent the need for applications to have lots and lots of code to support lots and lots of devices itself. Allowing a process to access a video card's command queue tends to defeat the purpose of having a device driver.
> >
> > To avoid repeated context switches (e.g. for polygon drawing), a process can give the

> > video driver an entire list of polygons that need to be drawn, then do one context switch to the video driver (which draws all polygons in the list).
>
> In my design, the driver will be able to download itself into a process, so that those functions can be called directly, as an optimisation sacrificing security over speed. (and if you take care selecting the proper subset of controls, security may not even be an issue)
>
> And there's still the FIFO problem you have yet to address: (for the demonstration, the fifo can hold one command, IRL the fifo holds ~2 while a third is executed):
>
> **CODE: SELECT ALL**
>
> ```
> Naive approach:     Batch processing:       Direct access:
> create               create                  create
>    -switch-          create                  render
>         render       create                  create
>    -switch-             -switch-             render
> create                       render          create
>    -switch-                  -wait-          render
>         render               render          create
>    -switch-                  -wait-          render
> create                       render          create
>        .                -switch-                 .
>        .                    .                    .
>        .                    .                    .
> ```
>
> True, that doesn't hold with the recent cards that have comparatively huge FIFOs and/or DMA commands.

CODE: SELECT ALL

```
Elite processing:
  create
  render (using GPU)
  create
  render (using GPU)
  create
  render (using CPU - FIFO full)
  create
  render (using GPU)
  create
  render (using CPU - FIFO full again)
```

Better performance than just using the GPU alone (or the CPU alone), with no unnecessary context switching... 😉

More seriously, AFAIK for most video cards the FIFOs are meant to be IRQ driven. For example, when the FIFO becomes empty enough the video card generates an IRQ and the IRQ handler shoves more commands into the FIFO. The idea is that the CPU spends very little time handling the IRQ and is therefore free to spent most of it's time doing other useful work while the video card does it's work in parallel. In this case a micro-kernel design does give you lots of context switches, but this is usually true for all IRQ driven devices (floppy, serial, hard disk, etc). When you decide to make a micro-kernel you decide to sacrifice performance for things like security/reliability (which IMHO is a fair compromise), and this is no different (but, the same idea of getting the video card to do work in parallel still holds, and avoiding the graphics acceleration and the context switches will probably make performance worse).

> **Combuster wrote:**
>
> > Of course I'd go a step further - I'm planning to use resolution independence and colour depth independence, so that normal processes don't need to care about these device specific (video mode specific?) details either.
>
> Exokernel here.
> *feeds output to /dev/blasphemy* 😈
> *ducks and runs*

Ok, let's think about this... How about an example?

There's 4 monitors and 3 video cards. The first monitor is connected to the first video card, which is an old S3 running at 800 * 600 * 15 bpp. The second monitor is connected to onboard Intel video and is running at 1024 * 768 * 16-bpp. The third and fourth monitors are connected to a nice ATI video card (with dual-output), and the third monitor is running at 1600 * 1200 * 32-bpp while the fourth monitor is running at 1280 * 1024 * 32-bpp.

The 4 monitors are arranged in a square, like this:

CODE: SELECT ALL

```
+-----+-----+
|  1  |  2  |
+-----+-----+
|  3  |  4  |
+-----+-----+
```

The 4 monitors are setup as a single virtual screen, and act like one big monitor.

You want to draw a green rectangle in the center of the virtual screen. To do this you create a "video script" that describes the virtual co-ordinates for the top left corner and the virtual co-ordinates for the bottom right corner of the rectangle, and the rectangle's colour. You send this script to a "virtual screen" layer, which create 4 copies of this script (one for each monitor) and clips them, so that each of the 4 new scripts describes a quarter of the original script/rectangle. Then these 4 new scripts are sent to the corresponding video device drivers, and each video driver renders it's script (including converting from virtual co-ordinates into video mode specific co-ordinates, and converting colours from the standardized representation into the video mode's pixel format).

Of course if there's only one video card and one monitor, the "virtual screen" layer can be skipped entirely (the script from the GUI/application can go directly to the video driver); and one green rectangle is fairly boring (but the exact same idea would work for complex 3D scenes, or windows, dialog boxes, icons, etc, complete with full hardware acceleration where possible); and there's no reason the same script couldn't be sent to a printer or something else instead (or stored "as is" in a file for later, or maybe sent to a ray-caster to create an insanely high-quality image rather than an acceptable "real time" image).

To me, this sort of arrangement is both flexible and elegant.

Now, try describing how something like this would work when the application is plotting pixels directly into the video card/s frame buffer (and please realize that I deliberately didn't say that all of the video cards are in the *same* computer)... 😉

Cheers,

Brendan

For all things; perfection is, and will always remain, impossible to achieve in practice. However; by striving for perfection we create things that are as perfect as practically possible. Let the pursuit of perfection be our guide.

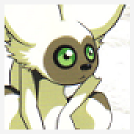
Re: Best way to do I/O port protection

by **Combuster** » Fri Jun 05, 2009 1:25 pm

I'd like to kick the S3 folks probably as much as you do for that matter. PCI was designed to solve this mess, rather than being raped.

Still, what has not been answered yet is why X.org (with their infinitely more NDA knowledge) decides to never MMIO the Trio64(V) while allowing it for its direct successor - Trio64V2 DX/GX. Both of which are available in PCI versions.

You mind if I copy that MMIO description to the wiki? (And thanks for the book tip)



Combuster
Member
★★★★★

Posts: 9301
Joined: Wed Oct 18, 2006 3:45 am
Libera.chat IRC: [com]buster
Location: On the balcony, where I can actually keep 1½m distance

Contact:

> Elite processing:

owned.

> The idea is that the CPU spends very little time handling the IRQ

Which is exactly the assumption that might be a tad too optimistic. Especially when that happens every 2-3 commands.

> Now, try describing how something like this would work when the application is plotting pixels directly into the video card/s frame buffer (and please realize that I deliberately didn't say that all of the video cards are in the *same* computer)...

Where did I say that all drivers have the same under-the-hood behavior towards all applications

"Certainly avoid yourself. He is a newbie and might not realize it. You'll hate his code deeply a few years down the road." - Sortie
[My OS] [VDisk/SFS]

Re: Best way to do I/O port protection

by **Brendan** » Sat Jun 06, 2009 12:40 am

Brendan
Member
Posts: 8561
Joined: Sat Jan 15, 2005 12:00 am
Location: At his keyboard!
Contact:

Hi,

> **Combuster wrote:**
> I'd like to kick the S3 folks probably as much as you do for that matter. PCI was designed to solve this mess, rather than being raped.

I'm guessing that when PCI was first introduced there was a time where device manufacturers tried to make older chips work with PCI without fully redesigning those chips (square peg, round hole, big hammer).

> **Combuster wrote:**
> Still, what has not been answered yet is why X.org (with their infinitely more NDA knowledge) decides to never MMIO the Trio64(V) while allowing it for its direct successor - Trio64V2 DX/GX. Both of which are available in PCI versions.

I'm not sure if there's a technical reason (e.g. dodgy MMIO), or a practical reason (lack of documentation), or an open source reason (old code that works versus better code that isn't implemented).

> **Combuster wrote:**
> You mind if I copy that MMIO description to the wiki? (And thanks for the book tip)

Ok

But, in the part about MMIO the book I've got only mentions the 864/964 chipset, and I'm not too sure about older S3 chips. Also, the book was first printed in 1994 (which was before Trio, ViRGE, etc), so I'm not too sure about later S3 chips either.

> **Combuster wrote:**
>
> > The idea is that the CPU spends very little time handling the IRQ

> Which is exactly the assumption that might be a tad too optimistic. Especially when that happens every 2-3 commands.

Yes. For each different video driver, the programmer/s writing the driver would need to consider possible methods (and maybe even do performance measurements) before they decide exactly how they'll implement the driver.

> **Combuster wrote:**
>
> > Now, try describing how something like this would work when the application is plotting pixels directly into the video card/s frame buffer (and please realize that I deliberately didn't say that all of the video cards are in the *same* computer)...
>
> Where did I say that all drivers have the same under-the-hood behavior towards all applications

My point here is that allowing other (arbitrary) processes to access the video card's framebuffer (or command FIFO or anything else) becomes a design nightmare once you go beyond the simplest possible scenario.

For special cases (e.g. an "ATI 4850 configuration utility" that's intended to only be used on ATI 4850 video cards) the driver could have special purpose (non-standard) interfaces; but I still don't see the need for special case utilities to have direct (rather than indirect) access to the device's hardware.

Cheers,

Brendan

For all things; perfection is, and will always remain, impossible to achieve in practice. However; by striving for perfection we create things that are as perfect as practically possible. Let the pursuit of perfection be our guide.