

Re: Weird behavior of dd

Source: <http://linux.derkeiler.com/Newsgroups/comp.os.linux.misc/2006-11/msg00053.html>

- *From:* The Natural Philosopher <a@xxx>
 - *Date:* Wed, 01 Nov 2006 10:00:04 +0000
-

Allan Adler wrote:

The Natural Philosopher <a@xxx> writes:

No, but you do.
That's where you start writing code to control them – a BIOS if you like.

OK

The usually a library of IO stuff over all that..I rewrote the stdlib essentially..so things like gets and puts and printf and fopen existed..

This also goes into the "BIOS"? What do you mean by "over all that"?
Do you mean you used gcc as a cross compiler on a Linux system to your PIC (or whatever) and used your own version of stdlib as the library to call when you compiled it?

Well I was using a nastier C compiler than that..

But what the process is is basically you need to write SOMETHING that gets burnt into ROM..enough to run probably a screen and keyboard, if its a computer with a console, and enough to read a disk and load whats on it and run it. After that its all software rather than firmware.

The BIOS is the first shim – the bit that tries to make whatever actual chips you have look like an idealised standard chipset. It usually also handles teh hardware interrupts from the hardware devices. Typically we USED to code that in assembler, because a lot of it was IO calls, which C didn't handle, a lot was interrupt service routines, and it had to be very small to go into a few K of ROM.

I was asked to replicate a C library on this particular board – it was about as complex as a basic PC – so at other programmers would have essentially stdlib there in the bios ready to call into. The actual stdlib itself then became a dummy piece of code that translated calls to the library into calls to the Bios.

Re: Weird behavior of dd

The "BIOS" got burned to a ROM chip?

Yes.

A multitasking kernel is not that hard if you have timer interrupts available. Just make your library re-entrant, or with multitasking in mind. I cheated and just made any call to the hardware library make the calling thread full priority and uninterruptible. Like switching off scheduling for the duration.

OK.

I always heard that DOS wasn't re-entrant and sort of knew that meant DOS routines had to return without calling each other or calling themselves. I guess the way to avoid that is to have a stack of some kind on which to store registers every time one calls another routine and from which to pop them when the call to the 2nd routine is completed?

Well this wasn't DOS.

The basic principle of a multi tasking scheduler, is that a timer interrupt comes along, and, if other tasks (held on a memory array somewhere) are detected as being needed to run (most tasks are generally asleep, waiting to be woken up by an IO event, like pressing a key) then the interrupt routine saves all the current registers in a fixed part of the task array, loads them from the new task array, and issues a return from interrupt. This returns to where the NEW task had been running, with all the registers representing the state of that task. This is called a 'context switch'. The processor will find itself in the middle of some completely different routine with a completely different stack and carry on as if nothing had happened.

Non re-entrant access to the hardware is taken care of by a combination of two things.

(i) you don't switch contexts inside the BIOS itself. Any task whose code thread enters the BIOS becomes top priority and is not rescheduled.

(ii) You don't write a BIOS that can sit there and hang waiting for e.g. a disk driver to respond. Say you want to get some data off a slow disk. What you do is at the STDLIB level, you issue a command to the bios to get the data, put the task to sleep pending a flag from somewhere to tell it that the disk actually has the data, with a timeout function to return an error if the data isn't there in time. So the thread gets suspended until either the data is there, or it times out. The BIOS then takes the request for disk activity, and issues the read command, and then returns. leaving e.g. a flag set to show which process to give the data to.

When the disk controller issues an interrupt to say that the data is ready, the interrupt service routine reads the data to a disk buffer area, and flags the library to say it's ready. The sleeping task wakes up, reads the data from the buffer and returns with it.

The effect of the calling program is that the call to get the data simply takes a long time to come back with the data, but in the meantime another task can run.

Re: Weird behavior of dd

I'm probably not ready for writing a multitasking system, so I'd probably cheat the same way for starters.

No one is. There will always be bits of non re-entrant code. Imagine switching contexts whilst reading something long..from a drive..

At another level, Linux IIRC has things like a disk caching thread, that is 'in charge' of all disk IO, and does smart things like predicting which bits are likely to be read next, and delaying writes until a nice chunk of data needs to be written, and then optimising writes to minimise head movement over the disk subsystem.

The last task was to port a FORTH interpreter to it all. And a basic sort of 'command.com' type shell.

Where did this FORTH interpreter live in the resulting system?

16K of bloody ROM IIRC.

Who talked to the FORTH interpreter if everything was in assembler?
Was this for being able to program your PIC in FORTH?

Its along time ago now...I THINK that there was a basic command shell that understood things like 'run forth' or 'run a program off the floppy'.

The actual board was part of a large minicomputer. Its final function was to be the first thing that booted..and it communicated by serial bus with about 20 other actual boards and had to load them with THEIR BIOSSES, and get them to bootstrap, and if they failed, switch between them..MIL spec minicomputers, with a LOT of diagnostics and internal self monitoring. I suspect my boss at that point wanted FORTH to play with, and as a quick and dirty way to prototype algorithms for the final system. There is no doubt that FORTH is about twice as compact (though twice as slow) as ASSEMBLER when it comes to putting code in ROM.

The hardware emulator isn't strictly necessary, but it makes tracking bugs a lot faster. I had one obscure one where the machine would lock up every few hours. Turned out to be a timer interrupt in a critical piece of code..

Apparently, there is an intel x86 simulator called bochs. I may have downloaded it once but was unable to figure out how to install it.
I guess I can try again. According to the description at sourceforge, Linux, Windows 95 and Windows NT will run on top of the simulator.

Re: Weird behavior of dd

A simulator is good, but it isn't an emulator. An emulator is a bloody great box full of shit into which a computer plugs, and which replaces the actual chip on the motherboard..so you can trace what is happening to every pin on the chip, and halt it if something interesting happens, and see what caused it.

Logic analysers are nearly as good – they simply monitor all the lines on a real chip, and record what happened, the emulator can stop it and actually even allow you to manually 'change' a pin state and the like.

Simulators are merely bits of software that behave LIKE a chip, so you can run code for that chip on them. You can use them to debug code, but not debug hardware.

Actually the simplest thing to get going these days is a PIC. I'd start there.

Thanks, I'll look into it. At www.piecmulator.com there are links to various PIC emulators.

You can always use it as an IO controller for a bigger machine later..

You mean, e.g., I take my ancient PC and have it use the PIC as an IO controller somehow? Or do you mean that I can take some of the components of an old PC and have the PIC control them instead of the BIOS and OS of the old PC?

I meant you could use a PIC to e.g. become a terminal IO controller when you built your 256 x X86 computer with 150TBYTE RAM to crack the NSA codes ;-);-)

embedded systems is the whole name of all this stuff..making processors and ram into a computer that doesn't necessarily have a console or a disk. Its huge fun.

That sounds very interesting. How expensive is it to do this with one PIC?

Cheap as chips haha. You can get a PIC board and software to put on it for e \$100 IIRC or even less. The PICs have mostly flash ROM that you can program via USB. Plus loads of assemblers, simulators and cross compilers. All of which mostly run on Linux, those WINDOZE is better served..

And you end up making things like clever burglar alarm controllers and the like. A great way in to low level

Re: Weird behavior of dd

programming without having to actually design any hardware.

What about having a lot of PICs running together as a kind of multiprocessor?
At what point do you have to stop using your wall socket and have to start
designing special purpose power supplies to handle all the PICS?

Dunno. That's when you are into designing your own PCB's and the like – you would probably simply take a PC case and PSU, and design a motherboard for it loaded up with PICS..but PICS are not the best thing to use at this point..you probably would use Pentiums, or ARM risc chips. there are loads of interesting chips out there – DSP's for one thing. All are optimised in different ways.

The PICS do IO well – but they are limited in power and memory space. The DSP's do ultra fast maths, and the general purpose CPUs are optimised to run operating systems and code.

Building your own computer though, is a several year task..