OS/400

IBM i: An Unofficial Introduction

Getting Started With The Future Systems Legacy

Hugo Landau

Contents

Chapter 1

What is IBM i?

1.1 The Strangest Operating System In The World

Actually, probably not by far. There's plenty of strange history in the field of computing. But that makes a good introduction, so let's go with it.

IBM i is *different*. You might already know this and think you understand how different it is, but that's just peanuts to how different it actually is.

Here are some things that make it *different*:

- It doesn't have a filesystem.[1] Instead, it is an "object-oriented" operating system. Objects are stored in "libraries", which are roughly analogous to directories. Unlike a typical filesystem, these objects aren't just bytestreams; they have a type, the internal representation of their contents is not exposed, and you can only perform operations on them appropriate for that type.

- Instead, it has an SQL database built into the kernel. An example of a type of object is a "Physical File" (PF), which means a table in SQL terminology. You can create a PF with some given schema, insert rows into it, query it, etc. The internal byte representation of this table is undefined and inaccessible to the user — the operating system only exposes these high level semantics.

- It is impossible to run native, machine code programs. In this regard, it's a bit like a web browser: programs are written in some source language (in the case of a web browser, JavaScript) and (for performance's sake) translated to machine code by a trusted just-in-time compiler. Besides the kernel itself, all machine code executed by the system is generated by the kernel. It's a bit like if the Java VM was the basis for an entire operating system — but in fact, OS/400 predates Java entirely.

[1] Actually, in the end it got one tacked on for its POSIX compatibility environment ("Integrated File System" (IFS)). It's not part of the original architecture and is very much something bolted on the side, though, so we don't discuss it here.

In fact, it's interesting that this approach, originally invented by IBM, has been resurrected so many times; with Java, with .NET, and with JavaScript. The concept of using a trusted compiler at the kernel level also has been resurrected; Microsoft's Singularity project sought to define an entire operating system in C#/.NET. Because all machine code was generated by a trusted translator from .NET bytecode, security and isolation invariants are enforced by the trusted translator, not hardware virtual memory protections implemented by the CPU. The Singularity research paper mentions that this allowed virtual memory to be disabled, avoiding the need for TLB lookups and increasing performance. More recently, an open source project (Nebulet) has been experimenting a kernel that executes only WebAssembly, a bytecode designed to be easily translatable to machine code for common architectures, again using a machine code translator as part of the *trusted code base* (TCB) rather than hardware memory protection.

Another benefit of this approach is hardware architecture independence. The IBM i kernel could be ported to a different CPU ISA, and all programs built for IBM i would continue to be able to run without recompilation. In fact, such a transition has happened in the past, when IBM moved from running IBM i on CISC CPUs to running on Power ISA RISC CPUs.

- Its "single level storage" (SLS) architecture. Broadly speaking, this means that every object in the system (on disk, etc.) has a 64-bit virtual address assigned to it, whether or not the object is currently in memory. The kernel automatically pages in these objects when the corresponding area of virtual memory is accessed. This means that the bookkeeping structures on disk predominantly deal with the mapping of 64-bit addresses to disk block addresses (rather than mapping filenames, strings, to disk block addresses as in a conventional filesystem).

  Aspects of this approach can be found in modern incarnations of other OSes. POSIX-style OSes allow you to open files (using a filename) and then map them into memory using the `mmap` call. The data of the file won't all be loaded into memory at once, but will be paged in by the kernel as you access different regions of the virtual memory assigned to that file by the `mmap` call. But each process has its own address space and the virtual address assigned to a file will vary with each invocation of the program. If you want to think of SLS in POSIX terms, think of it as though every object stored on every disk in the system is mmap'd into every process at all times, and at persistent, unchanging addresses.

- You may wonder how this can possibly be secure. The answer is given above; because the only machine code executed directly is that generated by the trusted translator in the kernel, the kernel can assure itself that no program can access arbitrary memory locations by simply refusing to generate machine code that does so. On a POSIX system, you can write `*(int*)0xDEADBEEF` in C — the C specification would tell you off for doing so, and you'd better hope there's valid memory mapped at `0xDEADBEEF`, but it will in practice work so long as that is the case. In IBM i, pointers have to come from a legitimate source and cannot be "forged".

To enhance performance, this is tracked using a tagged memory system, in which a single bit of metadata is stored for every four bytes (one word) of ordinary RAM. A pointer in memory IBM i is 16 bytes long[2] — and is only considered valid if the metadata bits for each of its four words is are all set. The CPU automatically clears the metadata bit if one of these words is modified arbitrarily. The machine code generated by the kernel uses a special CPU instruction to set these tag bits, but it only generates that instruction to "bless" pointers which have come from a source considered legitimate by the kernel.

This metadata system is an architectural extension of the Power ISA on which modern IBM i runs. It's important to emphasise, though, that these architectural extensions provide a way to set and retrieve one bit of metadata for every word in memory, but don't really implement any security logic around it. The CPU won't fault if trying to dereference an untagged pointer; the trusted translator must emit instructions, as part of the machine code it generates, to check if a pointer is untagged and raise an exception if this is the case. If an attacker could generate arbitrary machine code, the security of the system would collapse. These architectural extensions should be thought of simply as a means to enhance the performance of the code generated by the trusted translator; it would be possible to implement such a trusted translator on a machine without these architectural extensions and without changing the semantics of the programs it executes, but performance would be poorer.

1.2 Back To The Legacy

Let's be honest: IBM i is a legacy system. Yet even given this fact, in some ways it comes across as more futuristic than contemporary, mainstream operating systems. A product of IBM's "Future Systems Division" — their mission being to bluesky new ideas in systems design — it was ahead of its time in the 1980s and arguably is *still* ahead of its time. At the same time, anyone who has accessed an IBM i system via a terminal full of lime green text will appreciate that this is a system that time forgot.

As one of IBM's "midrange" systems, IBM i exists to service boring business functions. Payroll, warehouse inventory; if you have some deeply boring bookkeeping to take care of and need an extremely reliable system to that end, IBM i has you covered.

So IBM i is at once extremely advanced technology, applied to the most mundane of ends. If you are a retail employee, you could easily use an IBM i system via a dated greenscreen and never appreciate the absurd and improbable sophistication of the system that lies behind it. It's a bit like encountering an unreal computer of extraterrestrial origin and almost incomprehensible sophistication that was transported back in time 100 years in a freak timewarp but which is now being used solely to keep the accounts of a random credit union in Calgary.

[2] Since IBM i uses a 64-bit virtual address space, half of this goes unused. This is extreme futureproofing on the part of IBM.

I don't say this to express snobbery at these business functions: they're important, and when they stop working, everyone pays. In the last few years in the United Kingdom, multiple banks have had their IT systems explode, temporarily causing people to have difficulty accessing their money. IBM i's reliability is to its credit. At the same time, the more one learns about the weird and little-known internals of IBM i, one can't but help but feel that this is an architecture that should have changed the world.

I've heard it said that IBM is simultaneously the best and the worst custodian for IBM i. The best, because any other company would have canned it years ago; the worst, because at the same time it has been woefully neglected, with the IBM i department devoid of funding for anything really other than maintenance.

We have IBM to thank for IBM i continuing to be a maintained OS. At the same time, without the internal funding to develop the OS further, it's withered and is being kept on life support.

At times, I've written about the "cultural defeat" of Microsoft in recent times. What I mean by this is that POSIX environments (Linux, OS X) have come to dominate as the preferred environments of developers. Windows does not come from a POSIX culture; modern Windows is a result of the combined influence of OpenVMS (which informed the design of Windows NT) and Windows 98, itself influenced by DOS and OS/2. As a platform, it does not have a "blood" relation to any POSIX-like system. But increasingly few developers view that OS environment as their desired development environment, which inevitably means that more and more developers will approach Windows with a POSIX mindset. This diminishes the viability of the cultural differences of Windows as a technology stack. In an effort to respond to this changing situation, with Windows 10, Microsoft has added binary compatibility for Linux executables in the form of the new "Windows Subsystem for Linux" (WSL). In essence, you could view this as a form of cultural capitulation — an acceptance that POSIX has won, and trying to lure a new generation of developers to a Windows-centric way of thinking is a lost battle. It logically and inevitably follows that more and more of Windows will be "converted" to a POSIX-esque environment; not because the existing components will be changed (they must necessarily remain as they are for compatibility purposes), but because more and more development of Windows itself will be to add functionality to the new Linux-compatible subsystem, and to other new "alien" annexes of Windows that are added to appeal to the new dominant sensibilities of how an OS should work.

The same dynamic has occurred for IBM i, with the tacked-on PASE environment — a POSIX compatibility environment which was added to IBM i, providing some level of binary compatibility with AIX, IBM's UNIX variant. More and more development on IBM i now happens within the PASE playpen of a UNIX shell environment. The native IBM i environment, by comparison, is renovated less and less as time goes on.

In both the case of Windows's WSL and IBM i's PASE, the "impedence discontinuities" between the native and POSIX-compatible environments are enormous, which of course discourages developers attracted by the POSIX environment from jumping the gap to the native environment, thus further reinforcing the cultural trend towards POSIX, POSIX, POSIX. These impedence discontinuities are conceptual and semantic conundrums; they cannot be coded away by engineering prowess. The concept of objects, libraries, databases that constitute an IBM i system simply do not map coherently to POSIX's concept of a filesystem formed of structureless, byte-oriented files.

If it's not obvious, although I think this cultural effect is undefeatable, I'm nonetheless not a fan of it. POSIX has won; but with it OS research is even more dead than it has been for the past few decades. In this book, I therefore intend to focus wholly on IBM i as it was originally envisioned; little attention will be given to the PASE environment.

1.3 IBM Block Terminals

There are, broadly, two types of terminal: character terminals and block terminals. If you have ever used a command line on a POSIX system, you've used a character terminal. The Digital Equipment Corporation (DEC) made a popular line of character terminals such as the VT100, and modern terminal emulators such as xterm hail from this long legacy, implementing a compatible set of control commands.

Most people are accustomed to character terminals and how they work. IBM's block terminals are quite different and tend to provide more sophisticated, more "GUI-like" user interfaces. Two block terminals in particular are widely known:

- The IBM 3270, used to interface with IBM mainframes such as System/360;
- The IBM 5250, used to interface with IBM midranges such as IBM i systems and their predecessors, System/34, System/36 and System/38.

While these were once physical devices (they are in fact model numbers), terminal emulator programs are now invariably used instead. But just as with the VT100, these numbers are used to denote the type of interface a terminal is expected to conform to. A TELNET binding for 5250 datastreams is defined, and TELNET is thus commonly used to connect to IBM i systems over the network. Use of SSL for security is now common.

The reason these are called "block terminals" is because communication to and from the terminal occurs in blocks of information, rather than individual characters. When you use a character terminal, you've probably noticed at some point or another the fact that characters are processed one by one, as fast as they can be produced (by whatever is writing to the terminal) and processed (by the terminal). `cat` a large text file to a terminal and watch it scroll by; the terminal tries to show all of the characters as fast as possible.

With a block terminal, entire screenfuls of information are sent at a time, in a single unit. Conversely when typing, information is not sent to the computer to which the terminal is connected immediately, but only when some particular action happens, like pressing Enter to submit a form being filled in, or pressing a function key. This reduces load on the computer to which the terminal is connected, and means that the responsiveness of the UI is not dependent on the round trip time to the computer with every single keypress. If you have ever had to connect via SSH to a computer on the other side of the world, you have probably noticed the delay between typing a single character of a command and seeing it appear. With a block terminal, this unavoidable round trip delay is suffered less frequently, such as after having written out an entire command, at the time it is submitted.

Block terminals can be compared to modern business web applications. Both wait until a form on screen has been fully filled out to contact the server, so their UI can be adequately responsive for such interactive applications. Whereas modern character terminals might mainly be used by systems administrators and developers, block terminals were intended for a wider audience — ordinary persons without high levels of skill in IT who needed to interface with large, expensive business computers.[3] As such, the 5250 block terminal was the primary interface to IBM i at the outset. A single IBM i system might have a large number of 5250 terminals attached to it, serving many employees simultaneously; the 5250 is well suited to displaying pages of (monospaced) information, and forms for performing database searches or data entry. The web is, and was always intended as, a hypertext platform for the publishing of information, but rapidly became co-opted as a sort of more modern and capable block terminal system; the modern enterprise web application can be thought of as a direct successor of this kind of architecture in which terminals were used by lay persons to access a single, central computer. The increased sophistication of block terminals relative to character terminals is a product of these requirements, and modern web applications appear to have come to similar decisions, driven by similar requirements.

To connect to an IBM i system, you will need a 5250-compatible terminal emulator. Such emulators are of course available from IBM, but if you are not in possession of such software, one option in a pinch is the free `tn5250` program for POSIX systems. (This program actually runs in a character terminal, so it can effectively be thought of as an adapter from the 5250 block terminal to a DEC VT-style character terminal.)

Just as the web environment has been evolved and augmented with many new abilities over the years, the 5250 (and for that matter DEC's character terminals) also have stories to tell of accumulated features over the years. For example, the 5250 protocol at one point had the ability to display images added to it. However, support for many of these now obscure and little known capabilities tends to be varying, at least in third party terminal emulators.

1.4 Anatomy of an IBM i System

1.4.1 Terminology

Table 1.1 gives a guide to some common IBM terminology; you'll need it.

[3] Anyone familiar with the history of computing will be aware of the "personal computer" revolution — itself a cultural insurgency against a then predominant IBM culture of having one big, expensive computer ("the" computer), with terminals simply serving as dumb access points to this one computer. If you consider that such terminals must contain microprocessors to implement the various fancy functionality that block terminals require, this arguably constituted a case of artificial dumbness; were these terminals not simply deliberately and obstinately single-function computers? The PC revolution was surely inevitable, being as much dictated by technology as a humanist vision. Our modern "terminals" are rich with capability beyond the wildest dreams of our forebears.

IBM Term	Common meaning
IPL (Initial Program Load)	Booting.
DASD (Direct Attach Storage Device)	A hard disk.
Load Source	The DASD from which a system IPLs.
HLIC (Horizontal Licensed Internal Code)	Microcode.
VLIC (Vertical Licensed Internal Code)	Firmware.
PLIC (Platform Licensed Internal Code)	Firmware — the kernel of PHYP.
PHYP (Power Hypervisor)	PowerVM.
LPAR (Logical Partition)	A virtual machine.
SLIC (System Licensed Internal Code)	The IBM i kernel.
OS/400 (aka XPF)	The IBM i userspace.
PTF (Program Temporary Fix)	OS patches, hotfixes, etc.

Table 1.1: Common IBM terminology

1.4.2 Kernels and Userspaces

The "Licenced Internal Code" terminology deserves discussion in itself. IBM used to call this "microcode" and now calls it "Licenced Internal Code" (LIC). In particular, IBM i is formed of two installable components, one of which must be installed after the other: System LIC (SLIC) and OS/400. Although IBM used to refer to LIC as "microcode", IBM's definition of "microcode" is extremely broad, including both what people would ordinarily understand as microcode (microarchitectural code controlling the decoding of CPU instructions, now referred to by IBM as "Horizontal LIC"), but also what most people would call firmware ("Vertical LIC"), a hypervisor kernel ("Platform LIC", forming the core of PowerVM) and, in the case of IBM i, the OS kernel ("System LIC").

Once upon a time, IBM i ran only on machines designed specifically for it, and had the whole machine to itself. Eventually, IBM elected to unify IBM i and their Power Systems servers, and now IBM Power Systems servers are the only supported way to run IBM i. These servers have a hypervisor, PowerVM (aka PHYP), built into their firmware. Any OS installed on these servers, whether it's IBM i, Linux or AIX, always runs under PowerVM. Multiple "LPARs" (IBM-speak for virtual machines) can be created, allowing multiple instances of IBM i, Linux and AIX or any combination thereof to all run simultaneously on the same hardware. Modern IBM i is designed to run exclusively under PowerVM and quite closely coupled with it.

Don't expect IBM to spell out in much clarity what SLIC exactly is; they're rather coy about it. Instead, I will: it is the kernel of IBM i — it handles system calls, generates the machine code, and also contains the integrated DB2/400 SQL database.[4]

[4]Yes, IBM i is an SQL database engine running in kernel mode.

1.4.3 Installation and IPL

The process of installing IBM i on a modern IBM Power server essentially requires three steps:

1. Create an LPAR in PowerVM configured to run IBM i.
2. Install SLIC into that LPAR.
3. Install OS/400 into that LPAR.

When compared to installing a POSIX operating system, installing SLIC is analagous to initializing a filesystem and copying the kernel onto it, but not any of the userspace utilities such as `cat`, `ls`, etc. The "load source" (the disk which IBM i boots from, that is, the system disk) is initialised and the SLIC code is copied onto it. It's impossible to make any use of IBM i without also installing OS/400, however, and having installed SLIC, the system will prompt you to install OS/400. This is essentially the userspace of IBM i, and is just as essential as the contents of `/bin` and `/usr/bin` are to a UNIX system.

There are a few things to note about IPLing (that is, booting) an IBM i system that differ from more mainstream systems you may be used to. When performing an IPL, you must select an "IPL type" and an "IPL mode". The IPL type is one of the following:

Type A Boots the "A-side", which is IBM i without any of the PTFs (patches) applied. This is essentially a "golden side" kernel which remains pristine.

Type B Boots the "B-side", which is IBM i with PTFs applied. This is the most common IPL type; Type A would only used when IPLing for the first time after installation, or when installation of a PTF has caused a serious issue and needs to be rolled back.

Type C Used to boot into special diagnostic and recovery tools. Can be thought of as a recovery mode, or emergency debugging mode, and not generally used unless one is specifically advised to.

Type D Used to install IBM i for the first time; boots from optical or tape media instead of a load source DASD. In other words, this is equivalent to "booting from DVD" on a garden-variety x86 server.

While the "IPL type" determines what is booted, the "IPL mode" determines in what style it is booted. À "normal" IPL boots the system automatically without asking questions, whereas a "manual" IPL is a verbose and interactive process to boot the system. A normal IPL is usually desirable, but a manual IPL must be performed in some circumstances. Type C and D IPLs must always be manual.

Once upon a time, the IPL type and mode were set via a physical front panel on the front of an IBM i machine. The "IPL mode" would generally be set via a keyswitch, and the "IPL type" could be set via buttons under a small LCD. This LCD would also show an eight-character hex code known as a “System Reference Code” (SRC); this code indicates what the system is currently doing while it is IPLing, and the condition of the system once it has finished IPL. These codes can be looked up in IBM documentation.

Nowadays, with IBM i only existing in an LPAR of the PowerVM hypervisor, the IPL type and mode are set as part of LPAR configuration; SRCs are available in the same manner.

Chapter 2

Using a 5250

To be brief:

- A 5250 traditionally has 24 function keys. Since modern keyboards don't have 24 function keys, all 5250 terminal emulators map Shift+F1 as F13, Shift+F2 as F14, etc.
- PgUp/PgDn and Tab are used a lot. If you don't know what you can interact with on a given screen, press Tab.
- The meaning of function keys is generally highly standardised, so that they always mean the same thing in no matter what context:

  F1 displays help.

  F3 exits whatever program you are currently in. It's similar to F12, but more major.

  F4 is used to display what values are valid for the field the cursor is currently in (if supported by the field).

  F12 cancels; goes back a screen.

- Fields (that you can type text into) are almost always underlined. You can navigate between them with Tab.
- Pressing Enter will generally "submit" the current screen. (If you don't want to proceed with some action, press F12 or F3 instead.)
- Don't underestimate the help system; it's surprisingly effective. If you position the cursor over a specific area of the screen before pressing F1, the help it prints may be context-aware. Underlined terms in displayed help are hyperlinks; move the cursor over them and press Enter to follow them.
- If you try and type in an area of the screen that isn't a field, the terminal will get angry at you. If this happens, just press Tab to move back to a field.

- First impressions may lead you to believe that IBM i is a purely menu-driven OS. In fact, most menus have a command line at the bottom, and allow commands to be entered as well as menu option numbers. For guidance on available commands, press F4 at the command line. (In fact, seasoned IBM i users can set an option to disable the menus entirely, leaving more room for the command line.)