Subject: Re: XPL and GORDO OS query
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 3/11/25, 1:38 AM
To: ken rector <kdrhoo@yahoo.com>

> On Mar 10, 2025, at 5:56 PM, ken rector <kdrhoo@yahoo.com> wrote:
>
> Hello DR Booth,
>
> I'm working on a project to implement the ARPANET IMP interface on a simulated Sigma 7 as it was in 1969.
>
> It seems the UCLA guys may have used an OS called GORDO for the host support and GORDO seemed to support the XPL compiler. The SDS Sigma User Group had four tapes containing XPL , two for Sigma and two for GORDO. I've acquired a copy of one of the XPL(GORDO) tapes, The others don't read that easily after 50 years.
>
> Could you help me in any way to find out more about this stuff?
>
> Regards
> Ken Rector
> Woodland Hills CA

Yes.

Are you asking about XPL, or GORDO? And what is on the tape that you have?

I am not certain why you would want to implement an IMP on a Sigma-7, simulated or otherwise. The IMP was a separate piece of hardware from the Sigma-7 when it was used in the ARPANET. The Sigma-7 was a host. The IMP was the interface to the network that was connected to the Sigma-7. At least that is my understanding.

I was in the Sigma-7 for the first few years that I was at Livermore, starting in June 1968. Initially, I was a summer student but converted to full-time in September of that year. My assignment was working on a Sigma-2 computer (much smaller than a Sigma-7) that was intended to host a “remote” grahpics display in another part of the site (there was a high-speed connection designed by SDS between the Sigma-7 and the Sigma-2). Eventually, I became the primary programmer for the GORDO O/S after the original team had departed. The system was broken into the A System (what would now be called a kernel) and the B System (most of the what would not be considered the API to the O/S.

Both the A and B systems were written entirely in Sigma-7 assembly language. This remained the case for the A system and I think also for the B system. XPL, which was somewhat new at the time, was ported to the Sigma-7 by Charles Wetherell (I had a small hand in that because of my expertise in assembler, which allowed me to “optimize” the IBM 360 emulator that Charles had written to run on the Sigma-7 to support bootstrapping the XPL compiler to the Sigma-7. The first step was to run the XPL compiler that produced 360 executable using the emulator (so compilations were very slow). The second step was to modify the compiler to emit Sigma-7 executable. The compiler was written in XPL, so once the second step was complete, the compiler could compile itself into Sigma-7 executable and then speed was not an issue. Further modifications could be made to iomprove the emitted code without taking overnight to compile.

Once the XPL port was complete, we wrote utility programs in XPL that ran on GORDO. There was already a FORTRAN library for the GORDO API. I forget whether we linked to that or just made an extension to XPL to

invoke system calls directly to the a system (these were not hardware system calls, they were standard subroutine calls but they crossed an access control barrier that essentially triggered the equivalent of a system call in the IBM 360 SVC sense. As far as I know, we did not program either part of the O/S in XPL, but I might not be remembering this correctly. If we did, it would only have been for the B system. The A system was sufficiently complete by the time XPL came along and there would be no advantage to reprogramming it. The A system was qutie small. It handled paging for the virtual memory, inter-process communication, time-slicing to divide cpu time among the various user processes, and the very lowest level of interrupt handling for devives other than disks (magnetic tapes, line printer, card reader, and card punch) for I/O not related to page swapping, and support for the graphics displays (both local to the Sigma-7 and remoted to the Sigma-2 that was later replaved by a Sigma-3).

Can you tell me what is on the one tape you have?

When Kleinrock's group was developing the IMP, they used a Sigma-7 for reasons that I do not remember. This is clear from the whiteboard image of this video, which shows a diagram with the UCLA ARPANET host being a Sigma-7. Most of the hosts were PDP-10s, as far as I know.

https://conferences.ucla.edu/ucla-birthplace-of-the-internet/

SDS did not have a real-time operating system for the Sigma-7 that was suitable for the ARPANET work (or for what Livermore was doing). My recollection is that their O/S did not use the paging hardware until UTS (Universal Time-sharing System). Prior to that, they had BPM (Batch Processor Monitor) and BTM (Batch Time-sharing Monitor). I believe the the Wikipedia page has a lot about this.

https://en.wikipedia.org/wiki/Universal_Time-Sharing_System

My recollection is that Kleinrock's group began working on their own O/S for the Sigma-7, but like many such projects it took longer than expected and got bogged down perhaps in feature-itis. They contacted Livermorre to see if GORDO might be a solution to their problem. This was while I was still a "junior" on the GORDO team. The two key Gordo developors, Ken Bertrand (A system) and Gary Andersom (B system) were still leading the project under the direction of Dick Conn. Bob Milstein had been part of the design team and I think he may have written some of the A system code, but he was just walking out the door when I arrived, so Ken and Gary did most of the programming.

After discussion that I was not involved in, Kleinrock's group decided GORDO would work for them. A deal was struck between UCLS and Livermore (I know nothing of the details — it may have been "for free"). Vint Cert was I think the project leader at UCLA. He and Steve Crocker, Charlie Kline, Jon Postel all visited Livermore a couple of times for meetings to discuss the deal and later to ask questions about the GORDO implementation. Wikipedia has some background on this (all but Charlie went to Van Nuys High School, it seems).

https://en.wikipedia.org/wiki/Vint_Cerf

GORDO was named after the comic strip Gordo by Gus Arriola.

https://en.wikipedia.org/wiki/Gordo_(comic_strip)

Somewhere I might have a copy of a letter from Arriola to Dick Conn giving permission to use "Gordo" as the name of the O/S. My reading of the letter is that Arriola was a bit sarcastic. The letter requesting permission had noted that there had been or would soon be a paper presented at the 1968 Annual ACM Conference held in Las Vegas NV. Arriola made what I thought was a caustic remark to the effect compared the "sin city" reputation of

Las Vegas with the ethics of nuclear weapon design. But perhaps I misread his intention.

Here is a link to the paper that was presented in Las Vegas.

https://dl.acm.org/doi/10.1145/800186.810559

Usually our system was called “Gordo” with only an initial capital letter, but it probably got all-upper-cased in various documents because 128 characters was not yet the norm and letters were often only upper case.

I remember Steve Crocker and the others explaining that they had spent a lot of time trying to figure out a name for the Sigma-7 O/S they were developing and he told a joke that after a number of clever acronyms had been suggested a rejected, a woman on the team had said the dicussions were really a waste of time — why didn’t they just call a spade a spade. I think they might have decided to call the system SPADE because of that. One of the acronym guys couldn’t resist coming up with an acronym for SPADE — Some Poor Ass’s Design Effort. Whatever the name was, it died with whatever partiallty complete O/S they abandoned when they took on Gordo. But their urge to name systems lived on. I am not sure who came up with the name, but they called the system SEX and I remember Charlie Kline saying he had always wanted to write a Sex Manual, so this was his chance.

Kline was I think an undergrad at the time. He apparently developed fairly comprehensive documentatiion for Sex (nee Gordo) that was much better at documenting that was the Livermore Team, in part because we had a very small user base. I remember that Charlie sent back a few corrections to the code that he discovered while writing the documentation. My recollection is that we did not adopt his documents because UCLA added new features that we did not need and so Sex and Gordo diverged enough that the Sex Manual was not accurate enough for our use.

Discussions pretty much died off within a year because the two projects had different goals, and of course UCLA was part of the ARPANET Project and thus fairly “open” whereas Livermore was (and is) a nuclear weapons lab with high security. Nothing in Gordo was classified, but very little got outside the lab except for a few conference papers that were ot particular detailed (standards in those days for conference papers in computer were not that high). Bertran and Anderson left the lab and formed a company that developed GEM, a modified version of Gordo. The company survived for a while, but in the end was not successful, I think in part because SDS (by now renamed XDS after being acquired by Xerox) came out with UTR and the CP-V that finally made good use of the hardware. If you search this article for GEM about halfway down, there is some information about it but I cannot vouch for its authenticity.

https://www.uwyo.edu/infotech/aboutit/history/sigma-era.asp

The above article mentions Steve Booth. I have no idea who he is / was. As far as I know, we are not related in any way.

I cannot remember if I heard this remark directly, or only heard it second-hand. I think I heard it directly but it was a repeat of the original. Sid Fernbach, the head of computation at Livermore, said that “most computer hardware companies ought to quit developing new hardware for five years until their software catches up with the hardware. In the case of SDS, they should quit for ten years.” I cannot claim to know this was 100% true, but it did seem to me that SDS had built a very nice hardware design for the Sigma-7 but the did not take advantage of it. There were probably many reasons for this, including the owner / founder Max Pavlesky who apparantly was a chaotic force in the company. Xerox acquiring it was more or less its death knell. I am told that Xerox PARC was told they could not use DEC PDP-10s when PARC was starting up in 1970 because Xerox had acquired SDS in 1969 and XDS was competing with DEC in the same market with the Sigma-7 as the PDP-10. PARC researchers balked and designed the MAXX machine that had a PDP-10 emulation mode rather than run Sigma-7 hardware.

Their objection was not the hardware, it was the software. The PDP-10 had good operating systems, including Tenex, which was dominant in the ARPANET community.

The 1968 ACM National Conference paper has a good overview of the architecture of Gordo. I can probably fill in details if you are interested. I also have some comments about how SDS failed to follow the advice of Butler Lampson (and I think Peter Deutsch) gained from the earlier SDS 940 system that made the Sigma-7 less than perfect for its task. But one example is that the hardware did not support handling page faults nearly aa well as the Tenex paging box did, which resulted in significant overhead in the kernal when there was a page fault. Another is that it seemed to me that SDS tried very hard to avoid a problem IBM had with the 360 series and its character instructions (such as MVC — move character) when paging was first introduced in the 360/67 (I think that was the model number). If an MVC crossing a page boundary caused a page fault, the entire instruction was rolled back until after the page fault had been serviced. This required significant hardware to achieve. SDS designed its byte string instructions (such as MBS — move byte string, more or less the equivalent of MVC on a 360) to be “interruptible” by both external interrupts and also internal traps (such as page faults). But they did not fully succeed. The execute instruction (EXE) could trigger a page fault that was impossible to diagnose if it was executimg an MBS instruction or any of the other interruptible instructions such as load/store multiple or push/pull multiple instructons that could access up to 16 consecutive words of memory.

Cheers,

Kelly Booth

Subject: Re: XPL and GORDO OS query
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 3/11/25, 3:56 PM
To: ken rector <kdrhoo@yahoo.com>

Ken,

I will follow up later this evening with more of a reply. I may go into a cone of silence after that for a while, but I am interested in continuing the conversation.

The DIGITEK compiler was a classic. It was advertised in Datamation (and other publications) with just a large photo of a soldering iron and the explanation that this was a “do it yourself” compiler kit. It emitted extremely basic assembly code because the intent was to get a working compiler onto new hardware quickly. It was never intended to be a production quality compiler that had high-level optimization or other features. This lead to an interesting episode at Livermore related to a FORTRAN program that essentially was unloadable because the SDS loader proceeded at a snail’s pace to the point where the person trying to load it gave up. I eventually discovered the problem and made a very simply change to the loader that sped things up immensely. I actually wrote about that last night in a different message to someone else who had posted about the Sigmas and had mentuoned the slow loader but without the connection to the DIGITEK compiler — the compiler triggered the bug but was not the cause of the bug,

Yes, as I wrote earlier, SEX as the name that UCLA adopted for Gordo. It was never an SDS product or, as far as I know, available through SDS. Neither was Gordo. I do not think that SEX derived from anything SDS had written. I think it was a joke by Charlie Klein because he liked the idea of having a SEX Manual.

After sending you the email last night and then reading some other articles I found on the web, I wrote a few things to someone else hoping to hear back (I have not yet, but am still looking). I will salvage that tonight and send a revised version to you that answers some of your questions.

Briefly:

a) I do not have any of the code for anything related to Gordo. I left the group after a while but remained one of the local “experts” until roughly when the Sigma was decommissioned and I left the Lab. My close colleague John Beatty took over my role as maintenance programmer for Gordo and doing odds and ends to assist users. John died in 2020. My memory is not great about this, but I know we definitely discussed re-writing the B System in XPL. Whether we actually did that or not, I cannot remember. If we did, I suspect it was John Beatty to did it. The timing, however, is a bit suspicious because I think by the time we might have done that our connection to Kelnrock’s group at UCLA was dormant. They had what they needed, Charlie Kline had sent a few bug reports back to us, and both Gary Anderson and Ken Bertran had left the Lab for a start-up to commercialize Gordo as GEM out of Utah. Mike Stone (who hired in a year or so after I did) went with them. That left no one of the original development team and I was then the “old fart” and the only one (until John came up to speed) maintaining the A and B Systems. The intimate hardware dependence of the A System meant it would not make a lot of sense to write it in XPL and there was no need to rewrite. But the B System only depended on a very minimal interface with the A System and it made sense to re-write it in XPL because that made it easier to add features — but as I said, I am not certain we actually did that.

b) This paper provides a good overview of the system design, but few details. I can fill in a lot of details because while learning the code I “rewrote” all of the A System in the sense of adding detailed comments and in a few places fixing inefficiencies or (in a very few places) fixing bugs. Ken Bertran was a very good programmer. I inherited the A System from him. Gary Anderson was also pretty good, but he did the B System precisely, I think,

because Ken was the better programmer down at the hardware level. The B System was designed to almost independent of the hardware other than be aware that Gordo used virtual memory and the B System was the interface to the A System where the paging and protection hardware was actually manipulated. So the B System maintained was in effect a somewhat abstract model of a virtual machine and the A System hid all of the gory details.

c) SEX diverged from Gordo in a number of ways. As I recall, Gordo had 16 “keywords” that were essentially file descriptors or file control blocks. This proved limiting for a few applications where multiple data files needed to be in use at the same time (if too many files were in use, the application code would have to close and open them again to avoid running out of keywords). SEX moved to 256 keywords (or maybe just 255), which was the maximum number of virtual pages so even if every open file had only one page in virtual memory there would be enough keywords for all active files. This required a change to some of the data tables in the B System (not the A System, as I recall, because of the good division of labor between the two). So after the changes to SEX, application code woud no longer be compatible unless it were written in XPL or FORTRAN (the only two languages for which we had compilers compatible with Gordo). Machine language programs would have to be modified a bit (not much).

d) I can probably resurrect the memory layout for both the A and B Systems (the latter is very easy — the top high-address pages of virtual memory were reserved for the B System which shared the address space with application code. Calls to the B System were ordinary branch-and-link (BAL instruction on the Sigma-7) subroutine calls using I think register 14 for the return address and register 13 or 15 (I forget which) to pass arguments that could either be values or the address in the application code where a table of parameter values was stored. Security was provided by having the memory protection hardware set to no-access for all of the B System pages when the applicaton code was running. The BAL instruction caused a page fault trap that the A System caught and after determining that the target address was in the B System the protection registers were changed to read-execute for the code pages and read-write for the PACT (program activation table, which was one 512-word page that had all of the context for the process). I could have a few details wrong, but this is close. (NOTE: moving to 256 keywords would mean that more than one page would be required for the PACT because it had 256 entries for virtual pages and one entry for each keyword, which would already fill up a 512-word page.

e) The important details about the A System are how it used the various hardware registers to support paging in order to detect page faults but also to keep track of when pages became “dirty” and needed to be written back to disk before being over-written in physical memory. Some rather poor choices on the part of SDS in the basic design (primarily the decision to have hardware registers that could be written but not read) meant that the A System had to keep a software copy of the mapping and protection registers. The details of context-switching were more or less what SDS had intended when it designed the hardware. The Program Status Doubleword (PSD) had sufficient state information for the A System to suspend a process and re-start it later, and the mechanism for fielding traps and interrupts that suspended subsequent events during the execution of the first instruction after an event was a great design that made it easy to implement a full context switch without worrying too much about mutual exclusion and it also meant that clock interrupts (for example) did not have to do a context switch because a single instruction was sufficient to advance the clock one or trigger a second interrupt if the clock was a timer that had run out.

If you were at Honeywell until 1992, did you every know Morven Gentleman from the University of Waterloo? Waterloo never ran Sigma equipment, but it did have a large Honeywell installation and did software development for Honeywell under contract. The reason I ask is that Morven (who died in 2018) told me a wonderful story he hard from someone high up in Honeywell about the SDS 940 systems that Honeywell eventually supported. According to what Morven told me, Honeywell had records of where every single 940 ever shipped was, including those that had been dismantled for spare parts. At some point (not sure of the date), they got a request from the USSR for some part for an SDS 940. This was puzzling because none of the 940s had

ever gone to Russian and to the best of Honeywell's knowledge, the only 940 whose whereabouts was unknown was the one that had been on the U.S.S. Pueblo that North Korea had seized. Honeywell did not follow up on the request, but the thought was that this was actually a covert message from the Soviet Union to US intelligence letting the US know that the Soviets had possession of the 940 and all of the other reconnaisance gear from the Pueblo. The 940 itself would not have been of much use (data might, but probably not even that). But the other equipment and manuals and logs might have been important. That was all back in the days of the Cold War.

More later. Let me know if the above is helpful.

In the meantime, look at the Las Vegas paper if you have not already looked at it. It has a high-level description of Gordo but not enough details to fully describe it.

https://dl.acm.org/doi/10.1145/800186.810559

Cheers,

Kelly

> On Mar 11, 2025, at 10:17 AM, ken rector <kdrhoo@yahoo.com> wrote:
>
> I have a couple of questions about how the B and A systems connect and what the memory layout should be. So far you are the only person who seems to have any knowledge of these kinds of things.
>
> Could I ask more detailed questions?

Subject: Re: XPL and GORDO OS query
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 3/11/25, 5:24 PM
To: ken rector <kdrhoo@yahoo.com>

> The source code file begins with "GORDO MASTER / COMMAND LANGUAGE PROCESSOR" and has your name as author 1969 and modified, 1972. It has a bootstrap section at 0x200
> , working pages at 0x400 and executable file header page at 0x1d000 followed by code. It ends at 0x1dfff. the code is divided into sections some commented as SUBMONITOR SERVICE CALL: xxx

No, this is a user-level application, more or less a very simplistic shell as in Unix. It was a command line processor that accepted keyboard input and did various things. It could be used to launch (fork) another process that ran an executable file. It also had various utility command including something to list the files in a directory (so like 'ls' in Unix), rename files in a directory (files did not have names themselves only directory entries and a file could be in multiple directories sort of like links in the Unix file system).

The submonitor service call section is I suspect the code that handles command lines specifying the various B-system calls. Applications could not directly access the B-system.

> It has references to G:xxx entry points in a transfer vector starting at 0x1e000 defined by EQU statements. There is no comment about B-SYSTEM. Could I assume it is?

The EQU statements were the virtual addresses of the various B-System calls (services). They were fixed by the Gordo architecture. If this were modern C code, it would be something like b-system.h

> The AP assembler claims * NO EXTERNAL SYMBOLS .

That means there are no other object modules that need to be linked to form a complete executable. In Unix and other systems, one usually links to system libraries that in turn deal with the Unix kernel and other components. Gordo was pretty simple and all of this linkage was via the EQUs that were internal symbols.

> The memory image file I assume starts at location 0 extends to 0x1ffff, has 0xc000 at location 0002, has some kind of a vector from 0x160 through 0x17b and similar vector stuff up to 0x19f. It has sensible looking code starting at 0xc000 through 0x13dff and 0x13e00 through 0x1ffff is all zeros.
>
> Could I assume this is an A-SYSTEM?

I will need to think about this. It probably is the A-system. If you look at the Sigma-7 instruction set and architecture, it shows the vector for interrupt and trap handling, which are hardware-defined locations. I think those usually had XPSD instructions (exchange program status doubleword), but I would need to look again at the architecture. The code starting at xC000 is probably the actual A-System instructions. It was not very large and remained resident at all times. The B-System ran almost like a user process and was paged in and out as needed. If the applications were not making calls to the B-System, its pages would eventually be replaced in physical memory with pages that were in use by the applications (code and data).

> There is nothing in the image at 0x1e000 or above.

The rest of memory woud be reserved for pages coming in and out for virtual memory in applications (and the B-system). All I/O except for the graphics displays and the swapping disk (and the com line to the Sigma-2/3) was done through virtual pages that were locked into memory when I/O was underway because devices only

talked to physical memory. The A-System made sure this worked and the B-system was how an application (that was trusted) could ask for pages to be locked for I/O and for I/O instructions to be initiated on its behalf by the A-System.

> How might the vector referenced in the source code get initialized?

I need to look at the hardware manuals to see exactly how interrupts worked. I think that the assemly language source specified all of the data in the locations designated by the hardware. It was intended that specific types of instructions be used (such as XPSD for most things but MTW — that would increment counters — for clock and timer interrupts).

Some place else in low core would be a vector of 4-word block, each a pair of PSDs (program status doublewords). When a page fault took place (for example) the hardware masked all interrupts for one instruction cycle and turned off memory mapping and access control and then executed the instruction at the location in low core corresponding to a memory protection violation. If the instruction was an XPSD instruction, the current PSD (for the application process that had caused a fault) was stored in the first double word pointed to by the address field of the XPSD and a new PSD was loaded from the next doubleword. So the second doublewords would be set in the assembly code and essentially baked into those locatons. The first doubleword would change each time there was an interrupt or trap. The new PSD and the XPSD instruction had fields that specified whether to use memory mapping, whether to continue to mask interrupts, and which of the eight register blocks to use (or maybe there were only seven and a zero code meant keep the current register block — need to check on that). This made for every fast context-switching.

> Fifty years! I'm impressed if you remember any of this.

Either Ken Bertran or Gary Anderson once said that they thought Charlie Kline must have slept with the source code at UCLA as they gook on Gordo because he knew it so intimately. At Livermore, I was the “new guy” who took over from both Ken and Gary, so I had to learn the code inside and out. I also had ot know the hardware intimately because my initial job was dealing with the graphics displays and they were down at the hardware level and there were some quite interesting anomalies, especially on the Sigma-2. I also handled a couple of bugs that required thinking a lot about the hardware. And the final impetus was porting the XPL compiler to the Sigma-7 from IBM 360. The first step in the bootstrap was to run the 360 code image via a simulator running on the Sigma-7. The simulator war written in FORTRAN and worked well, but slow. I was asked by Charles Wetherell (who was the person doing the entire XPL effort as a side project) so see if I could speed up the simulator. I had a lot of 360 assembler experience from a year-plus at Caltech as a programmer on a natural language research project (don’t ask — we did use just assembler) and one summer writing systems-level code in assembler. So I knew both architectures and was able to speed things up a bit. The biggest win was profiling the execution and noting that on the 360 by far the most executed instruction was L (load). I optimized the simulator to identify as quickly as possible when the instruction was L and then treat it as a special case. This slowed all other instructions by one or two instruction cycles, but it was worth it. Most of the other optimizations were minor after that, but I did manage to use the INT (interpret) instructions in a very clever way to break out the address field, the op-code, and the base/index register from the “normal” instructions more quickly than had been done in the FORTRAN. Ken Bertran had told me he never figured out whate INT was intended for (he never saw a use for it) and another programmer told me he only used it to load a halfword into a register without sign extension (LH or load halfword) extended the sign bit so the 32-bit value was the same as the 15-bit value in two’s complement arithmetic). I was quite proud of myself for that, but probably only I felt that way.

I can explain the “data structures” that the A-system used. They were rather straightforward. Mostly copies of hardware register contents because the Sigma-7 could not read back most of the special registers such as for memory mapping and protection and the information was ot served up anywhere when a memory fault was

triggered, so you hade to decode in software what was going on, including simulating virtual memory to get at the actual instructions in physical memory. Butler Lampson told me (more than once) that he had been very explicit with SDS that no register should ever by write-only. They didn't listen. If they had, I suspect the Sigma-7 would have been that much better and quite possibly Xerox PARC would have adopted it so that today XDS might still be around and if Xerox had listened to Lampson and the others the Alto might have become an XDS product rather than storming the market after Apple took away most of the ideas. Many of which were freely available, but PARC had such a commanding lead and a powerhouse team, Xerox "could have been a contender".

> Ken

That's all for now. I suspect you recognize the contender quote from Marlon Brando in "On the Waterfront" but it also comes up in Jackson Browne's "The Pretender", which is something I have running through my mind these days (Gee, that was another two Jackson Browne call-backs "These Days" and "Running on Empty" which is sort of how I feel).

Kelly

Subject: Re: XPL and GORDO OS query
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 3/11/25, 11:52 PM
To: ken rector <kdrhoo@yahoo.com>

Yes.

A few more comments.

> On Mar 11, 2025, at 11:17 PM, ken rector <kdrhoo@yahoo.com> wrote:
>
> Maybe I have the picture?
>
> The code in the memory image from 0xc0000 - 0x13dff is the A-System that contains low level OS functions.

Yes. This what would not be called the kernel, the permanently resident portion of the O/S that runs with full permissions for the hardware and does not use virtual memory.

> The B-System code will run at 0x1E000 - 0x1FFFF and provides the transfer vector at 0x1E000 that the command Language Processor uses to call the functions for those services. The first couple of services are documented in the reference manual. We are missing this code.

Yes, but the addresses are virtual addresses in a user processes.

User processes run without permission to do anything regular instructions (no I/O, no context switching, no manipulation of the memory mapping or memory protection registers, just fetching and storing in those parts of virtual space that are available to the process and the usual arithmetic and logical operations. At any time, a subset of the 256 pages in virtual space are associated with specific pages on disk. The A-system insures that pages that are not associated with disk pages are inaccessible (no read, write, or execute permission) and that those that do have disk pages can be accessed but only if the page is in memory (in which case the A-system guarantees that the virtual page will map to the correct physical page). The A-system doles out access a bit at a time, so that once a page is in memory it can be read or executed by a user process, but not written. An attempt to write is permitted if the file allows writting (ditto for execute too) and if that happens, the first write attempt causes the A-system to allow further writing and the A-system marks the pages as “dirty” so it will eventually be written out to disk before being over-written. A key feature is that there is never more than one copy of any disk page in physical memory, so all processes share that same copy even if the page is mapped to different virtual pages.

Users processes are further constrained to run in one of two modes, which is enforced by the A-system via memory protection. Initially none of the pages after 0x1E000 allow execute or write (I think read was OK). The only way a user process can gain permission is to make a subroutine call into the transfer vector at the start of that space. When that happens, the A-system detects it as a page fault (lack of permission to execute) and it notes that the process is now in B-system mode so that subsequent execute attempts or write attempts to the writeable part of the B-system address space is allowed. When the B-system is ready to return control to the calling routine, it asks the A-system to lock down the B-system memory again and resume execution in the lower part of the user process address space.

B-system services include open / close files into one of 16 keywords, map page N of the file open in keyword K onto virtual page M in the user process’s virtual space, sleep until a wakup signal from some other process, wakeup another process, interact with the graphics displays (a distinct subset of the B-system that gets passed

on to the A-system that had the device drivers), execute an I/O instruction on an external devicce (only privilged user processes are allowed to this) and the related “lock a page in memory” so the I/O instructions do not have to deal with virtual addresses, plus create a new file into a keyword or make a new entry in a directory that is open in a keyword for a file thst is open in another keyword (and setting a name in the directory for that file), and of course delete an entry from a directory (if the ref count goes to zero when this happens, the associated file disappears and its disk space is reclaimed, but the ref count includes being open in a keyword so not all files are necessarily in directories).

> The code that assembles to run at 0x1d000-0x1dfffis known as MASTER and is the Gordo Command Language Processor and is the users command interface. This includes a bootstrap routine to read in some stuff somewhere.

Yes, I think. I am not sure why the addresses would be up against the B-system, but maybe that was the case. Perhaps it was because the CLP could act as a debugger with “regular” code in the lower addresses. That may well be the case. I actually forget how that worked. It would make sense, I just do not remember. If it was like that, I am not sure what the mechanism was for a program to return to the CLP but maybe I am forgetting some bit of code that all applicatons had that essentially made them look like subroutines to the CLP./

The CLP is runs another user process. There is nothing special about it. It reads input from the keyoards on the graphics displays and then works more or less like a command line shell in Unix or other sysstems.

Surprisingly, although I designed ad wrote the CLP, I remember the least about it. Probably because I did not have to study it, as I did the A-system and the B-systeme, which were written by others. And also because it was a pretty simple piece of code with a very rudimentary scanner for text commands and a very straight forward set of things it coud do. Its capability as a debugger was very limited. It could set breakpoints (using hex addresses), display memory content, and patch values, but it could not (for example) catch changes to memory because that would have requred hardware support.

Subject: Re: Old LLNL systems and source codes
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 4/14/25, 1:01 PM
To: ken rector <kdrhoo@yahoo.com>

This is the Gordo Manual version 2.1. Authors are John Beatty and Charles Wetherell. Date is 1973 March 28 although the running header says 1972. One of those dates is a typo. This would have been after the SEX Manual was done at UCLA, I think.

John Beatty took over the system code from me. Charles Wetherell was a user and did some systems work (he had two or three stints at Livermore).

This was definitely produced well into the lifetime of Gordo when it was largely in “maintenance” mode.

John Beatty passed away in 2020 (I was the executor for his estate). Charles Wetherell is still among us. I may have contact information for him.

This really was a User Manual, not a system manual.

The EDIT program was written by Doug Gotthoffer quite early on. It was simple, but effective.

SYMBOL and FORTRAN were the SDS/XDS products with minor modifications to their I/O so they ran under Gordo (otherwise they were just as on SDS/XDS systems).

DLOADX was I think the SDS/XDS loader (again, more or less as it was originally) but I think there were some modifications made (not sure what they were). I definitely fixed an egregious mistake (I think I already mentioned that) where the loader was maintaing a literal queue of fixup addresses that should have been a stack. Order made no difference in the semantics, but a queue as coded meant terrible performance in a paged environvment.

FLD and PEEK were small utilities. I believe FLD took the output of the DLOADX (which would be binary files with semi-compressed core images) and loaded them into a process’s virtual memory or perhaps did some minor conversion to whatever format Gordo wanted instead (I cannot remember that part).

XPL and SLRK were the XPL compiler and the Simple LR(k) grammar analzer (think “yacc” on Unix but without most of the bells and whistles, just the grammar analyzer and table generation for the table-driven parser in the XPL compiler).

> On Apr 13, 2025, at 5:02 PM, ken rector <kdrhoo@yahoo.com> wrote:
>
> <102686951-05-01-acc.pdf>

Subject: Re: A-System memory
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 6/26/25, 5:06 PM
To: ken rector <kdrhoo@yahoo.com>

See comments below.

> On Jun 26, 2025, at 12:30 PM, ken rector <kdrhoo@yahoo.com> wrote:
>
> I'm kind of interested in GORDO and I'm thinking about maybe recreating a simplified version starting with your MASTER, XPL and the reference manual. I'm not sure how far I might get with this but I have time just now to think about it. I wonder if you would mind if I ask you things like this once in a while?

I am happy to help out with whatever information I can recall, which may not be much.

Here is a recap of what I recall from previous emails:

You do not have a copy of the A-system code (you thought you had a binary, but it ended up being the binary for the XPL compiler).

You have the assembler source code for the Gordo Master program that I wrote, which is essentially a very simple shell-like command language that launches processes and performs some debugging and other utility functions.

You also have the Gordo manual, which describes Master and some other parts of the system from the viewpoint of a user (i.e., the manual is not an implementation document), describing the text editor, assembler, Fortran compiler, and other utility programs many of which were “lifted” from SDS..

I can probably describe the various B-system functions that are not documented, at least at a high level. Most are easily guessed from the name.

I can definitely describe the interplay between the Master, the B-system, and the A-system but not down to the bit-level.

> I had several years experience in development and maintenance of CP-V back in the day but since I don't have instant recall about CP-V, I appreciate that may also be the case with you and GORDO, and I'll be happy with any reply or, if your are busy and don't have time for questions, just say so. You've already helped me get started.

I think the most interesting aspect of Gordo is the minimal design and how virtual memory was handled, including the notion of “coupling” pages, which is not really found in other systems that have virtual memory. Unfortunately, a lot of the very important details about how context-switching is performed (especially when mutual exclusion is required) are not things I am going to remember very well. But I can provide some insights into how (for example) inter-process communication of data was done via the file system (rather than through a mechanism like a Unix pipe), including the mailbox mechanism that as I recall linked each mailbox with a process (although I am fuzzy about the nature of the linkage — I can explain what this means).

> It's really disappointing to know that there was a GORDO system submitted to the Xerox User Group Library that seems to have disappeared. Darn!

Yes. Source code for the A-system and the B-system would be very nice to have. I think I could decipher most of it and from that to gain a better understanding of the specifics of context-switching and some of the mutual exclusion issues (an example being how disk pages are allocated to files).

More thoughts later.

Kelly Booth

Subject: Re: A-System memory
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 6/27/25, 6:18 PM
To: ken rector <kdrhoo@yahoo.com>

> On Jun 26, 2025, at 9:16 PM, ken rector <kdrhoo@yahoo.com> wrote:
>
> As I remember, The CP-V kernel that did all the privileged stuff was mapped one to one and occupied low memory below 0xa000. So all user programs and processors were biased there. The kernel was able to see the current user above 0xA000.
>
> Then is it so that in GORDO, the A-System runs in physical pages in low memory and isn't found in any virtual memory. Doesn't the A-System need to access user pages? I guess not. Does it just live by getting data to and from the user via the B-System?

The A-system ran in physical memory. Its role was to support the virtual machine model that user programs saw, which included the B-system as part of a user program serving a role similar to traditional operating systems that shared memory space (in this case virtual memory space) with user code and data. User programs never interacted directly with the A-system, which existed outside the virtual machine. They did interact with the B-system, which ran within the virtual machine.

I assume you are American, so you may have heard this before:

> Here's to the town of Boston,
> The land of the bean and the cod,
> Where the Lowells speak only to Cabots,
> And the Cabots speak only to God.

So it was in Gordo. User programs interact with the B-system, which in turn interacts with the A-system. This restriction is literally true if you consider service requests. It is not true if you consider page faults, but page faults are not anything a user program can predict or control. They are in fact "outside" of the virtual machine in which User programs reside. That same virtual machine is also where the B-system resides, sharing virtual memory with the user program through a mechanism that is essentially a very simplified version of Multics rings.

The B-system ran in the high-address portion of every virtual machine. The Master program ran in the next highest addresses, but unlike the B-system it was not "protected" or priviliged in any manner and it could be over-written by a user program if the user program so desired. The B-system could not be over-written or modified in any way by a user program and it could only be entered via specific entry points defined in the B-system call vector (a range of addresses within the B-system portion of virtual memory).

Aside: Multics rings were layers within the software that (as I recall) partitioned the memory segments in a hierarchy (multiple segments could be in the same ring but any segment was entirely within a single ring). Any transfer of control (branch or jump instruction) that crossed a ring boundary triggered code that ensured that proper access control of execution privileges were appropriately adjusted.

In Gordo, user processes ran in virtual memory. The high-address pages were reserved for the B-system (program and data) and the low-address pages (most of virtual memory) were reserved for the user program. For the rest of the discussion, I will make no distinction between the Master program and the user program — because much like in early operating systems without access control hardware the O/S and the user programs

essentially ran as one program where control simply went back and forth between the two parts without any oversight (hence the O/S could be corrupted if its date or program was overwritten). This was the relationship between a user program and the Master program.

The relationship with the B-system was similar to later operating systems where there was some degree of hardware protection separating the system and user programs that depended on the hardware. For Gordo, the access control hardware provided the guardrails that kept a user program from abusing the B-system. If a user program needed to do something that user programs were not allowed to do for themselves, the user program invoked the B-system. This was done by a simple subroutine call (normally a BAL or branch and link instruction) but any branch instruction could be used with the understanding that whatever was in register 13 (I think that is correct) was the return address and register 14 had either parameters or the address of parameters for the B-system call. The trick was that when user code was in control access to the B-system pages in the process's virtual memory were read-only. So any attempt to execute instructions or modify memory in the B-system (including a BAL to the B-system) would trigger an access control fault that was handled (outside the virtual machine) by the A-system.

The A-system handled access control faults (traps), regardless of whether a user process was executing in the user program or in the B-system. This was invisible to user programs because it was outside the virtual machine. The A-system kept track, for each process, of whether the process was running user code (low-address instructions) or B-system code (high-address instructions). Every page in virtual memory for each process was either unused (no disk page was associated with it and thus it was not accessible to the user process) or it had a disk page "coupled" to it. Attempting to access a page that was not coupled resulted in the user process aborting (whether it was executing user code or B-system code, although the B-system was written so it would never do that, so in practice only user programs could trigger a process abort). The process had a map of its 256 pages in its virtual memory that was in the B-system portion of its address space, with each entry containing the disk address of the page that was coupled to the corresponding virtual page (or a zero indicating no page was coupled) and it had access control bits mimicking the hardware access control register settings to indicate whether the process had read, write, and/or execute access to the page. At any given time, the hardware access control registers reflected this, but usually only for a subset of the pages because not all of the pages coupled to a process were in memory.

Initially, the hardware access control was set to disallow access to all pages for a process. This meant that the first instruction could not be executed because it could not even be fetched from (virtual) memory. This caused a fault and the A-system went through a calculation to determine which virtual page was being accessed, what type of access was required, and — this is the critical part — whether the page was in physical memory and if the type of access requested by the process was allowed. If the page was not in memory, the page was put in the queue of requests to be brought into memory by the A-system. When the page eventually arrived, the process would be allowed to run again but the exact same fault would be triggered again (the hardware access control registers would still be set for no read, write, or execute access for that process) but this time the A-system would see that the page was in memory and it would set the corresponding mapping register to map the appropriate virtual page to the physical page where the disk page resided and then it would set the access control register for that virtual page to permit read and/or execute if (and only if) the process was allowed that access. If the process was not allowed that access, it aborted, but otherwise the A-system resumed execution of the process in user mode (memory protection / access control and memory mapping enabled and privileged instructions forbidden). If the access required then or subsequently was for writing, yet another fault would be triggered and the A-system would check to see if the process was allowed to write into the page and if it was the access control register for the page would be set to permit that and the disk page would be be marked as "dirty" so the A-system would not to write it out to disk to ensure that whatever changes were made were preserved. If write access was not allowed, the process would abort. All of this worked the same way for instructions in the user program or instructions in the B-system. The A-system simply assured that whenever a virtual page was access by either

part of a process (user program of B-system) the page was in memory and the harware mapping and protection registers were properly set to map the virtual page to the correct physical page and allow whatever access was permitted to the process (but not more access than permitted).

As long as a user process was executing in low-memory (outside the B-system), execution continued in this manner with the A-system managing virtual memory and ensuring that the process did not improperly access virtual pages that were prohibited or that were not yet in physical memory. If the user program transfered control (branched or jumped) to the B-system, this caused a fault because none of the B-system pages had their access control registers set to allow execution. So the A-system intervened, as usual, but when it detected that the address of the instruction causing the fault was in the B-system high addresses and the process was marked as executing a user program, it treated the fault differently. A check was made that the instruction address was strictly within the transfer vector for the B-system (i.e. it was a legitimate target address for a B-system call). If it was not, the instruction address was set to correspond to a software-simulated trap that the B-system would handle by changing the address to the address within the B-system transfer vector corresponding to the trap handler. Either way, the A-system then restarted the process which subsequently would typically page fault as it needed access to B-system pages but because the A-system knew that the process was executing B-system instructions (it had set a flag when the transition happened) it allowed read-execute access to code pages in the B-sytstem and (when required) write access to data pages in the B-system. One of the data pages was the PACT (Process Activation Page) that was unique to each process. This has information specific to the process, including the 256 pages that were coupled into the process's virtual memory. The B-system code pages and the PACT were by default coupled into the high-address pages for each process. The B-system would also couple other pages it needed, such a file header pages when a file was opened or a page was being coupled by a user program. These were ephemeral and changed as the B-system did its work.

Aside: The Master ran as a user program. It was located in the upper part of the low-address space reserved for user programs. It shared that address space with the actual user program with no protection mechanism for its data (its program instructions were protected because the pages were read-execute with no write permissions, but user code could jump into any part of the Master so it was not secure in any meaningful sense and its process-specific data could be corrupted by the user program because it was writeable.

The virtual machine that user programs saw was essentially an SDS Sigma-7 but without any of the privileged instructions (no I/O instructions, not memory mapping or memory protection instructions, and no instructions that would change these settings). So the computation model was one of a CPU and up to 128K words of memory where not all of the pages might be present and some that were present might have access lilimations that might prohibit reading, writing, or executing. Access to B-system pages was restricted to read-only unless a process was running in B-system mode (as previously described, the A-system enforced the transition back and forth between B-system mode and regular user program mode using the access control registers and monitoring when a user program branched to a B-system entry point or when the B-system indicated that it was returning to the user program after performing a B-system service.

There was no notion of I/O for processes (except for graphics displays, which are a separate issue that I can explain some other time). There was also no inter-process communication other than some siimple signalling (any process could wake up another process) and the abiliity to fork and kill other processes with a job (a job being a collectionn of processes). Data was only shared by one or more processes coupling the same disk page into their respective virtual memories (not necessarily in the same virtual address). This could be concurrent (i.e. two processes could couple the same page simultaneously and thus communicate using a circular buffer — for example — using well-known protocols that relied on semaphore-like techniques utilizing regular non-privileged instructions such as Exchange Word or Modify And Test Word. Data could also be shared simply through the file system where files would be created and written to and then put into directories where other process could subsequently read them as in traditional operating systems.

I/O was of course necessary, so certain privileged processes were allowed to issue I/O instructions, but the A-system trapped those and did a bit of magic to ensure that they had the correct physical addresses and that the process was not doing anything that accessed memory outside its virtual space. I believe that the integrity of this depended on trusting that the privileged processes followed the rules, which is why only trusted processes written by the systems programmers were allowed to execute I/O instructions. There was one process each for the card reader, card punch, line printer, and tape unit. Regular user programs accessed these through the shared file system using directories that were designated as mailboxes. A process could put a file with card images into the Card Punch mailbox and the corresponding process would wake up (because a new file had been added to the mailbox) and that process would execute the I/O instructions necessary to activate the card punch hardware. The card reader would use the header card at the start of a card deck to determine the name of a new file that would receive the information in a deck of cards and the user ID for which the card deck was intended. The file went into the Card Reader mailbox with access restrictions that allowed only processes belonging to the intended user to retrieve it from the mailbox. I think for input mailboxes, a process opening a file successfully automatically deleted the file, but perhaps that had to be done by the retrieveing process.

As you can see, the computation model was one whose primitives were the “normal” machine instructions, the concept of files and pages within files (and directories of files) and the notion of coupling pages from files into a process’s virtual space all wrapped up with the ideae that at any time a process was either in user mode or B-system mode that provided sufficient protection for the B-system against errors or malicious behavior on the part of a user code. The A-system was not part of the computation model. It’s role was to implement that abstraction of the virtual machine but with the B-system doing most of the work related to the semantics of the file system and process creation and destruction.

END OF DESCRIPTION

Subject: Re: A-System memory
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 6/28/25, 2:53 PM
To: ken rector <kdrhoo@yahoo.com>

There are a number of questions I have about the specifics of how the B-system and A-system interact for actions that the B-system takes that I need to think about. Having source code (even though it is assembler) would have made this much easier, but we seem not to have that.

A lot of questions are related to how disk pages are managed. When the system image is initialized, there are disk pages that have the B-system code (instructions), and at least one disk page that is the PACT for the first process that exists, plus disk pages that have user code for that process. The initial process would work as follows (I am making this up from my knowledge of how things probably worked — not how they actually worked because I never was involved with the code for the system initialization at this level — it was written and worked and there was never a reason to modify it when I was in charge of the system).

a) The initial process would create the various processes, such as the logger, the I/O processes for the hardware devices, and the garbage collector (the nature of the file system was that there could be circular links runninig through directories, so simple reference counts would not identify disk pages that were no longer accessible). I assume that the code files for these were part of the initial disk ocnfiguration and there might have been a directory that has these files in them that was “hand built” and part of the disk image. The Gordo manual talks about this, but I have not reviewed it in detail. As part of this initialization, I assume the iinitializer process would also create the rest of the basic directory structure, including the mailboxes, and it would associate the respective I/O processes witht he corresponding mailboxes via B-system calls that would make the connection. I think the A-sysem would not have any direct involvement in this (just the normal supporting role maintaining the virtual machine model, as I previously wrote).

b) Part of the initialization would be the creation of at least one user account that corresponded to the system that would own the logger and probably the I/O processes. User programs running under this user ID would be able to update a file owned by the system user with new user accounts (IDs and passwords) and create root directories for the new accounts whose names would be the user IDs.

c) A free list of disk pages had to exist for the B-system to use. I am not sure if that was part of the hand-build system image on the disk, but I suspect so. I suspect it might have been just a simple bit-vector indexed by physical disk page address or it might have been a linked list (which would have presumably required a halfword or double word for each entry, one for the disk page address and one for the link depending on how large those numbers might be). Originally the system had only one disk. I forget if when we upgraded to a bigger faster disk the old disk was still used, but if so the system either already had provision for multiple disks or this was added. That would mean multiple bit vectors. One question I have is about the exact mechanics of disk page allocation and deallocation.

Obviously when a process coupled a disk page that was not already existing in a file (the couple B-system called had three parameters: a keyword that identified an open file, the virtual page into which a disk pages was to be coupled, and the disk page within the file that was to be couple to the page in the process’s virtual memory). If they keyword did not have an open file, it was an error. If it did, the B-system would couple the file header into one of its virtual pages and reads the header information to see if the file page requested existed. If it did not, the B-system would allocate an unused disk page and assign it to the requested page in the disk by modifying the file header. The B-system would then put that address of the disk page into the process’s PACT in the slot in the map page corresponding to the virtual page specified in the couple request. The B-system would then signal the A-system that it wanted to return to the user program at the location specified in register 13 (or whatever the

convention was). This was done by the B-system executing an instruction that trapped to the A-system, which did not do anything except toggle the mode for the process from B-system mode back to user program mode (this was a bit in the subpact that the A-system kept in its memory for each process) and also reset all of the access control registers for the B-system virtual pages to prohibit execute or write access (i.e., return the virtual machine to one in which the user program was restricted to working outside the B-system virtual space).

The A-system then restarted the process. The new page that was coupled would not be brought into memory until the user program first referenced it, at which time the usual page fault mechanism would confirm that there was a disk page coupled to the virtual page and the A-system would fetch the page from disk and then set up the mapping registers for the process so the physical page was the mapping target for the virtual page. This was typical of the minimal role the A-system played in the semantics of the virtual machines (including the file system). The A-system had no knowledge of when new disk pages were allocated. My one question is how the B-system manage mutual exclusion when the B-system for a process was allocating (or deallocating) disk pages. The usual problem of not allowing two processes to claim the same page would have to be resolved. As far as I can remember, there was no easy Sigma-7 instruction that would remove or add entries to a linked list as an atomic operation. I think the solution was a simple “lock” that the B-system employed that probably used something like Exchange Word to write a 1 that in a register into a word and fetch the previous contents of the word into the register as an atomic operation. This would be a simple P-V sort of mutex that would allow the B-system to know that some other process was accessing the disk free list if a 1 was the result, in which case the B-system would perform a sleep operation and then try again when it woke up. A more elaborate scheme might have been used, but I suspect not. For allocating and deallocating disk pages there would be no not much danger of circular dependencies that might cause deadlock. I think if the lock were set just before the B-system checked to see if there was a page allocated in the file and then unlocked once there was a page allocated, all would be well and nothing in the semantics would lead to deadlock.

Deallocation would be similar, which took place when a file was destroyed, which I believe only happened when it’s reference count when to zero (references being either entries in directories or the file being open in a keyword — the garbage collector would find circular chains in unreachable parts of the file system and remove the links and that would eventually lead to closing the last open keyword for such files after which deallocation would take place). I am again making this up based on what I know of the system and how the most logical way of implementing things would be. The actual deallocation would be done when it was known that no process could have access to the page (except the one whose B-system was deallocating the page). The page was zeroed and then returned to the free list, which would involve locking while the entry in the file header was set to zero and the page was added back to the free list. It is worth noting that the A-system knew nothing about any of this. It only knew about disk pages, it did not know about directories, files, or keywords. What it saw was that the page had been read into physical memory because some process wanted to write into it and because it was written into it was dirty and needed to be written back to disk. If for some reason the disk page was reallocated before it was written back to the disk, that would not be a problem because the copy in memory had been zeroed and either it would eventually go out to disk as all zeroes or it would be written into by some other process accessing some other file for which the page was not a part and the A-system would know that a subsequent write had taken place so the page would again be dirty. The one implication of this is that the system could not be shut down until all processes had been stopped and the A-system had written all dirty pages back to disk. I am pretty sure this was how the A-system code worked. If the system was stopped abnormally, there would of course be no guarantee that all of the disk pages would have been properly written back to disk — but that would be true of any system. The mechanism for making a checkpoint of the system presumably took this into account and gracefully halted the A-system after it had written out all pages, and I assume it also waited until all processes had exited B-system mode, which would have been easy to do because the A-system would simply not schedule any process that was not in B-system mode and for those that were they would eventually return to user mode (this of course assumes the B-systems could not have deadlock but that was part of the B-system design and almost by definition deadlock could not be created by user code (at least deadlock that compromised the system

— user codes could deadlock themselves in various ways but that was not a concern for Gordo itself).

END OF THIS INSTALLMENT

Subject: Re: JOBS,
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 6/29/25, 4:03 PM
To: ken rector <kdrhoo@yahoo.com>

Responses embedded below.

> On Jun 28, 2025, at 8:42 PM, ken rector <kdrhoo@yahoo.com> wrote:
>
> There are a limited number of user graphic consoles, maybe 6? That means a limited number of JOBs equal to number of consoles plus system tasks like the A-System SBL, Logger, I/O and Garbage Collector. The JOB Table is then, rather small and can be assembled in the A-System code.

I do not know how my system jobs there were, but certainly no more than you listed. I am not sure if the I/O ran as a single job with a process for each device, or as a separate job per device. Either way, the total would have been small. All other jobs would be associated with a user logged into a graphic display. I forget how many consoles we had (I think four), but we only had two graphics controllers (one local and once remote in another building) and each was capable of driving I think up to four consoles.

I am not sure that the A-system kept track of jobs. It kept track of processes, but I think it might have only been the A-system that kept track of processes. The A-system had a subPACT for each process. I think the documentation may say what was in it, but it would have been pretty simple: job number, process number within job, disk address for the PACT for the process, at least one pointer used to maintain various process lists (the queues maintained by the A-system to determine which process ran next), and some minimal status flags (such as whether the process was awake or sleeping).

I forget how the A-system handled the graphics displays (which is weird because I wrote the code — but not the part related to scheduling and the association of processes to displays). Each display knew the job it was associated with and it might have known which process to wakeup if a lightpen event, function box status change, or keyboard input arose. But that would have been a small bit of information. So the subPACT would have been just a few words (I sort of remember eight words, but that could wrong). There was no inherent bound on the number of processes. Jobs could and did fork off subprocesses. The A-system would have had a limit on these. I cannot remember if there was a static number of subPACTs or if this was dynamic, but I think it was static because the A-system did not storage allocation except for the graphics display lists and that was done from a fixed storage pool (I wrote the code for that part) that was only used for storing display lists (instructions for the graphics processors) because display lists had to be accessed by the I/O hardware and thus located in physical memory (memory mapping was not performed by the I/O hardware and obviously display lists could not be paged in and out of core memory they had to be resident at all times because the I/O ran asynchronously with the CPU).

> The job table records information about the user and the processes for each job. Maybe a chain of PACTs, or pointer to PACTs on a queue.
> The JOB Table entry contains things like the user name, user ID, pointer to process chain.

That sounds right. If you are looking a documentation describing this as being in the A-shystem, it makes senses that there would be a table of jobs. The chain for processes would be to subPACTs. PACTs were only dealt with by the B-system other than the A-system needing to know which disk page was the PACT for a process because it had to access the PACT to handle page faults when a process was running (but when a process was not running, the PACT might not be in physical memory). I am not sure if the subPACT or the PACT kept track of the program counter and other information for processes that were not running (it could be either way). All of this was in the Program Status Doubleword, which I am guessing might have been in the subPACT although there are

reasons it might have been in the PACT.

> The I/O job has a process for each device, card reader, card punch, line printer, operator console and magtapes. There are mailboxs for each process.

Right. No reason to have separate jobs.

> There is one computation file for each JOB called USERCOMPF. It contains the sub-pacts for each process of the job.

That too makes sense. I believe that one of the keywords always had this file open (and probably users could override that) because the PACT had to be coupled into the process's page map and that required knowing the keyword because if a keyword was closed all of the pages coupled from the file open in the keyword had to be uncoupled.

> Page 84 suggests that the A-System code begins at location x'200'.

Yes. That would be first second page of memory (first page if you number from zero). The low 512 words were generally reserved by the hardware. The first 16 locations were actually the registers — not physically, but when addresses in the range 0-15 were used the corresponding registers were accessed rather than core memory. The I/O hardware did not do this (so 0-15 referred to memory) and if memory mapping was in use and a virtual page other than the virtual 0 page mapped to the physical 0 page the hardware referenced the memory not the registers because it was the CPU that enforced the convention that 0-15 were registers.

Low memory included trap and interrupt locations (a single word each) that were executed in priviliged mode without memory mapping or memory protection but with one instruction cycle of uninterruptibilty. Usually these were XPSD instructions (Exchange Program Status Doubleword), which had the address of a pair of doublewords, one where the current PSD would be stored and one from where the new PSD would be fetched. Uusally the new PSD had interrupts disabled so the trap or interrupt handler had a chance to save registers and other context before a subsequent higher-priority interrupt might come in. Once context was saved, interrupts could be enabled again, which was important for the I/O processes.

The A-system took advantage of there being multiple register blocks on the Sigma-7. I forget how many we had (I think up to eight were available but I am not sure if a configuration could have less then eight but more than one — the extra blocks were definitely an option). This meant that when the A-system was called by the B-system no registers needed to be saved, although for page faults the A-system often needed to know the contents of the user process registers in order to decode the virtual address and access mode (read, write, execute) that caused the page fault. I forget if the A-system did in fact save user process registers to memory, or if it is only got the ones it needed (this would be done by switching register blocks to the user process block, storing the register in question, and then switching back to the A-system register block — a total of three instructions plus another instruciton to load the value of the register into an A-system register after the second block switch so it could be used in the decoding steps).

In general, the A-system would have had to follow most of the rules that any system running on a Sigma-7 did because those rules were dictated by the hardware. The B-system and user code would have been less constrained because the virtual machine presented by the A-system did not have any of the extra hardware features such as traps, interrupts, and protection or mapping registers.

> Page 85 suggests that the A-System System Bootstrap Loader is 4 pages, 2K wds.

That makes sense. It would not be doing much. It mostly needed to initailized some of the A-system. I read that part of the manual a few days ago, but now forget if this also initialized the disk or if this was the bootstrap from the disk. Either way, it would have needed to start up the first process and that would required at the very least reading in the PACT for that process, which would have the disk pages for the B-system coupled (meaning those entries in the PACT would be set). I believe that the PACT for that process was part of the bookstrap process (right?).

Kelly

Subject: Re: JOBS,
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 6/29/25, 9:55 PM
To: ken rector <kdrhoo@yahoo.com>

> On Jun 29, 2025, at 8:55 PM, ken rector <kdrhoo@yahoo.com> wrote:
>
> Does the B-System call the A-System with the CAL1 instructions? It seems obvious to me but I have to ask as the manual doesn't mention CAL1.

I don't remember how the B-system invoked the A-system. I know about the CALL instructions, but my memory (vague) is that for some reason these were not used.

Reading page 71 of the Sigma-7 manual, it seems that a CAL1 instruction uses the 4-bit register field to specify an offset so that when a user program invokes it, the CPU traps to location x48 where there is an XPSD instruction. If the XPSD had a 1 in bit 9, the next instruction is not fetched from the address in the new PSD, but rather that address plus the 4-bit offset. It appears that the address field in the CAL1 instruction is jot used at all.

There are four CALL instructions. So there could be up to 64 services that user programs could invoke if the offset mechanism were used. The set up would be an array of 64 branch instructions, each going to code that handles a specific service call. The array could be located just about anywhere in memory (other than locations dedicated to hardware functions). Locations x48-x4B would each have almost identical XPSD instructions that would point to blocks of double-word pairs (each pair the new and old PSDs). The program counter part of each new PSD would be the address of a block of 4 of the branch instructions. The double-word pairs need not be consecutive in memory, but it would make sense to do that.

After the code for each service call was executed, the service call knows which of the four CALL instructions invoked it, so it knows where the old PSD is in the double-word pairs and it executes a Load PSD instruction to return execution to the user program.

The problem with the CALL instructions is that they are not privileged. So a user program not in B-system mode could invoke them. What I am not sure of is whether the CAL1 (or others) was used anyway, but the A-system checks to see if the user program was in B-system mode and if not it either ignores the instruction and returns to user mode, or it aborts the process as if it attempted an illegal instruction. It would just be a few instructions so not a lot of time if the program is in the B-system. If the program was not in the B-system, it does not matter how much time it takes because the process made a mistake (user programs should be execute CALL instructions).

HOWEVER, I think that because the SDS operating systems probably used CALL instructions, Gordo may have considered them to be legal and when they were encountered in a user program (not the B-system), the user program could have a software trap handler. I did not work on the application code that ported the various SDS programs such as the compiler, assembler, etc., but know that there was code written to run on Gordo that caught attempts to invoke the SDS operating system performed the corresponding actions in user mode on Gordo so those applications programs would run in user mode on Gordo. If this is correct, the B-system would have used some other mechanism. I do not remember what that was. It would of course be in the source code (and no doubt well commented), but I do not remember that part because — as with lots of other things — that was a mechanism that was basic to how Gordo worked so it had been implemented and did not need to be changed by the time I took over the source code.

Kelly

Subject: Re: JOBS,
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 6/29/25, 10:28 PM
To: ken rector <kdrhoo@yahoo.com>

> And, do you recall any kind of model number, or manufacturer for the Graphics terminal?

I doubt you will find anything about the displays. I have looked on the web in the past and could not find anything.

They were custom-built for Lawrence Livermore Lab speciically for the Gordo project by Information International Inc.

https://en.wikipedia.org/wiki/Information_International,_Inc.

III ("Triple-I") did a lot of things related to computer graphics, but most of their products were digrital scanners (of film), digital recorders (onto film), and some software under contract. Triple-I was once a leader in computer animation and special effects software, because of their film recorders and the team of application software programmers who at one time were at the top of the field, but Triple-I eventually moved out of that and merged with Autologic and specialized in computer-based typesetting and related areas.

The Triple-I Wikipedia page makes brief mention of "a number of one-or-two of a kind systems which included CRT-based computer displays used at the Stanford AI Lab" but no details. I am not sure if those were similar to the Livermore displays. I think not. The graphics displays were delivered Summer or early Fall of 1968. Triple-I's FR-80 came out in 1968, which was their major success. I think they lost interest in graphics displays and focused on the FR-80 and related products (Livermore eventually purchased two FR-80s).

I will try to remember the model number. It was something like TSD-1040 (time-shared display with 10-bit addressability — it was a vector system because this pre-dated the raster graphics era). It was called time-shared I think because the graphics controller could control up to four displays and drawing commands could be directed to any subset of the four displays. (Gordo never made use of this feature. Each display had its own set of commands and were executed in sequence so at any given time only one display console was being drawn on even though it was possible to draw on any subset for any subsequence of the commands.)

I met some of the designers when they came to Livermore to install the displays and then later when they came to fix problems. One person I did not meet was Stewart Nelson, who I am told designed the character generator (for drawing text) in the graphics hardware. Nelson left Triple-I to co-found Systems Concepts, which built a next-generation vector display but the company essentially lost out to Evans & Sutherland (and others) in the vector graphics display field. Their Wikipedia page does not meantion any graphics display products.

https://en.wikipedia.org/wiki/Systems_Concepts

The only publication I was involved that that mentioned Gordo was one for a Kriegspiel application. There is only brief mention of Gordo (no real details) and very little about the graphics displays that I have not already described. There are some "screen shots" of the displays, but those provide no insight into the hardware or software. They could come from just about any vector graphics display system of the era. I attached a PDF of the paper. The paper has no references for Gordo or the displays and does not name the display system.

The paper does say that there were six consoles. So my previous email was incorrect when it said I thought we only had four. We must have had four in the main location adjacent to the Sigma-7 and two more in another

building connected via a Sigma-3 computer to the Sigma-7 by a custome “data link” communication channel designed by SDS for Livermore for this purpose. I am not sure if they ever sold other versions of the data link. Perhaps.

Kelly

Attachments:

1972-The_Computer_Journal-Wetherell-Buckholtz-Booth.pdf	2.6 MB

Subject: Re: JOBS,
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 6/29/25, 11:15 PM
To: ken rector <kdrhoo@yahoo.com>

> The various SDS OSs always used the CAL1 instruction to enter the OS for service calls. That is its purpose. The instruction address referenced an FPT, (File Parameter Table) that described the parameters for the service and these were extensive. The CAL1 switched to master mode so the service had access to everything.

Right. I read the manual and see how they were used. The issue is that they can be issued by any user program, so in Gordo the A-system woujld need to confirm that it was the B-system issuing the CALL. That would not be difficult (but it would be different from what an SDS O/S would do because there was no distinction between user program and B-system within a user process, as there was for Gordo.

> Interesting. You think the standard SDS processors, like Symbol and Fortran weren't modified to run on GORDO, and the system interpreted the CAL1 calls. Thats ambitous as there were a lot of possible calls. I have wondered how they ran those processors without a lot of recoding.

Symbol, Fortran, the Loader, and a few other programs were ported to Gordo at some point a year or two into project. Prio to that, software was assembled, compiled, and loaded using the SDS O/S and then the binary executables were loaded onto Gordo. Initially there was no support for the Fortran I/O library, beacuse it made calls to the SDS O/S. Mike Stone, one of the project members, did a port of the I/O and Fortran support library to Gordo. I am not sure if that involved simulating CALL instructions in some way, or he instead replaced the low-level I/O routines with new code that opened files under Gordo and read or wrote them by moving bytes from and to the files rather than invoking low-level file I/O calls to the SDS O/S.

> Do you have any idea how many A-System services might have been provided by the B-A interface?

No, but I can take a guess. There would have been calls to put the current process sleep (perhaps with a wake-up time), to wakeup any project, fork a new process, kill a process, create a job, kill a job, and probably something that told the A-system that one page in virtual memory needed to invalidated (which the A-system could do simply by changing the access control for that page in the current process and everything else I think would work itself out through the normal A-system page fault handilng). There were also a few services related to the display, but only a few. I think there was on call that read a graphics console (which returned the status of the 16 function buttons and keyboard buffer — not sure if those were separate calls or one call), something to test for lightpen hits that the A-system captured, and one call to add a set of graphics commands to the display list and another to delete a set of commands (a set being a bufffer of up to I think 32 words, each command being one word).

> Since the B-System was slave mode code, entered from the user by a simple branch I imagine the B-System entered the A-System and switched to master mode with a CAL1 of some kind. The switch has to be by some kind of trap or cal.

Yes. The entry to the B-system was a simple BAL instruction, but because the B-system pages were not executable, the A-system caught the trap and checked to see that the target address was within the B-system transfer vector and if so it noted that the process was now in B-system mode so the access control was reset to allow execute access and write access to those pages in the B-system that permitted writing (for that process). Details were in a previous message I sent.

> Was GORDO somebody's PHD project?

Good question. Yes and no. Gary Anderson and Ken Bertran wrote most of the B-system and A-system code, respectively. Rob Millstein wrote some of the initial code and Karl Malmquist I think wrote the three-card loader and also most of the Gordo I/O processes (for the devices, for the A-system). But Gary and Ken were doing all of it by the time I was fully installed on the project and I took over when they left.

Gary was in the PhD program in EECS at Berkeley. At one time, my understanding is that he was expecting to have his dissertation be on Gordo. I do not think he ever completed a degree because he and Ken left Livermore ot start a company (Mike Stone went with them as well). This was the GEM system that was a revised Gordo. A search for Gary B. Anderson turns up a faculty member at UC Davis in Animal Science. Not the right person. Livermore had a close relationship with UC Davis, so this is a bit of a coincidence, but just a coincidence.

> I notice in the manual ref section there are several papers about GORDO listed from UCRL and CIE. I think I going to try to track those down.

There are not many. Ref [8] is this publication from an ACM conference.

https://dl.acm.org/doi/10.1145/800186.810559

Refs [9]-[13] might be available through Liveremore. I am not sure what the CIE meeting was (it might be internal to Livermore). I think [11]-[13] might have been conference presentations that the lab published as UCRL (University of California Radiation Lab) reports. If you get copies, please forward them to me.

Kelly

Subject: Re: JOBS,
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 6/29/25, 11:21 PM
To: ken rector <kdrhoo@yahoo.com>

> On Jun 29, 2025, at 10:36 PM, ken rector <kdrhoo@yahoo.com> wrote:
>
> Thats the purpose of the CAL instruction, to give a means to switch modes when required.

Exactly. But in Gordo, the user is not allowed to call the A-system (remember the Lowells and the Cabots in Boston). Only the B-system can call the A-system. But if the CALL instructions were used for this, the A-system could check that it was the B-system calling (as I noted in an earlier email) and allow that but not allow the user program to call the A-system.

There are other ways the B-system could trigger a trap that the A-system could uniquely detect. I just do not remember if some other mechanism was used or if the CALL instructions were used. I doubt very much that is written down anywhere other than in the source code.

Kelly

Subject: Re: swapper size question
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 7/14/25, 11:32 PM
To: ken rector <kdrhoo@yahoo.com>

> I found a linkdin page for Bob Millstein but haven't received a reply.

I managed to track him down a few years ago (i.e., I found a trail of his employment post-Livermore). I cannot remember if he and I actually exchanged email messages because it was just long enough ago that I have forgotten. My vague recollection is that he did reply, but I might be imagining that.

His LinkedIn profile is the following.

https://www.linkedin.com/in/bob-millstein-a8834/

unless there is another Bob Millstein who worked at Livermore. After Livermore (I htink 1963-1968 — the 1978 is almost certainly a typo) he worked as Mass. Computer Associates. Millstein had been in the AI group, which at one time was run by Jim Slagle

https://en.wikipedia.org/wiki/James_Robert_Slagle

who worked on symbolic integration when that was still considered an AI problem — before the Risch algorithm pretty much turned it into solved problem for a wide range of functions.

https://en.wikipedia.org/wiki/Risch_algorithm

Gary Anderson had, I think, been in the AI group although he was junior to Millstein. Slagle had left before I arrived. His January 1968 JACM article with Bursky has an LRL affiliation but his 1969 JACM article with Dixon has an NIH affiliation. Given the delay from submission to publication, and the fact that when I arrived in June 1968 he had left Livermore, I would guess that Slagel might have left in 1967. That would make sense because Millstein probably moved in the Sigma-7 group in part because Slagle's group was shut down when he left. There were a few people still doing AI who reported to Dick Conn (Dick was definitely not an AI guy). Gary Anderson was a LISP person. He "documented" the B-system in LISP (he told me — I never saw it in part because I had no interest in LISP as a documentation tool).

Samara Technology seems to exist as a company. I think that is where I tracked Millstein down a few years ago. My recollection is that he was no longer active in the company, or at least not very active. Not sure if I got that from someone at the company or from him. This was probably pre-COVID, but I am guessing.

If you manage to track him down, let me know. I suggest starting at Samara, but it may be an entirely new company that took over the corporate name (this often happens).

> I believe there were only two RAD models for Sigma, the 7212 with 5.3 mbytes and the 7232 with 6.3 mbytes. The GORDO Reference Manual (pg 1) claims a 6 million word swapping disc but that must have been a typo. He must have meant 6 Mbyte as in the 7232 RAD. The ACM paper claims a three-quarter million word (.18 Mbyte) swapper which is way too small.

You may be correct about the RAD having only two models, although I was quoting from the SDS hardware manual. There is a 1968 RAD manual.

https://bitsavers.org/pdf/sds/sigma/periph/901557A_7231_7232_RAD_Aug68.pdf

that describes the Model 7231 RAD Controller and a 7232 Storage unit. It says each second is 1024 bytes (256 words, as you pointed out). There are 512 tracks (each with its own head!) and 12 sectors per track (so 6 pages).

> https://s3data.computerhistory.org/brochures/sds.sigma.1967.102646100.pdf

where it says:

> Rapid-Access Data Files. RAD units provide storage capacities from 750,000 to 192 million 8-bit bytes per control unit; transfer rates range from 188,000 to 2.2 million bytes per second; fixed read/write head-per- track design eliminates the time delays associated with movable-head disc files and produces average access times as low as 17.5 milliseconds. A no-latency program- ming feature on most models further reduces access time to less than 30 microseconds.

I know for sure that when the Sigma-7 was ordered for GORDO, the larger disk had been announced but was not available. So they purchased a small RAD. We later upgraded, some time in 1969 or 1970, well after the better disk had come on the market because (as I explained before) we had purchased the smaller one and before the performance issues starting hitting us most of the team (including Millstein, Anderson, Bertran, and Conn) had left so no one remembered about the recommendation in Millstein's memo that the larger disk be purchased once it was available because it would be necessary to insure desired performance.

The spec (from 1967) above talks about capacity per control unit. Each RAD mauy well have been as you say, in which case the larger and faster disk we purchased may have been a much better controller with multiple disks.

> I think what I started to ask you about swapper size concerned the difference in memory page size (512 words) and swapper sector size (256 words). Do you recall having to deal with this?

Every page was two sectors on the disk. I do not remember the details of I/O, but I suspect it was possible to specify 512 words (2048 bytes) in a single transfer and the disk simply moved from the first sector to the second automatically. Only when doing seek operations would sector addresses really matter. For GORDO, all seeks would be to even-number sectors. There are details in the RAD manual about data and command chaining and how this affects timing, and also that a single read or write can span multiple sectors.

I forget how many sectors per track or how many tracks. I sort of remember there being 16 tracks but that is a guess.

Assuming the 750K bytes per RAD, that would be 366 pages (or 375 pages if 'K' mean 1024) as I calculated in my previous message. That is not a lot of pages. So perhaps the RAD we had was not the smallest. The system, the system processes, and a few users would require hundred of pages, but not thousands of pages for immediate use. The file system was not used for permanent storage except for key files. Most users had their files on tape and read them in when they used the system, or they used card decks. The amount of disk space was definitely a problem! Virtual space was 128K words, or 256 pages. But most user programs were far smaller than that and the B-system code (which did not take many pages) was shared (except for the data pages) across

all users.

I am sure there must be manuals for the disks online some place.

From what I remember, we changed to the larger disk(s) when I was maintaining the system. I do not remember having trouble reprogramming. So I am assuming that at worst the device address had to be changed in the system code, and perhaps a few parameters such as the number of pages and possibly the track and sector mapping parameters, but whatever had to be changed, I do not remember it requiring much effort. The underly logic of the disk driver was already sufficient for the new hardware.

Kelly

Subject: Re: swapper size question
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 7/14/25, 11:53 PM
To: ken rector <kdrhoo@yahoo.com>

Ken,

This is a bit garbled:

> On Jul 14, 2025, at 11:32 PM, Kellogg Booth <ksbooth@cs.ubc.ca> wrote:
>
> After Livermore (I htink 1963-1968 — the 1978 is almost certainly a typo) he worked as Mass. Computer Associates. Millstein had been in the AI group, which at one time was run by Jim Slagle.

I meant to write:

After Livermore (I believe Millstein was there 1963-1968 — the 1978 end date in LinkedIn is almost certainly a typo), Millstein worked as Massachusetts Computer Associates. At Livermore, Millstein had been in the AI group, which at one time was run by James (Jim) R. Slagle, before he moved to the Sigma-7 project.

I assume Millstein was one of the founding team members for the Sigma-7 project, and I suspect the most senior. I never met Tokubo (who had left the Lab by the time I arrived, as far as I know), but I did know all of the others (Anderson, Bertran, Conn, and Malmquist) fairly well. I overlapped just a bit with Millstein the first summer I was at Livermore, which was a summer student, not a full-time employee.

As a summer student, my immediate supervisor was Ken Bertran. I was given the task of writing code for the Sigma-2 that hosted the remote graphics displays. It is hard for me to remember the specifics of my summer work because the first few weeks I was not allowed into the classified area and thus only read manuals. So it was around July that I started working “inside”. Ken showed me a number of things that were not specific to the Sigma-7 or the Sigma-2 and I was more or less turned loose on the Sigma-2 to figure out how to write code (using paper tape only — no peripherals except for a teletype with paper tape reader and punch).

By some time in August, I had been reclassified from 2-S (student deferment) to 1-A (ready to be drafted) by the Selective Service System. I rather quickly applied for a full-time position and interviewed with a few groups within Computation. The interviews were a bit of a sham (as at least on interviewer admitted). It had already been determined I would be hired and also that I would be assigned to the Sigma-7 project. The interviews were just to make sure all the rules were followed.

The importance of the history of my summer is that by sometime in mid-August it was determined I would become a full-time employee. I had been schedueld to start graduate school at Stanford (in computer science) in September. I told Stanford I was not coming. I could have gone to graduate school part-time, but my NSF fellowship was only good if I was a full-time student and the 2-A (occuprational) deferment I was expecting because of Livermore reqjuired I be a full-time employee. Concurrent with that, I talked to Millstein (really for the first time) because someone (I think Dick Conn) told me that Millstein had gotten a master’s at Cal State Hayward (nearby) and knew about the new program at Berkely (a computer science department in the College of Letters & Science that had split off from the EECS department in the College of Engineering — so Berkeley had TWO computer science programs until about 1974 when CS merged back into EECS). All of this is to say that in August 1968, Millstein was stil at Livermore but he was in the process of leaving and he was no longer active in the Sigma-7 project after that.

Just to finish up the story about graduate school, Millstein suggested I contact Michael Harrison, a former EECS faculty member who was instrumental in starting the new CS department. I did and after a short meeting with Harrison at Berkeley, I arranged for letters of reference from Caltech. This was in August. Before the term started, I had been accepted. My guess is that my credentials were good enough, but also it was a new program and quite probably they did not have full enrollment because many people did not yet know there was a new program and some who had applied and been accepted no doubt took up offers elsewhere. I was lucky. Tuition was a lot cheaper than at Stanford and because University of California ran Livermore, I my tuition was paid by the lab (after the first quarter because I had not figured out all the angles yet). My first term I took a course from the Unabomber.

Kelly

Subject: Re: PACT?
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 7/18/25, 6:27 PM
To: ken rector <kdrhoo@yahoo.com>

> On Jul 18, 2025, at 12:43 PM, ken rector <kdrhoo@yahoo.com> wrote:
>
> I saw something that said the A-System included buffer memory for the Consoles.

Yes. This was for the four consoles local to the Sigma-7. The Sigma-2 (later a Sigma-3) maintained this information for the remote consoles but it passed through the A-system on its way to and from the Sigma-2. So the A-system had just a single buffer that was used to transfer to and from the Sigma-2 and each transfer was “atomic” in the sense that it was completed before the A-system returned to the B-system (and the B-system to the user code). See below for an explanation. The unit of storage was a “glyph” (I think limited to a maximum of 32 words) but each console could have multiple glyphs chained together within the display list that the graphics processor read in a continue loop as it refreshed the displays it controlled.

> Do you think the A-System had a memory pool for job/process and file system related stuff, like PACTs?

Yes. Each process had a subPACT that was I think 8 words. Jobs had something similar that might have been only 4 words — although it might just have been a special type of subPACT either formatted differently for jobs than for proccesses or literally the subPACT for process 0 or 1 (not sure what the number system was) for the job.

> Maybe thats obvious but I don't see it mentioned anywhere.

Well, obviously only in the sense that the information had to be stored some place and it had to be somewhere in the A-system. Each of these was a fixed size (number of buffers, number of subPACTs, number of job records. So the size of the entire A-system including all of its data was known at assembly time and did not change when the system was running. All of the remaining memory (all of the remaining pages) were a pool of buffers for disk pages. This size was also known at assembly time because it was simply 128K minus the size of the A-system memory (in pages).

I think I wrote about this a month or so ago in one of my longer emails.

The A-system used the low-core memory locations that were hardware-specified for interrupt and trap locations (each had a dedicated location, including for locations for clocks where each clock tick the instruction in the location was executed — it was presumed to be a MTW (or similar) instruction that would return a condition code so if the result were zero (or negative — not sure which), an associated interrupt would be triggered so the A-system (or whatever system was running) could service it — this is how processes were stopped after they had used up their allotted quanta of time, for example).

After that, starting at I think location x200 (i.e., page 1), the A-system had its executable code (instructions) and its static variables (more or less in the sense of C — these were the finite number of variables that maintained the state of the A-system. In additon, there were dynamic variables that maintained some (not all) of the state information for processes and the displays — only the bare minimum required by the A-system to work in real time.

The rest of the physical memory was used for swapping (starting of course on a page boundary). The A-system was I think completely agnostic about the content of those pages (i.e., it never looked at the content, it only

swapped the pages in and out of physical memory and it kept track (elsewhere) of the disk address of each page that was in memory, whether the page was dirty or not (dirty mean it had been modified since being read into memory to it had to be written back out to disk before the physical page could be overwritten with a different disk page). Keep in mind that (a) no disk page was evern in memory in more than one location, (b) the A-system had no real idea which processes might be using a disk page and it definitely did not know at any deep level what virtual page address the disk page had because a disk page could have a different virtual address in each process that used it and in fact a disk page could be coupled into more than one virtual page in the same process!

The static A-system variables that kept track of disk pages knew for each physical page in the swapping memory the disk address associated with that disk page (it was really the other way around — for each disk page that was currently swapped in the A-system knew the physical page. It also had a dirty bit for the page. And it had some pointers (possibly halfwords) to link the pages on the laundry list or the clean list and the list of pages that were being read in (i.e., for maintaining a couple of queues). The amount of memory of course depended on how large the swapping space was, but because most of physical memory was for swapping, just assuming there were 256 instances of each of these things (bits, halfwords, or whatever) is a good upper bound on how much static memory was required. Other static memory included software copies of the memory map (8 bits each for all 256 entries), the memory protection registers (2 bits each for all 256 entries), and the write-lock registers (need to look that up, but I think only 1 bit per page, meaning all 256 entries). So again, it is easy to compute how much memory these took and there were simply arrays declared (in machine language — so a BSS statement if I am remembering correctly)

I believe there were only two types of “dynamic” variables within the A-system. One was the subPACT (PACT was the Process ACtive Table, which was one page within the B-system address space for each process, where every process had its own disk page that had 256 entries that kept track of the disk pages that were coupled into the process’s virtual memory, 16 keyword entries that kept track of the open files, various other quantities the B-system needed including I think a push down stack that the B-system used (not the user code) so that all of the rest of the B-system virtual pages were either B -system code (shared by all processes) or were pages coupled by the B-system to help it do its job — so for example the B-system call of couple disk page I in file J into virtual page K would need to read in the header page for file J (using information in keyword J to determine that there was an open file in keyword J and the disk address of the header page). The B-system would couple the header page into its virtual space, then check to see if page I of the file existed — if not it allocated a new page that was guaranteed to be all zeros — and then assocate the address of that page to the process’s virtual page K (K would have to be a page from the user virtual pace, not in the B-system virtual space).

You can see that the B-system did all the work to maintain the virtual memory abstraction except for insuring the pages were actually in memory when needed and that the mapping and protection hardware were set properly. Tha A-system did that.

The subPACT was I think 8 words. It lived in the A-system and was one of the two types of dynamic memory. There needed to be enough of these so each process had a subPACT resident in memory. The main purpose of the subPact was keep track of the disk address for the process’s PACT so that when a process was run, the A-system could make sure that the PACT was in memory. Everything else the A-system did to manage swapping and memory protection followed from that. When a process on one of the ready queues was allowed to execute, the only memory page guaranteed to be in physical memory was the PACT. The first instruction would always cause a page fault, which allowed the A-system to fetch the necessary pages to fetch the instruction and the data for the instruction — if necessary swapping those pages in if they were not already in memory). The subPact had other information about the process (job number and process number) and pointers to maintain a few lists (the ready list, the sleep list, and the waiting for a page list) that were used to schedule processes. At assembly time the number of subPACT entries was set in a declaration. The number was arbitrary, but not particularly large (I

imagine perhaps 100?). There were also some dynamic memory keeping track of jobs, which would have been less.

The other dynamic memorty stored the graphics commands for the display. This was a fixed block of memory whose size was determined primarily by estimating a reasonable bound on the number of graphics commands tht the display processor could accommodate while maintaining a roughly 30 Hz refresh cycle (i.e. the number of commands that could be processed in 1/30 of second). The time for commands depended a lot on how long vectors were (or how much text and what size it was), so in the end whatever the block size was, it was an estimate. It could (and was changed) from time to time if the performance characteristics for GORDO were adjusted. The block of memory was a pool of buffers. I think originally they were fixed-size 32-word blocks, but at some point I implemented a buddy-system scheme that allowed 4-word to 32-word in powers of two because that made better use of the block. This was a rather minor concern. User programs would transfer up to 32 words of display commands at a time to the A-system by making a B-system call. From my memory, there were only two B-system calls: Insert a glyph (the name for set of the display commands) after an existing glyph (after ‘0’ meant at the front of the list of glyphs) or delete a glyph. Insert returned the A-system address of the glyph’s memory as an ID. Neither the user code nor the B-system code could use the ID as an actual address (they had no access to the physical memory), so the ID was really an abstraction. There was probably also a clear or delete-all B-system call. The B-system handled these calls by passing all of the paramters directly to the B-system using a block of words in the PACT for the process. So for an insert the user called the B-system with a one-word ID for the glyph after which a new glypy was to be inserted and the address and word count for the graphics commands that were in the user’s address space. The B-system copied the ID, the count, and the buffer contents to fixed locations in the PACt and then invoked the A-system with an insert request. In this way the A-system only needed to know where the PACT was in physical memory for it to extract the parameters and move the graphics commands into one of the glyph buffers in its memory space. There might have been a B-system call to read back a glyph give its ID, but I am not sure of that. The A-system was responsible for checking that the glyph ID was valid and that the process making the request owned the glyph (the B-system would not have enough information to do that).

The A-system also had a small amount of memory for keyboard buffers, lightpen hits, and function box status that it maintained by servicing interrupts from the display processor. When this information changed, the appropriate process was awakened (if it was sleeping) and then the process could (if it chose to) issue a B-sytstem call that would in turn ask the A-system for this information.

Kelly

Subject: Re: PACT?
From: Kellogg Booth <ksbooth@cs.ubc.ca>
Date: 7/18/25, 11:35 PM
To: ken rector <kdrhoo@yahoo.com>

There is a bit more to say about how the A-system kept track of disk pages as it swapped them in and out of memory.

> On Jul 18, 2025, at 6:27 PM, Kellogg Booth <ksbooth@cs.ubc.ca> wrote:
>
> The static A-system variables that kept track of disk pages knew for each physical page in the swapping memory the disk address associated with that disk page (it was really the other way around — for each disk page that was currently swapped in the A-system knew the physical page. It also had a dirty bit for the page. And it had some pointers (possibly halfwords) to link the pages on the laundry list or the clean list and the list of pages that were being read in (i.e., for maintaining a couple of queues).

The first additional bit of explanation is that the above is not the full picture. One might imagine that for each physical memory page the A-system would have a struct (not stored in the physical memory page because that memory had the content of some disk page) and the struct would have a field that specified the disk address that was currently occupying that physical page of memory (zero of -1 or some other special value indicating that there was current no disk page occupying that memory — or possibly a separate one-bit field with yes/no, but I think it is almost certainly a special value and quite probably -1). Other fields in the struct would be a one-bit “dirty” flag to indicate that the copy in memory had been modified and might not match what was on the disk, plus I think two pointers (forward and back) that were used to link the struct into one of a couple of mutually exclusive lists of memory pages. The lists would be a list of memory pages not current in use (no disk page was stored there), a list of memory pages that were waiting for a disk page to be read in (so they were allocated but not yet usable), a list of memory pages that were being written out to disk because they were dirty (the laundry list), and I would assume a fourth list that was clean memory pages that were available to use if some process needed that disk page.

My guess is that there was not just a dirty bit, there was probably a two-bit field that specified which of the four queues the memory page was on, so that being on the laundry list meant a page was dirty and it was waiting to be cleaned (written out to disk).

Initially all memory pages would be on the unused list. They contained no disk page image (i.e. the data in them was meaningless).

f page fault took place and the desired disk page was not in memory already, one of the unused pages would be removed from the list and put onto the list of memory pages waiting to have disk pages read into them. The exact sequence of operations would be something like the struct for page would be removed from the unused list, then the disk address that was needed would be put into the struct and the field indicating the queue would be changed the input queue and then the struct would be linked into the input queue. The unlinking and linking steps (just a few instructions) would presumably be done with I/O interrupts disabled to ensure that the disk I/O interrupt handler was not simultaneously manipulating the lists. All of this would be an atomic action wrt to user processes but I/O interrupts had higher priority than memory protection traps.

Side note: There would not need to be a list of empty pages because initially empty pages could all be on the clean list but with -1 for the disk page address and if a physical memory page was needed the first one in the list (which would be “oldest”) would be used. See what follows for the semantics of a clean page being over-written.

Once the page needed to fix the page fault was in the input queue, the A-system would put the process that had caused the page fault in a list of blocked processes (the subPACT for the process would be put on the list) and the disk page struct would point to the subPact so that when the page eventually came into memory the A-system would know to move the subPact off the blocked list and onto the ready list for processes that were no longer waiting for pages. If while the page was in the input queue another process page faulted on that same page, presumably the subPACT for the second process would be “pushed” onto the list ahead of the other subPACT (order would not matter because when the page came into memory all of the processes waiting for it would move to the ready list and that could be by putting them one at a time at the end of the ready list but each successive one just in from other previous one (i.e., reverse the order of the subPACTS as they were being appended the ready list).

The disk I/O manager could schedule pages in using whatever strategy it wanted. I am not sure if it did more than just first-in first-out, but in principle it could keep track of where the disk was and schedule pages that were coming up. I think this is what was done and the mechanism was that the input list (pages to be read in) was actually mutiple lists, one for each double-sector (page) boundary. In an earlier message I said there were 6 pages per track, which would mean six input queues, so not much extra storage overhead (six pointers instead of one — and if these were indices into an array they would only need to be one byte because there were less than 256 physical pages for swapping.

The laundry list would be similar, but for pages going out to the disk. While a page was on the input list, no process would be running that used it. Any process that tried would page fault and the second and subsequent page faults for the same page would not affect the page, they would only result in more processes being blocked on that page. When a page was sheduled for reading (i.e., when the I/O was initiated) the page would still be on the input list but the disk manager would know that it was the page being read, so when the I/O was complete the page would be marked as in use and clean and put onto that list, and then all of the processes that were blocked on the page (whether they wanted to read, execute, or write to it did not matter) would be put on the ready list. For pages on the laundry list, they two would remain in use while they were waiting to be written out to disk, but if they got a page fault and needed write permission, they would be moved to the end of the laundry list (and thus be marked as dirty). Similarly, if a page was being written to disk and a page fault happened that required write permission, the page would also be moved to the rear of the laundry list and a note would be made to essentially ignore the completion of the I/O. Otherwise, when the I/O was complete the page would be moved to the clean list. One important step in this is that when the I/O to write the dirty page to disk was initiated, this would be with the A-system running so no user process was executing (whatever had been running was interrupted). The disk I/O interrupt handler would figure out what page to write and after initiating the write to disk it would turn write access in the memory protection registers (which were what determined the access a user process had to a virtual page) but leave the read and execute permissions as they were (on or off). This would have to be done for ALL of the virtual pages in the suspended process that that were coupled to the disk page (there could be more than one!). I am not sure how this was done efficiently, but one way to do it would be to the struct for each disk page include a pointer to a linked list of virtual page numbers to which it was coupled in the current executing (or suspended) process. These would be one-byte pointers and one-byte values, so halfwords and there would only need to be 256 of them. The alternative would be a linear search through all just-about-256 possible virtual pages, which would be rather wasteful.

OK. I think this is almost everything. Initially, all physical pages in the swapping space are on the list of pages that are in use and clear but with disk addresses of -1 so the A-system knows there is nothing useful in them. As page faults happen, the pages are one at a time filled up with disk pages (never the same disk page in two different physical pages) and put on the clean list and if processes need them for read or execute they stay there but if a process needs them for write they get moved to the laundry list where they eventually are written out to disk and returned to the clean list with care being taken that if they are dirtied before the I/O completes they move to the

laundry list instead so they will be written back to disk again. The one case not covered is when a new page has to be read in from disk and the page at the front of the clean list is in use by the current process (which is temporarily suspect while the A-system is doing its thing). In that case, the A-system turns off all of the access permissions for that disk page in all of the virtual pages for the current process (much like it did when a page was laundered but in this situation read and execute also not allowed). The physical page then gets marked as not having useful data but is put on the input queue so when it does have data processes can use it but before that any process that needs that page will be blocked (as was the process that first caused it to be put on the input queue). Well, actually there is another case as well. The clean list is empty, so all pages are on the laundry list. In that case the A-system blocks the process and waits for a clean page so it can overwrite that with the desired page. This requires a bit of fiddling, essentially making sure that the page does get laundered but no process is allowed to dirty it because it is scheduled for over-writing. This could require yet another status bit in the struct for physical pages because there could be more than one page being scheduled for over-writing, although perhaps this just means there is a third I/O queue for over-writing.

The I/O queues would presumably be overwriting first (because at least one process is waiting not just for a read from the disk, but also a write to clear the physical page and then the read). After that would be the read queue (because at least on process is blocked on that), and last would be the write queue because no processes are blocked on that. When the system shuts down, all processes are stopped (or blocked if the system is to be started up again) but the disk manager continues to run until the laundry list is empty — at which point all pages on disk are synchronized with what was in physical memory.

Note: Because the Sigma-7 use indexes into arrays different for bytes, halfwords, words, and doublewords, C-like structs are often not the best approach if there is an array of structs (as there would be here). Instead, each field would be a parallel array of the appropriate size. Machine instructions would be something like

    loadword wordfield,index — for a 32-bit field

but

    loadbyte bytefield,index — for an 8-bit field

etc.

One final point. When a page fault happens it is caused by a virtual address in a user process accessing a memory location for which the memory protection registers prohibit the operation (read, execute,write) needed by the instruction. This cause a trap and the A-system determines the virtual memory location that was requested and the level off access requested. It then looks in the process's PACT (which is a disk page for the running process) and within that it looks at the PMT (I think this was the name — process mapping table) for the disk address associated with the virtual page for the virtual memory location (i.e., the top 8 bits of the virtual address). The A-system always guarantees that the PACT for the running process is in memory (the process scheduler insures this is always the case). So once the virtual address is known, it is just a couple of instructoins to extract the virtual page number and use it to index into the PMT to get the disk address for the desired page). The A-system then checks (as we discussed above) to see if the page is in memory and if it is it adjusts the memory protection registers (and the memory mapping registers too if necessary) and restarts the process — after of course checking that the process is allowed this level of access — if it is not, the process aborts). If it is not in memory, the A-system arranges for it to be brought in, again as discused above.

The one detail we have not described is how the A-system figures out if the disk page is already in memory some place. It could of course just search through all of the almost-256 physical pages to see if one of them has the desired disk page, but that will be very slow. Instead, there is an "associative lookup" that essentially provides on

O(1) time for this step. As you may have noticed in the ACM conference pager, Peter Deutsch is thanked for suggestion the association table. The actual scheme (there are many ways to do this) was a hash table. It might have been closed hashing, which means there is an array with all the entries and a bunch of zeros where are not entries. The disk address being searched for is “hashed” (just taking the low bits is probably good for this particular application, but I think there was a slightly more sophisticated had function in actual practice). I forget the exact size of the hash table, but it was a bit less than twice the number of pages that would fit into memory. A hash table with double the number of entries means that almost certainly the average number of probes is at most two per look up. Recall that with closed hashing, the hash code is used as an index into the hash table and the hash key at that index is compared to what you are for (both are disk addresses) and if it does not match you go to the next location in the table and try again, eventually finding the entry if it is there or finding a zero entry if it is not there. Once the entry is found, the other information associated with the hash key is retrieved (this is the struct information) and if it is not found the zero entry is replaced with a new entry. Removing entries is tricky. So I think instead it was open hashing, where each entry in the hashtable is the head of a linked list (a bucket) and the idea is that if the hash function is reasonable there will be about the same number of entries in each bucket. In our case, for paging, this means each of the structs has another field that is the pointer for the hash chains (the linked list of disk addresses that hash to the same bucket). This can be singly-linked because we do not jump into the middle of a hash chain so we alway are able to add an element or delete an element knowing only forward pointers.

OK. That’s pretty much everything about how paging was managed.

Not described was process scheduling. Some aspects related to page faults were described, but not the details of priorities amongst processes, and things like wakeup and sleep calls, and process creation and destruction that interact a bit with how paging works, and the mechanics of coupling a page when no disk page is currently assigned (so the B-system needs to allocate an unused disk page to that page in the file before the process can be restarted with the PMT updated after which the page fault mechanism does the rest).

And one other really big detail was glossed over: How the A-system determines the virtual address that caused a page fault and the access level that was being requested to that page. It should be simple, but it was not. SDS did not provide this information in the hardware. All the trap handler for memory faults got was the PSD (program status doubleword) for the user process. Included in the PSD was the virtual address of the instruction that was being executed. That was easy to retrieve. From that, the A-system knows if the process was in user mode of B-system mode (the program counter was either in low memory or high memory with the boundary well defined as part of the GORDO design). But that’s all you got. So the A-system actually has to essentially simulate the hardware in software. And SDS did not allow the mapping registers or the memory protection registers to be read back. So the first step is the the A-system maintains at all times a software copy of both. The decoding steps to find the virtual address that caused the problem are as follows:

Step 1: Retrieve the PSD and get the virtual address for the instruction that was being executed. This is trivial. It’s one instruction to load the relavant work in the PSD and then a mask (and operation) to just keep the program counter and a shift to get just the high eight bits, which is the virtual page number (all you really wanted because page faults are determined by the virtual page).

Step 2: Retrieve the actual instruction that was being executed. Sounds siimple. Just use the virtual page number to look into the software copy of the memory mapping registers to get the physical page number and then use the low-order bits of the address to index into that page to get the instructions word. Works fine if the page is in memory, but the page might not get be in memory! So you have to look at the software copy of the permission registers to see if execute access is allowed for the page. If it is not, the page fault was caused by the instruction fetch and we are done figuring out what caused the page fault.

Step 3: If execute access is allowed, then we can use the software copy of the memory mapping registers to get the actual instruction. We then need to decode the instruction to see what type of data access it was doing. For simple cases that is not too hard. Pull out the address field from the instruction and then repeat the process in Step 2 to figure out what access is allowed for that address and if no access is allowed it means that was the cause of the page fault so we are done figuring out the cause, but otherwise we know the page is in memory but we need to check to see if write access was requested and if it was whether that is allowed. If it is not allowed, the process is aborted and there is no page fault and similarly only read access was required but it not allowed the process is also aborted. But it is not this simply because the instruction might involving indexing, in which case we have to first retrieve the contents of what register is specifed for indexing and add that (with an appropriate shift depending on whether the instruction wants bytes, halfword, words, or doublewords) to get the actual virtual address of the data and then we do the process just described. But wait, there could be indirect addressing. In that case we might find there is no page fault but we then need to use the software memory mapping registers to figure out the physical page where the data is and then pick it up and use it as the virtual address (and I think we do that first, before indexing, because indexing happens after indirection). What I cannot remember is whether multiple levels of indirection were allowed. Maybe. But if not, we still have some at least as bad still to come!

Aside: Byte string instructions, load/store multiple, and the stack push/pull and push/pull multiple instructions all involve more than just one memory access and there are more than one address that could cause the page fault. Some of this is simplied becuase the byte string and multiple instructions are “interruptible”, which means if they cross a page bound and a page fault takes place they stop and a trap is triggered but when the instructions restart after the page fault has been fixed the instructions continue where they left off. What this means is that we just have to figure out if the current address(es) for the step on the instruction that trapped caused the page fault. It is fairly straightforward, but each piece typically involves a bit of extraction to get an address that is checked for access for each of possibly two data addresses. And then there is the execute instruction, whose effective address (operand address) is where another instruction is that is what was supposed to be executed. So we repeat the process starting at Step 2 and possibly iterate multiple times if a sequence of execute instruction is encountered until we finally get to a real instruction and figure out whether the page fault needs to abort the process (because it tried to access something it was not allowed to) or we simply need to bring the appropriate disk page into memory and start up the process again so it can get a bit farther along until the next access problem.

Bad news: If there is a circular chain of execute instructions, the decoding never ends and the A-system is running around chasing an endless loop looking for a page fault that will never happen. Of course this means that there was no access violation, so one wonders why there was a page fault. Well, the answer is that the execute chain was not in the original code, but it is what we see as the result of (for example) a byte string instruction that moved a bunch of bytes that overwrote the instruction itself with an execute instruction that happens to start a loop and then the byte string instruction got caught crossing a page boundary and encountered an access violation so the hardware traps to the A-system and the A-system is stuck trying to decode garbage because the original instructions is long gone and what is left has the execution loop. The load/store multiple and the push/pull multiple instructions can also cause this. Oops! So the software decoding had to add a counter so that after some arbitrary number of instructions (I think we decided 1000) the A-system gives up and simply aborts the process with an abort code that indicates there was no cause that could be found so it must have been something like this that happened.

That’s all for now.

Kelly