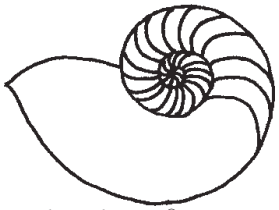
Volume 4, Number 7 May 1992 $2.50



TUUG Lines

Technical UNIX® User Group

Newsletter of the Technical UNIX® User Group

User Group Survey Results

By Roland Schneider

Thank-you for the excellent response to the survey. We got 29 surveys back, that's almost 50% – better than voter turnout for American elections! I'm just going to summarize some of the more interesting results here – if you want a complete breakdown of the results please feel free to approach any member of the executive.

All about us

The most common applications are software development (59%), word processing (55%), database (52%) and electronic messaging (45%). (Totals can add to more than 100% because more than one reply was allowed.) Word perfect and Lotus 1-2-3 were the most popular commercial software packages, although "Other" was the true winner – I guess there's a lot os specialized software in use. About half of us named programming as our primary job function, followed by support and consulting.

What we want

The most popular topic for presentations was networks and communications (69%), closely followed by UUCP, system administration, database applications, X-windows, and system configuration (all over 50%). Most of the other topics were also very popular. Most of us seem to be interested in most aspects of UNIX, from technical details to high-level applications.

Most of the members who replied favoured presentations, demonstrations, and discussions, as well as site visits and workshops to breakfast, lunch, or dinner meetings.

As far as other services which could be provided by the group, members are most interested in public domain software exchange (62%), newsletters and symposiums (59%), e-mail, bulletin boards, and day seminars. In other words, everything.

What's next?

Your executive will use the survey results to guide future directions and initiatives, but if you *really* want to have a say, come out to an executive meeting. Your ideas are always welcome, especially if they are backed up by a commitment of the time and energy required to implement them.

THIS MONTH'S MEETING

Meeting Location:

This month, the meeting is to be held at the U of M again – room 234B in the new Engineering Building (near the Senate Chambers). If you've found the Engineering Building in the past, you'll have no problem. If not, it's just south of University Centre. The meeting is set to start at the usual time – 7:30 PM, on May 12, 1992.

Meeting Agenda:

See last page for details.

INSIDE THIS ISSUE

Newsletter Editor's Ramblings
President's Corner
Hardware: Static Electricity
Industry: Incompatibility and Software Portability
Hands-on: Shared Memory for Inter-Process Communication
Technology: Digital's Alpha Architecture
April 14th Meeting Minutes
May 12th Meeting Agenda

UNIX is a registered trademark of UNIX System Laboratories, Inc.

The Machine That's Still Changing the World

By Gilbert Detillieux

An interesting five-part series currently playing on PBS is entitled *The Machine That Changed the World*. It started off looking at early mechanical, electro-mechanical, and electronic computers, and moved on to today's technology, it's impact on society, and speculation about what the future will bring. As I sit and work on this newsletter, I realise the extent to which it's affected my life. (I don't really want to think about whether it's been for the better or for worse right now. :-)

Certainly, this newsletter and the group itself would not be possible without a lot of the technological changes that have come about in the last few decades. I'm certainly thankful for the technology that permits me to accomplish the work I do, usually with relative ease. It's also interesting to speculate about how the job could be made much easier with new and future technology in place. But despite all these marvels of the modern world, behind it is still a group of people dedicated to a cause they believe in.

Our newsletter wouldn't be the success it is without the efforts of a lot of people in the group who help me out regularly. I'd like to thank our regular contributors, Susan Zuk, Roland Schneider, Scott Balneaves, and Peter Graham, as well as other occasional contributors, Allan Moulding and Richard Kwiatkowski. I'd also like to thank Pat Bessler, who typed in this month's article from UniForum Monthly, and has offered to help with future articles too.

On the subject of machines that change the world, and of people dedicated to a cause, there's an interesting organization in the city that has been brought to my attention, called *Computers for Charities*. They maintain a database of charities that need either computer equipment or computer expertise, and match them up with people who can provide assistance. If you have some time to spare, or have some equipment that you're about to retire, consider helping change the world for some worthy cause. For more information, contact Dennis Bayomi at 788-6725.

The 1991-1992 Executive

President:	Susan Zuk	(W) 788-7312
Past President:	Eric Carsted	1-883-2570
Vice-President:	Richard Kwiatkowski	589-4857
Treasurer:	Rick Horocholyn	(W) 474-4533
Secretary:	Roland Schneider	1-482-5173
Membership Sec.:	Allan Moulding	269-8054
Mailing List:	Gilles Detillieux	489-7016
Meeting Coordinator:	Kathy Norman	474-8311
Newsletter editor:	Gilbert Detillieux	489-7016
Information:	Susan Zuk	(W) 788-7312
		(FAX) 788-7450
(or)	Gilbert Detillieux	(H) 489-7016
		(FAX) 269-9178

Copyright Policy and Disclaimer

This newsletter is ©opyrighted by the Technical UNIX User Group. Articles may be reprinted without permission, for non-profit use, as long as the article is reprinted in its entirety and both the original author and the Technical UNIX User Group are given credit.

The Technical UNIX User Group, the editor, and contributors of this newsletter do not assume any liability for any damages that may occur as a result of information published in this newsletter.

Our Address

Technical UNIX User Group
P.O. Box 130
Saint-Boniface, Manitoba
R2H 3B4

Internet E-mail:
tuug@cs.umanitoba.ca

Group Information

The Technical UNIX User Group meets at 7:30 PM the second Tuesday of every month, except July and August. The newsletter is mailed to all paid up members one week prior to the meeting. Membership dues are $20 annually and are due at the October meeting. Membership dues are accepted by mail and dues for new members will be pro-rated accordingly.

PRESIDENT'S CORNER

What's New with TUUG (or is it MUUG???)

By Susan Zuk, President

Well, what has the UNIX Group been up to in the last month? We had an excellent executive meeting last week and discussed a number of exciting topics.

Our involvement at the Muddy Waters Computer Fest will help us to promote our group and UNIX. This will be one of our first promotions of the group to the Manitoba community. We expect the group will continue to grow with having a booth at this show.

UniForum affiliation is a very exciting step for our group. We will now be able to work with other Canadian UNIX groups to share ideas and information. I am looking forward to meeting representatives from these other groups. UniForum holds its yearly UNIX Show every spring. At this event they also bring together the Presidents of all the affiliates. This group is called the National Council. We will be meeting at the end of May to discuss local and national opportunities.

Locally we will be promoting UniForum Canada and will be encouraging individual membership. Please stay tuned for more information. The group has been very busy trying to coordinate and set policy as to how to integrate UniForum memberships with our own. If you really want to join UniForum immediately please call me and I will provide you with this information.

My final note is on the donation of a Sun workstation from the University of Manitoba. We really appreciate the University's commitment to and support for the group. TUUG wishes to thank those that made this possible, particularly Kathy Norman and Bill Reid.

We will be using the workstation to provide Internet service for members. This service will start in the next month or so. The service will be free until October if you are a member. Please see the enclosed account application form. We will probably be holding a tutorial to show you how to use and set your system up to connect to the workstation.

In the next while you will be seeing the group promoted as both the Technical UNIX User Group as well as the Manitoba UNIX User Group. We will still be known officially as TUUG until the name search on the new name has been completed, although we are already promoting ourselves under the new name.

Finally, I would like to thank Amdahl for their presentation last month on their flavour of UNIX called UTS.

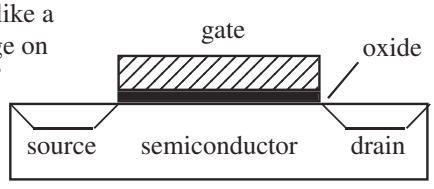
ROLAND'S HARDWARE CORNER

Static Electricity and Your Computer

When you buy a board for your computer, it comes in a pink, green, or grey plastic bag, usually with dire warnings printed on it about avoiding static electricity. The bag itself is conductive, so that any static will be drained away safely, and you won't have any problems if you're careful. Why are modern computer components so sensitive to static electricity?

Most of the components in computers consist of NMOS or CMOS integrated circuits. In either case, the MOS part stands for Metal Oxide Semiconductor, the three layers making up an integrated transistor. The metal layer (actually polysilicon, nowadays) is on top, the semiconductor (silicon) on the bottom, and the oxide (silicon dioxide) is the insulating layer between them.

The transistor is like a switch, and the voltage on the polysilicon "gate" controls whether current can flow from the semiconductor "drain" on one side of the gate to the "source" on the other. Because the gate is insulated from the semiconductor by the oxide, which is more or less glass, no current flows from the gate.



An Integrated MOS Transistor
(side view)

This "glass" is only 0.025µm thick, and a static jolt can have a voltage of 1000V, so the electric field is around forty **billion** volts per metre — much worse than a lightening bolt hitting window glass. What happens is that an ionized path (an electrical hole) is punched through the oxide, and the gate is no longer insulated from the semiconductor underneath.

Chip manufacturers use special circuitry to protect the sensitive internal components of integrated circuits by draining the static charge away before it gets to any gates. While this protection is effective, there is a limit to the amount of charge which can be handled. Here are a few simple rules to follow when handling static-sensitive chips and boards:

1) Don't remove a new board from it's protective envelope until you're ready to use it.
2) Touch something metal, like the case of the computer, before you touch anything inside the computer.
3) Don't touch the pins of chips or the edge connectors on a board. Handle everything by its edges.
4) Avoid opening your computer on cold, dry, winter days, and beware of carpets and plastic-soled shoes.
5) If you do a lot of fooling with hardware, you might consider acquiring an anti-static table mat and a grounded wrist-strap.

Roland Schneider *is currently the TUUG Secretary.*

Across the Chasm of Incompatibility

To deliver on the potential of software that runs on multiple platforms, vendors strive to port applications.

By Stephen Lawton

Reprinted with permission from the March 1992 issue of UniForum Monthly, published by UniForum.

As UNIX-based options show alternate routes for today's computing, users are finding some impasses on the road to open systems. One of the widest is the portability of application programs. Depending on how code originally was written, the task to port a program from one version of UNIX to another can range from straightforward to complex. And porting from MS-DOS or another single-user operating system to UNIX means bridging a wide gulf.

Ken Hobday, software engineering manager in the CASE group at DEC in Nashua, NH, defines portability as "the ability to move applications from one system to another at a reasonable cost." He emphasizes that no platform has 100 percent portable software – not even DOS, which many users site as a paragon of portability. In the DOS world, developers must be concerned, for example, with the type of monitor being used, such as EGA, VGA or Super VGA, and whether the program is run from a command line or under the graphical environment Windows. They also must consider which version of DOS they are using.

Users, Hobday asserts, are concerned most about *interoperability*, which includes passing data from programs running on one platform to other applications running on dissimilar platforms. Although it is important to leverage their previous investments in training when it comes to learning new programs on different platforms, users want even more to be able to work with data they already have. A ported database, for example, must be able to access the existing database files, as well as files on other systems. In such a case, it might be necessary to transfer statistical data from a UNIX workstation in manufacturing to a DOS PC in marketing and a proprietary minicomputer in accounting.

Systems manufacturers and independent software vendors (ISVs) must cooperate to build bridges for true portability. Unfortunately, they often start from opposite ends of the project.

Where The Work Is

Relational database vendor Informix Software of Menlo Park, CA, offers its products on a variety of UNIX platforms, as well as DOS and OS/2. The company develops software on Sun workstations, then tests it on what Gilbert Wai, product marketing vice president, calls "acceptance platforms": workstations from Sun and Hewlett-Packard and a third system that is chosen depending on the target UNIX version and hardware.

Software designed on a UNIX platform for UNIX systems is relatively easy to port, Wai says. Although UNIX versions treat some programming components differently – such as the location of the most and least significant bits in a byte – the amount of code rewriting during a port is small.

The bulk of the ISV's effort comes in testing the revised software. "Ninety percent of the time required to complete the port is testing and validation," Wai says. Paul Wensley, engineering vice president for Island Graphics Corp., a word processing and graphics software vendor in San Rafael, CA, agrees. He says that a working version of a port based on the Motif or Open Look graphical user interface (GUI) can be ready within days but testing the program takes a great amount of time.

The next most demanding task in porting is rewriting documentation for the new platform. "The operating system documentation doesn't always tell you what you need to know," says Jack Gold, marketing vice president for imaging software vendor PCS Systems of Northborough, MA. Other factors in the effort to recast documentation are largely logistical: making sure that all the features in the original documentation are listed in the new documentation and that they are described in a way that will make sense to users of the new target platform. Editing and proofreading the text, and printing it, also take time. Some vendors are choosing to publish documentation on CD-ROM so all such work can be done on line without paper.

Nuts And Bolts

According to Hobday, DEC generally approves porting projects that take up to one man-year (say, two engineers working for six months) to complete. Many ISVs, however, cannot afford to expend so much effort; often they measure the time for porting, debugging and verification in weeks rather than months.

Hobday estimates that some 80 percent of the code should require no changes in a port; only 15 to 20 percent of the machine-dependent code – the user interface, operating system interface and a small amount of other machine-specific code – must be modified. Even so, he says, "The finished product needs to be able to exploit the capabilities of the new system."

Wensley recommends that programs have an "auto-detect feature" that can determine the types of host platform on which they run. Island's auto-detect mechanism uses a standard X Library call that requests vendor-specific information such as the vendor ID, the version number and what company built the hardware. Using this information, the program identifies the host platform and optimizes itself for it, avoiding potential bottlenecks associated with some UNIX versions.

Wensley cites three key issues in deciding whether to undertake a port. Two are traditional marketing and sales considerations that apply to any new product: How many systems using that particular version of UNIX are in the field and how many units must the vendor sell to recover its investment?

The other consideration is technical and requires that the ISV understand the target platform's specifications. Among

the issues here are CPU speed, amount of system RAM, number of colors the graphics card and monitor can display, monitor resolution, types of input devices (such as mice) to be used and number of frame buffers. Once these issues are resolved, he says, doing the actual work on the port is straightforward.

Building Graphically

Generally speaking, application software includes, as well as the program itself, a user interface and an operating system interface. In DOS code it is common for user interface-dependent commands – whether character-based or graphical – to be woven throughout a program. Much of this user interface code describes how information is to be displayed. Being dispersed, it is not easily isolated and modified.

For UNIX GUIs, most of that information is contained in Motif or Open Look modules. Software developers can write code that is portable across versions of UNIX by keeping much of the GUI code in a discrete module. As the GUI is upgraded over the years, only that module has to be altered. Both Motif and Open Look are based on the X Window System, which was developed at the Massachusetts Institute of Technology in Cambridge, MA, to allow data on one compliant system to be displayed on others.

But various platforms have different performance characteristics. Because X provides the ability to show data on a platform that is different from the system on which the data was created, X itself must be generic. It is not optimized to take advantage of a particular platform's benefits or overcome its performance handicaps.

Crossing The Canyon

The vast majority of UNIX code is written in the C language, which was designed to be highly portable. A key component of moving a program from one UNIX version to another is *tuning*, the act of data collection, analysis and implementation to improve performance. It is significant because each platform looks at code a bit differently. Silicon Graphics computers, for example, expect a file format that is different from that of other hardware systems. File filters must be utilized so the operating system can recognize the data.

Similarly, window management differs in Open Look and Motif. Although Island Graphics uses standard UNIX tools, such as *make* and shell scripts, to port its programs, programmers still must write some machine-specific code. Both Island and PCS Systems base their applications on Motif and rewrite from scratch the presentation portion of a program when porting to a non-Motif-based UNIX.

PCS also makes use of callable routines in the operating system to reduce the amount of rewriting for revisions. When companies introduce new versions of an operating system, they generally provide backward compatibility for such callable routines. Using the operating system routines assures ISVs that their program will be compatible with later versions of the operating system.

DOS Plus And Minus

UNIX-based hardware vendors should take their lead from PC makers, Wensley advises. In the PC market, software suppliers know that all the hardware works alike. Hardware vendors add value by offering faster processing or optional boards that deliver additional features. In the UNIX world, one computer running UNIX System V with Motif might not be code-compatible with another because, although the operating system is the same, the CPUs come from different manufacturers, which necessitates recompiling the program. (For more on this issue, see "ABIs In Theory And Practice.")

Yet DOS systems tend to avoid some issues that UNIX developers must tackle. In particular, *scalability* becomes important when porting software from the DOS world to UNIX. Most PC-based software is designed for single-user platforms. UNIX-based workstations and servers, as well as midrange and mainframe systems, are designed for multiuser, multitasking environments. Porting software from one multiuser platform to another generally is easier than porting from a single-user platform because the original code already addresses multitasking issues. PC software may require extensive rewriting for use in a multiuser environment. For that reason, many companies port only local-area network versions of PC software to UNIX.

In addition to the single- and multiuser issue, Hobday says developers should anticipate what else their users will expect of the software. Do they want concurrency when executing commands? How large a data file do they expect to manipulate? The answers to these questions more clearly define if, as well as how, a port should be made.

"Some UNIX databases work exceptionally well when the directories are in main memory," he says. Performance drops off considerably when the database directory is not stored in memory. Similarly, small databases may perform well when they store tens of megabytes of data. When the user has hundreds of megabytes of files, those databases become too slow. What's more, users may have different expectations of levels of support. A developer must try to predict how much demand for technical support a product will incur on a new platform.

Porting Specialists

Some ISVs, rather than porting software themselves, license their applications to other companies that do the port and sell the software in the new market. Two such companies that port to UNIX environments are UniPress Software of Edison, NJ, and Hunter Systems of Palo Alto, CA. They take substantially different approaches.

Hunter Systems' strategy is to license DOS software, port it to run on UNIX and act as a reseller. Portable software is a compromise, says Colin Hunter, president and founder of Hunter Systems, between software that can really be created rapidly and software that can be moved easily between platforms. Today, according to him, most software developers write for the lowest common denominator in C to make their code portable. By doing so, they may not take full advantage of any platform. "The result is a cross-section rather than a program with a good look for a target market," he says.

Hunter Systems' XDOS porting tools work at the binary level with the hardware interface and device driver interface. The toolkit takes in code, decompiles it and creates an

intermediate representation, which then is compiled and optimized for the new platform. A run-time library handles the operating system-specific calls.

XDOS creates an application programming interface (API) that is consistent across UNIX platforms, the company claims. By being so, an application can run on various systems that have an XDOS Transformer, which links system-specific code to the API.

UniPress has a different strategy. It sells the XView Toolkit for writing X applications that run on Sun systems to ISVs who develop and sell their own programs. The toolkit converts SunView or XView applications to run on UNIX systems from DEC, Hewlett-Packard, IBM, Silicon Graphics and Sony.

Mark Krieger, UniPress president, advocates programming in C++, which creates object-oriented code. In conventional C, it is necessary to know how bits are defined in a byte and bytes defined in a word, bit masking and other technical issues. Therefore, code written in C is less portable than C++ code if the user does not have considerable knowledge of the target version of UNIX and its idiosyncrasies, Krieger says.

Even the UNIX *make* utility is not necessarily standard, he says. Using different *makes* under different versions of UNIX might not create the same result. Some machine-specific "bells and whistles" might make the code not directly portable.

UniPress' method is first to take the software in its original iteration and run it against standardized tests whose results provide a performance base against which the ported software is tested. The next step is running the program through XView Toolkit and creating new code. The new code is compiled, after which an engineer inspects it and fixes any errors or warnings. Then the code is tested again and new results are compared against the originals.

Building Better Bridges

ISVs generally agree that porting software from one version of UNIX to another is getting easier. There is general consensus that C++ is a stronger programming language for portability than C. And despite the time and expense, porting is still the only way today for developers to be on all UNIX systems.

Rewriting for each platform simply is not realistic, from a price or time-to-market perspective. So ISVs face a difficult decision: port their software themselves or license it to a third party, which then will be responsible for the porting.

Users remain at the mercy of such decisions. Whether a program will be available for many versions of UNIX depends on the developer's strategic plans and resources. Either way, the user generally ends up paying for the cost of the port in the form of higher purchase or maintenance prices. Users who demand software on multiple platforms and the most popular versions of UNIX must expect to pay for it.

Although developers are unwilling to reveal exactly how much it costs to port a program, we can specify some of the basic costs: approximately six months of engineering time for code rewriting and quality assurance; at least two months for documentation modifications and printing; plus marketing and administrative overhead. The accumulated cost can translate into hundreds or even thousands of dollars for each customer, based on the platform and number of copies sold.

Nevertheless, ISVs promise more programs ported to different versions of UNIX. Despite the costs, many of them see porting as a part of their commitment to customer service – as long as they can still make sufficient profit.

***Stephen Lawton**, a free-lance writer based in San Bruno, CA, has covered the computer industry for more than 13 years.*

Special Files

How does the UNIX kernel know that the file named "`/dev/ttya`" refers to a serial port, and that "`/dev/rst0`" is a tape drive and not a plain disk file? Unlike MS-DOS, it has nothing to do with the name. These are *special files*, created with the `mknod` utility. Instead of storing data, they have a flag indicating they're character (e.g. serial port) or block (e.g. disk) special files, and major & minor device numbers.

The major device number is an index into an array of structures of function pointers compiled into the UNIX kernel. The kernel calls the appropriate function for the device in question in response to a system call. For example, the `/dev/rst0` (tape drive) major device number is 18 on my computer. (You can find the major and minor device numbers with `ls -l`) Entry 18 in the kernel table `cdevsw` contains points to a structure containing the functions `stopen()`, `stclose()`, `stread()`, `stwrite()`, and `stioctl()`, which control a SCSI tape drive.

But what if there is more than one tape drive of the same type? That's what the minor device number is for. `/dev/rst0` has minor device number 0, `rst1` has minor device number 1, and so on. The interpretation of the minor device number is left to the device driver functions, which have to decide which physical device is being referred to.

The minor device numbers are good for more than selecting the appropriate device. For example, although `/dev/rst0` and `/dev/nrst0` refer to the same device, `nrst0` has minor device number 4, (bit #2 is set) which indicates to the driver that the tape should not be rewound when it is closed. Other bits may indicate what tape density to use, etc.

For serial ports, on Sun systems anyway, setting bit 7 (value 128) of a `/dev/tty??` minor device number tells the driver to allow the serial port to be opened even if the modem connected to it indicates that there is no carrier present. These new devices are traditionally named `/dev/cu??` because they are used for outgoing calls. (`cu` stands for "call up")

***Roland Schneider** is currently the TUUG secretary.*

Using Shared Memory for Inter-Process Communication

Part 1 of 3

By Peter Graham

More and more Unix programmers are finding themselves writing multi-process software. In many cases this is to take advantage of a networked environment and therefore the code makes use of facilities such as "sockets" (BSD Unix), "TLI" (System V Unix), and "R.P.C." (Remote Procedure Calls) to accomplish the necessary inter-process communication. It is, of course, possible to have multi-process software which runs on a single computer. When this is the case, a programmer can make use the above facilities (e.g. Unix-domain sockets) but other possibilities also exist. Among these are the old faithful "pipes", "fifos" (essentially named pipes), and, under many versions of Unix, shared memory.

With the advent of affordable multiprocessor technology (from companies including Sun Microsystems, Silicon Graphics, and Unisys), the importance of shared memory as a medium for inter-process communication has increased.

Efficiency is a prime consideration when developing parallel algorithms which use multiple processes to solve a single problem and shared memory is by far the most efficient mechanism available. As an additional benefit it also offers familiarity to many programmers who have not had exposure to network-based communications facilities.

The importance of shared memory will likely continue as computers with more and more processors become available. For example, at Stanford University in California, John Hennessy (one of the "inventors" of RISC) and his group have developed the DASH multiprocessor, the prototype of which is comprised of up to 64 RISC processors all accessing the same memory and all running Unix. Furthermore, DASH was explicitly designed to efficiently scale up to several thousand processors.

Using shared memory, multiple processes can construct shared data structures in an area of memory which they can all access. If you are familiar with the message passing paradigm, you can think of a message queue being built in memory with client processes placing messages into the queue and a server process removing them. In general though, it is better to consider the processes to be sharing a data structure since this is more typical of a parallel program. As an example, one application might have several processes concurrently analyzing an image which is stored in shared memory. After a little bit of setup, the data in shared memory can be accessed in the same way as any other data in the program.

There is one small "wrench in the works" though. In our suggested application, each process merely "analyzes" the image or, perhaps, a part of it. If we have a multiprocessor machine then a number (possibly all) of these processes may run concurrently giving us our performance improvement. But what happens if rather than analyzing (reading) the data, one or more processes are actually modifying (writing) it? Somehow we must provide "synchronization" between the processes so that they don't interfere with one another. (e.g. We might want to ensure that one process doesn't read the image data before another process has changed it in some way.) Thus, although we access shared memory variables just like any others, we must be careful about when we access them. The programmer must provide explicit synchronization control by coding "semaphore" calls. With message passing and R.P.C. the synchronization is effectively builtin. Having to deal with synchronization is the price you pay for the generality and efficiency of using shared memory.

We will begin by looking only at Unix's shared memory facilities and ignore the question of synchronization until later. A brief example which only reads from shared memory will be presented to illustrate the use of the shared memory system calls.

To begin with, since the processes which may want to access a "segment" of shared memory may not be related (e.g. child and parent), each shared memory segment needs a name. This "name" (actually just a unique identifier) is constructed in a rather odd way using the 'ftok()' function which is a part of the System V standard C library. The function maps a pathname and a single character to a unique "key" (our "name"). This key is then used in creating or opening a shared memory segment. It is, of course, assumed that all processes wishing to share a segment of memory have agreed upon a pathname and character to use a-priori.

Armed with a key, we can use the 'shmget()' system call to "get" a shared memory segment. This call accepts the key value returned by 'ftok()', a size for the segment (in bytes), and an integer flag argument. Without getting into too much detail (that's what man pages are for ;->) the flag specifies certain characteristics of the shared segment including the types of access you require to it (i.e. read, write, ...). In most cases, if the segment does not already exist, it will be created and in either case, an integer "shared memory identifier" (much like a file or socket identifier) will be returned.

At this stage, the shared memory segment has been created but it is still inaccessible. You must attach (or "map") the segment into your address space in order to make it accessible to your process. This is done using the 'shmat()' system call. You give 'shmat()' the segment identifier and some other information and it will return an address ('char *') which is where the segment starts in your address space (i.e. it returns a pointer to the segment). You can now freely access data in the segment using the pointer just as you would any other dynamically allocated data object.

When you are finished with the shared memory segment, you can call 'shmdt()' to detach the segment. This does not destroy the segment (since other processes may still be using it) but merely "unmaps" it from your address space. The final process, when finished with the segment, may detach and then actually remove it from the system. The final

removal is accomplished using a shared memory control operation which is performed via the 'shmctl()' system call.

In the following (contrived) example, there is a header file 'sv.h' which contains the type definitions for a "shared vector." The process running the code in 'init.c' creates and initializes the shared memory segment containing a shared vector of 10000 integers. It then sleeps for 60 seconds allowing us time to run ten processes which each read some data in the shared vector without fears of synchronization problems. Finally, it removes the shared segment long after the other processes are done with it. The program 'forker.c' is used to create the ten processes which will access the shared vector data. It simply forks ten time to create ten processes each of which executes the code in 'sum.c'. When a process running 'sum.c' executes, it receives a "process number" (not a pid) in the range 0 to 9 as its only argument. This is provided by 'forker.c' and is used by each "sum" process to determine which tenth of the array's elements it should sum. (e.g. Process number 4 sums elements 4000 through 4999.) Each process prints its result and then exits after detaching from the shared segment.

Here is some sample output from running on a SPARC-station. Notice that we run the init program in the background so we can start forker once it has finished the shared vector initialization. The order of the output statements from the sum processes is dependent upon the scheduling order of the processes. As can be seen in 'init.c,' the vector was initialized in such a way as to produce the easy to verify results. :-)

```
% init &
[1] 15021
Shared Vector is initialized, sleeping for 1
minute.
% forker
Process number 3 calculates the sum 3000.
Process number 4 calculates the sum 4000.
Process number 2 calculates the sum 2000.
Process number 7 calculates the sum 7000.
Process number 9 calculates the sum 9000.
Process number 8 calculates the sum 8000.
Process number 0 calculates the sum 0.
Process number 1 calculates the sum 1000.
Process number 6 calculates the sum 6000.
Process number 5 calculates the sum 5000.
[1]   Exit 1      init
%
```

Next time we will get into the details of doing process synchronization using semaphores so that we can do useful things with shared memory.

Peter Graham *is a PhD student in Computer Science.*

```
/********/
/* sv.h */
/********/

typedef struct sv {
      int   elts[10000];
} SV, *SVPTR;

/**********/
/* init.c */
/**********/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "sv.h"

#ifdef sun
/* necessary because of problem with */
/* SunOS include file shm.h          */
#define SHM_W 0200      /* shm write permission */
#define SHM_R 0400      /* shm read permission  */
#endif

main()
{
      int   i;          /* simple counter       */
      key_t sv_shmkey;  /* key to shm segment   */
      int   sv_segid;   /* shm segment ID       */
      SVPTR sv_segaddr; /* ptr to mapped seg    */
      /* need for call to 'shmctl()' */
      struct shmid_ds   *sv_shmbufptr;

      /* This code creates the shared memory    */
      /* segment and initializes it.            */
      /* Setup to access shared memory vector by*/
      /* creating a unique key/name for it ...  */
      if ((sv_shmkey=ftok(
       "/home/cs/staff/pgraham/misc/tuug/forker.c",
                  'M'))==-1) {
            printf(
           "Couldn't create shared memory key.\n");
            exit(-1);
      }

      /* calling shmget to create it            */
      /* (Note: IPC_CREAT)...                   */
      if ((sv_segid=shmget(sv_shmkey,sizeof(SV),
                  IPC_CREAT|SHM_R|SHM_W))==-1) {
            printf(
      "Couldn't get the shared memory segment.\n");
            exit(-1);
      }

      /* ... and mapping it into address space  */
      /* at address returned in 'sv_segaddr'.   */
      if ((sv_segaddr=(SVPTR)
                  shmat(sv_segid,(char *)0,0))
                        ==(SVPTR) (-1)) {
            printf(
       "Couldn't attach shared memory segment.\n");
            exit(-1);
      }

      /* Do the initialization!                 */
      /* The first 1000 elts will contain 0,    */
      /* the next 1000 will contain 1,          */
      /* the next 1000 will contain 2, & so on. */
      for (i=0;i<10000;i++) {
            sv_segaddr->elts[i]=(i/1000);
      }
```

```
      printf("Shared vector is initialized, ");
      printf("sleeping for 1 minute.\n");
      /* allow time to run summing processes    */
      sleep(60);

      /* detach the shared memory segment       */
      if (shmdt((char *) sv_segaddr)==-1) {
            printf(
      "Couldn't detach shared filename buffer.\n");
      }

      /* now remove shared segment altogether   */
      if (shmctl(sv_segid,IPC_RMID,sv_shmbufptr)
                              ==-1) {
            printf(
        "Couldn't remove shared record buffer.\n");
      }
}

/************/
/* forker.c */
/************/

main()
{
      int   i,    /* simple counter             */
            pid;  /* forked process' pid        */
      /* space for character form of pid        */
      char  argstr[16];

      /* This code simply creates ten identical */
      /* processes to run the program 'sum.c'   */
      /* which sums 1000 elts of the array.     */
      /* Each process is passed its process #   */
      /* (not pid) so that it knows what part   */
      /* of the array to operate on.            */
      for (i=0;i<10;i++) {
            pid=fork(); /* fork a new process   */
            if (pid==0) {
                  /* we are child process so... */
                  /* format our process number  */
                  sprintf(argstr,"%d",i);
                  /* execute the sum program    */
                  /* passing it the process #   */
                  execl(
            "/home/cs/staff/pgraham/misc/tuug/sum",
                        "sum", argstr,(char *)0);
            }
      }
} /* end main */

/*********/
/* sum.c */
/*********/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "sv.h"
#ifdef sun
/* necessary because of problem with */
/* SunOS include file shm.h          */
#define SHM_W 0200      /* shm write permission */
#define SHM_R 0400      /* shm read permission  */
#endif

main(argc,argv)
int   argc;
char  *argv[];
{
      /* process number received as an argument */
      int   myprocessnum,
            i,    /* simple counter             */
            sum;  /* the sum accumulated        */
      key_t sv_shmkey;  /* key to shm segment   */
      int   sv_segid;   /* shm segment ID       */
      SVPTR sv_segaddr; /* ptr to mapped seg    */

      /* This code simply sums up the 1000      */
      /* elements in the shared memory          */
      /* array specified by 'myprocessnum'.     */

      /* Setup to access shared memory vector by*/
      /* creating a unique key/name for it ...  */
      if ((sv_shmkey=ftok(
       "/home/cs/staff/pgraham/misc/tuug/forker.c",
                        'M'))==-1) {
            printf(
           "Couldn't create shared memory key.\n");
            exit(-1);
      }

      /* ... calling shmget to "open" it -      */
      /* it's already been created ...          */
      if ((sv_segid=shmget(sv_shmkey,sizeof(SV),
                        SHM_R|SHM_W))==-1) {
            printf(
      "Couldn't get the shared memory segment.\n");
            exit(-1);
      }

      /* ... and mapping it into address space  */
      /* at address returned in 'sv_segaddr'.   */
      if ((sv_segaddr=(SVPTR)
                  shmat(sv_segid,(char *)0,0))
                        ==(SVPTR) (-1)) {
            printf(
       "Couldn't attach shared memory segment.\n");
            exit(-1);
      }

      /* extract process # from first argument  */
      sscanf(argv[1],"%d",&myprocessnum);

      /* lets do the summing                    */
      sum=0;
      for (i=myprocessnum*1000;
                  i<(myprocessnum+1)*1000;i++) {
            sum=sum+sv_segaddr->elts[i];
      }
      printf(
      "Process number %d calculates the sum %d.\n",
            myprocessnum,sum);

      /* detach the shared memory segment       */
      if (shmdt((char *) sv_segaddr)==-1) {
            printf(
      "Couldn't detach shared filename buffer.\n");
      }
}
```

Alpha Architecture Technical Summary

Dick Sites, Rich Witek

Reprinted from a Digital press release. [Note: "Alpha" is an internal code name. An official name will be announced soon.]

What Is Alpha?

Alpha is a 64-bit RISC architecture, designed with particular emphasis on speed, multiple instruction issue, multiple processors, software migration from VAX VMS and MIPS ULTRIX, and long lifetime. The architects rejected any feature that did not appear to be usable for at least 25 years.

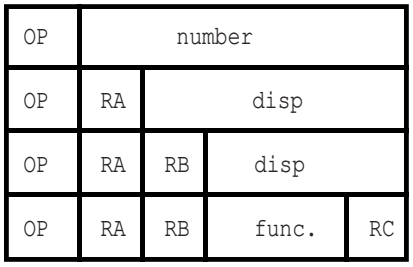
The first chip implementation runs at up to 200 MHz. The speed of Alpha implementations is expected to scale up from this by at least a factor of 1000 over the next 25 years.

Data Formats

Alpha is a load/store RISC architecture with all operations done between registers. Alpha has 32 integer registers and 32 floating registers, each 64 bits. Integer register R31 and floating register F31 are always zero. Longword (32-bit) and quadword (64-bit) integers are supported. Four floating datatypes are supported: VAX F-float, VAX G-float, IEEE single (32-bit), and IEEE double (64-bit). Memory is accessed via 64-bit virtual little-endian byte addresses.

Instruction Formats

Alpha instructions are all 32 bits, in four different instruction formats specifying 0, 1, 2, or 3 register fields. All formats have a 6-bit opcode.



PALcalls specify one of a few dozen complex operations to be performed.

Conditional branches test register RA and specify a signed 21-bit PC-relative longword target displacement. Subroutine calls put the return address in RA.

Loads and stores move longwords or quadwords between RA and memory, using RB plus a signed 16-bit displacement as the memory address.

Operates use source registers RA and RB, writing result register RC. There is an extended opcode in the 11-bit function field. Integer operates can use the RB field and part of the function field to specify an 8-bit zero-extended literal.

PALcall Instructions

The Privileged Architecture Library call instructions specify one of a few dozen complex functions to be performed. These functions deal with interrupts and exceptions, task switching, virtual memory, and other complex operations that must be done atomically. PALcall instructions vector to a privileged library of software subroutines (using the same Alpha instruction set) that implement an operating-system-specific set of these complex operations.

Branch Instructions

Conditional branch instructions can test a register for positive/negative or for zero/nonzero. They can also test integer registers for even/odd. Unconditional branch instructions can write a return address into a register. There is also a calculated jump instruction the branches to an arbitrary 64-bit address in a register.

Load/Store Instructions

Load and store instructions can move either 32- or 64-bit aligned quantities. The VAX floating-point load/store instructions swap words to give a consistent register format for floats. Memory addresses are flat 64-bit virtual addresses, with no segmentation. A 32-bit integer datum is placed in a register in a canonical form that makes 33 copies of the high bit of the datum. A 32-bit floating datum is placed in a register in a canonical form that extends the exponent by 3 bits and extends the fraction with 29 low-order zeros. 32-bit operates preserve these canonical forms.

There are no 8- or 16-bit load/store instructions, but there are facilities for doing byte manipulation in registers.

Alpha has no 32/64 mode bit or other such device. Compilers, as directed by user declarations, can generate any mixture of 32- and 64-bit operations.

Integer Operate Instructions

The integer operate instructions manipulate full 64-bit values, and include the usual assortment of arithmetic, compare, logical, and shift instructions. There are just three 32-bit integer operates: add, subtract, and multiply. These differ from their 64-bit counterparts ONLY in overflow detection and in producing 32-bit canonical results.

There is no integer divide instruction.

In addition to the operations found in conventional RISC architectures, there are scaled add/subtract for quick subscript calculation, 128-bit multiply for division by a constant and multiprecision arithmetic, conditional moves for avoiding branches, and an extensive set of in-register byte manipulation instructions for avoiding single-byte writes.

Rather then keeping a global state bit for integer overflow trap enable, the enable is encoded in the function field of each instruction. Thus, both ADDQ/V and ADDQ opcodes exist for specifying 64-bit add with and without overflow checking. This makes pipelined implementations easier.

Floating-point Operate Instructions

The floating operate instructions include four complete sets of VAX and IEEE arithmetic, plus conversions between float and integer.

There is no floating square root instruction.

In addition to the operations found in conventional RISC architectures, there are conditional moves for avoiding branches, and merge sign/exponent instructions for simple field manipulation.

Rather then keeping global state bits for arithmetic trap enables and rounding mode, these enable and mode bits are encoded in the function field of each instruction.

Significant Differences From Conventional RISC Processors

First, Alpha is a true 64-bit architecture, with a minimal number of 32-bit instructions. It is not a 32-bit architecture that was later expanded to 64 bits.

Second, Alpha was designed to allow very high-speed implementations. The instructions are very simple (no load-four-registers-unaligned-and-check-for-bytes-of-zero). There are no special registers that would prevent pipelining multiple instances of the same operations (no MQ register and no condition codes). The instructions interact with each other ONLY by one instruction writing a register or memory, and another one reading from the same place. This makes it particularly easy to build implementations that issue multiple instructions every CPU cycle. (The first implementation in fact issues two instructions every cycle.) There are no implementation-specific pipeline timing hazards, no load-delay slots, and no branch-delay slots. These features would make it difficult to maintain binary compatibility across multiple implementations and difficult to maintain full speed on multiple-issue implementations.

Alpha is unconventional in the approach to byte manipulation. Single-byte stores found in conventional RISC architectures force cache and memory implementations to include byte shift-and-mask logic, and sequencer logic to perform read-modify-write on memory words. This approach is awkward to implement quickly, and tends to slow down cache access to normal 32- or 64-bit aligned quantities. It also makes it awkward to build a high-speed error-correcting write-back cache, which is often needed to keep a very fast RISC implementation busy. It also can make it difficult to pipeline multiple byte operations.

Instead, the byte shifting and masking is done in Alpha with normal 64-bit register-to-register instructions, crafted to keep the sequences short.

Alpha is also unconventional in the approach to arithmetic traps. In contrast to conventional RISC architectures, Alpha arithmetic traps (overflow, underflow, etc.) are imprecise — they can be delivered an arbitrary number of instructions after the instruction that triggered the trap, and traps from many different instructions can be reported at once. This makes implementations that use pipelining and multiple issue substantially easier to build.

If precise arithmetic exceptions are desired, trap barrier instructions can be explicitly inserted in the program to force traps to be delivered at specific points.

Alpha is also unconventional in the approach to multiprocessor shared memory. As viewed from a second processor (including an I/O device), a sequence of reads and writes issued by one processor may be arbitrarily reordered by an implementation. This allows implementations to use multi-bank caches, bypassed write buffers, write merging, pipelined writes with retry on error, etc. If strict ordering between two accesses must be maintained, memory barrier instructions can be explicitly inserted in the program.

The basic multiprocessor interlocking primitive is a RISC-style load_locked, modify, store_conditional sequence. If the sequence runs without interrupt, exception, or an interfering write from another processor, then the conditional store succeeds. Otherwise, the store fails and the program eventually must branch back and retry the sequence. This style of interlocking scales well with very fast caches, and makes Alpha an especially attractive architecture for building multiple-processor systems.

Alpha includes a number of HINTS for implementations, all aimed at allowing higher speed. Calculated jumps have a target hint that can allow much faster subroutine calls and returns. There are prefetching hints for the memory system that can allow much higher cache hit rates. There are also granularity hints for the virtual-address mapping that can allow much more effective use of translation lookaside buffers for big contiguous structures.

Alpha includes a very flexible privileged library of software for operating-system-specific operations, invoked with PALcalls. This library allows Alpha to run full VMS using one version of this software library that mirrors many of the VAX operating-system features, and to run OSF/1 using a different version that mirrors many of the MIPS operating-system features, and similarly for NT. Other versions could be tailored for real-time, teaching, etc. The PALcalls allow Alpha to run VMS with hardly more hardware than a a conventional RISC machine has (the PAL mode bit itself, plus 4 extra protection bits in each TB entry). This library makes Alpha an especially attractive architecture for multiple operating systems.

Finally, Alpha is not strongly biased toward only one or two programming languages. It is an attractive architecture for compiling at least a dozen different languages.

Specifications (150MHz version).

Process Technology	.75 micron CMOS
Cycle Time	150 MHz (6.6 ns)
Die Size	13.9mm x 16.8mm
Transistor Count	1.68 million
Package	431 pin PGA
Number of Signal Pins	291
Power Dissipation	23 W at 6.6 ns cycle
Power Supply	3.3 volts
Clocking Input	300 MHz differential
On-chip D-cache	8 Kbyte, physical, direct-mapped, write-through, 32-byte line, 32-byte fill
On-chip I-cache	8 Kbyte, physical, direct-mapped, 32-byte line, 32-byte fill, 64 ASNs
On-chip DTB	32-entry; fully-associative; 8-Kbyte, 64-Kbyte, 256-Kbyte, 4-Mbyte page sizes
On-chip ITB	8-entry, fully associative, 8-Kbyte page plus 4-entry, fully-associative, 4-Mbyte page
Floating Point Unit	On-chip FPU supports both IEEE and VAX floating point
Bus	Separate data, address bus. 128-bit/64-bit data bus
Serial ROM Interface	Allows the chip to directly access serial ROM
Virtual Address Size	64 bits checked; 43 bits implemented
Physical Address Size	34 bits implemented
Page Size	8 Kbytes
Issue Rate	2 instructions per cycle to A-box, E-box, or F-box
Integer Pipeline	7-stage pipeline
Floating Pipeline	10-stage pipeline

TUUG Meeting Minutes

Tuesday, April 14, 1992, 7:30 PM
Senate Chambers
245 Engineering Bldg.
University of Manitoba
Ft. Garry Campus

Chair: Susan Zuk
Attendance: 44

Business meeting:

a) President's Report
* the name of the group is being changed from "Technical UNIX User Group" to "Manitoba UNIX User Group."
* we will be joining UniForum Canada
* membership renewals will be changed so that each member has his/her own "year" in order to harmonize membership renewal with UniForum.
* Manitoba Hydro on Taylor Av. will likely become our new "home" since Hydro has a good meeting room which we will be allowed to use.
* Eric Carsted is leaving for Houston. Kathy Norman will take over his job as meeting coordinator for the rest of the year.

b) Membership Report
* total membership is now 65.

c) Newsletter Report
* May issue deadline is April 17.
* Lots of articles coming in now.

d) New Business
* Muddy Waters Computer Society is having Computer Fest at the Convention Centre on April 26. We need volunteers to man the MUUG booth.
* University of Manitoba Computer services has donated use of a Sun 386i to our group.
  - this machine is connected to the Internet
  - will be used to set up UUCP, e-mail, network news, ftp, etc. for our members, available either via UUCP or interactively.
  - MUUG will have to provide disk space and is responsible for all maintenance.
  - usage policies, including fees, have not yet been decided.

Presented topic:
Unix on a Mainframe – Amdahl's UTS

Agenda

for
Tuesday, May 12, 1992, 7:30 PM
234B Engineering Bldg.
University of Manitoba
Ft. Garry Campus

1. Round Table 7:30

2. Business Meeting 8:00
   a) President's Report
   b) Membership Secretary's Report
   c) Newsletter Editor's Report
   d) Treasurer's Report
   e) Meeting Coordinator's Report
   f) New Business

3. Break 8:20

5. Presented Topic 8:30
   The Future of UNIX at Intel
   Jon Coxworth, Architect Manager
   Intel Corporation (Ottawa)
   There have been large chip wars going on the industry in the last while. One of the most controversial has been the discussion of RISC vs CISC technology. Jon will be discussing this topic as well as where Intel sees the future and what the company is doing to make it happen.

6. Adjourn 9:30

Note: Please try to arrive at the meeting between 7:15 and 7:30 pm. Thank You.

Next Month

Meeting:
Our June meeting is scheduled for Tuesday, June 2, at 6:30 PM (a week and an hour earlier than usual). This meeting will be the traditional TUUG June BBQ. This year, Roland Schneider is hosting it at his home in Selkirk. A map will be provided in next month's newsletter.

Newsletter:
We will likely continue with our Q&A column, and RPC Programming by Scott Balneaves, next month. We may also have part 2 on shared memory by Peter Graham, and several "filler" articles by Roland Schneider. Thanks again to all those who submitted those great articles throughout the year.