Pulling together for power on demand

MIKE CHARNLEY-FISHER DESCRIBES THE INNER WORKING OF HELIOS THE FIRST OPERATING SYSTEM THAT TRULY UTILISES TRANSPUTERS TO GIVE POWER IN A NETWORK.

While there are many machines that use the transputer as a source of extra power, as yet few use it as a central processor. In theory a transputer based network could supply computing power to whatever machine on the network needs it. Helios is a Unix compatible operating system that can truly exploit the power of this expandable micro-processor. In this article I will attempt to explain how Helios achieves this.

First of all, to define some terms. An operating system firstly provides an 'extended machine or virtual machine that is easier to program than the underlying hardware' (for example it lets me 'read' a disk drive without me having to manage starting the motor, moving the head). Secondly it provides 'for an orderly and controlled allocation of the processors, memories and I/O devices among the various programs competing for them' (it stops this program overwriting memory used by one of my background tasks).

Over the top of this most system suppliers provide a set of utility programs to make life easier and a 'shell' or command interpreter to provide access to them.

Increasingly, system suppliers paste yet another level over the top of this – a nice friendly user interface or 'windows' system such as GEM or the Apple system. Comments about this are definitely not about the operating system.

Apart from the niceties of using the machine, these two upper levels have little effect on how the main part of an application gets ported to a machine. Having said this, Helios offers all three levels so we shall discuss them in turn.

I should point out that what I have to say about Helios is applicable to any host environment. Helios currently runs on Transputer cards fitted to Sun, Vax, Meiko, Commodore, IBM PC, and a number of other manufacturers equipment. More importantly a Helios network is independent of the host system – all of these machines can talk to each other.

The core

Aside from the multi-user, networking, and multi-tasking features of systems such as Unix, Helios is unique in that it provides multi-processor support across the network.

With modern 32-bit processors, portability is more important than performance (on the whole 32-bit processors are fast enough anyway). With this in mind, programs should be developed using calls as follows (only go down a level if it really makes a difference).

- Standard International language calls (eg. ANSI C);
- Standard International operating system calls (eg. POSIX);
- Internationally agreed graphics calls (eg. X-Windows, GKS);
- Vendor Specific Operating System Calls (eg. Helios, VMS, DOS);
- Device Specific Operating System Calls (eg. Blitter Support);
- Assembler (eg. Intel, Inmos – You must be desperate to get here).

There are obviously exceptions to the above rule but the days of 'this program was written in 100 per cent assembler' are over. Many of the problems attributed to the delay of OS/2 have been reportedly attributed to a high assembler content.

For an in-depth understanding of Helios, the reader is referred to the Perihelion documentation. However, a word of warning, it is not for the light hearted. If you have not been taught operating system theory you will need to read some fairly advanced texts on operating systems. The only comment I will make is that Helios represents state-of-the-art programming techniques. In particular, it is significantly more modular than systems such as Unix or OS/2. This is likely to result in far higher stability, greater ease of update, and far greater opportunities for adding extra functionality.

The unified way in which everything appears to the user is a major feature of Helios – everything: memory files, programs, processors, an 'Object'.

It is also important to stress that everything is treated the same. For example, if I want to print some output all I do is re-route my output to /printers/default – this 'virtual' file looks after printing it for me. And the really impressive feature – the whole concept applies to any processor on the network – even one 100 yards down the corridor. Networking takes on a new meaning!

So far I have described how the multi-tasking and networking side of Helios looks to the user – these are the facilities which Unix, OS/2, can give (but less elegantly). The real point of Helios is its ability to handle the multi-processor environment.

Consider the set-up in the linked transputer figure. The keyboard, screen, effectively act through Transputer 00 (via the I/O processor). At the simplest level consider running a program on Processor 01. To do this under Helios it is very simple. For example, typing 'remote 01 emacs'. Helios would look after transferring the program (emacs) to the memory of <01> and, provided language or O/S level calls were used, it would manage displaying my results correctly on the screen – it makes no difference whether I use processor <00>, <01>, or one 100 yards down the corridor. In the background, Helios looks after sending a message from <01>, via a link, to <00>, asking it to display the text (in much the same way as a mainframe to terminal link words – although significantly faster). The same applies to graphics (as I soon discovered). It is no good writing a graphics program which addresses video memory directly if you want to run it on any processor other than the one connected to video memory – it does not work! – you have to send messages (and this is where X-Windows comes in).

This is all very nice – provided you have access rights, you can steal your neighbour's processor to run that heavy analysis program while you carry on working on other things – and it is all very easy. However, Helios takes it a step further. Provided your application is written in a very modular fashion (using facilities such as Unix pipes, for example), Helios will distribute your application over available processors. All you have to do is tell Helios how each module is connected (using an extension of the Unix/DOS pipe syntax) and Helios will hunt down free processors and execute each part of the program on them. If it cannot find a sufficient number, then it simply resorts to multi-tasking. This principle extends to putting multiple copies of the same program across multiple processors to split a big set of data into lots of smaller sets. Helios will 'farm' it out and receive results just as if it were one big set of data running on one transputer. This is the function of the Task Force Manager (tfm) program.

Because Helios uses messages passing and distributed search algorithms to hunt out processors, there is another advantage. Helios has built in reliability. Unlike any networks that I have used so far, pulling a transputer, or a complete workstation, off the network will only affect performance and remove resources attached to that node – it will not crash the network (unlike the Apple set-up, most PC systems).

The main point of all of this is that Helios will take an existing Unix application and run it with little or no modification. Furthermore, if it has been written to take advantage of pipes and multi-tasking, it is relatively easy to distribute that program across multiple processors – all without affecting compatibility with other Unix-based workstations. Of course, if you want to go into finer levels of detail Helios provides all of its specific operating system calls as well as access to the raw instruction set of the Transputer (Helios will manage processors not actually running Helios.)

The Shell (or Command Interpreter)

The Helios Shell is intended to be similar to that provided under Unix with all the usual two letter Unix commands. If you do not like these two letter commands and want to use DOS commands instead, it is very easy to set-up synonyms using the 'alias' command (eg 'alias dir ls –l' makes 'dir' work as it does under DOS). There are one or two really nice commands, my favourite probably being 'cache'. If you 'cache' a program and then execute it, it will always stay in memory as an object module. The next time you run it, it simply executes (no loading from a file, no copies in a RAM disk, just the object). This makes for very rapid edit/compile/link cycles.

Unlike the journalists who write for the typical PC magazines, I have to say that I much prefer my Unix type shell to that offered by MSDOS (and I used DOS for two years before using this).

The full X-Windows manager is due to be available soon. In the meantime Perihelion ships 'Eddie' – what they call a 'Dirty Window Manager'. Whilst this does not have all the icons associated with most windowing systems, it does allow multiple workspaces. On a multi-tasking machine this is a necessity. On the Abaq this makes writing programs a joy. I can be compiling in one window, editing the source against the errors in another, and running it in another. And, what is more, I am not limited to a few visible lines/columns per window. Each window is a full PC screen sat side-by-side (unless you have more than four, in which case they start becoming half-PC screens).

The User Interface (X-Windows)

X-Windows is rapidly becoming the *de facto* industry standard for windowing under Unix. X-Windows is a cornerstone of the X-Consortium standardisation effort. The X-Consortium consists of AT&T, Bull, DEC, Fujitsu, Hewlett-Packard, ICL, NCR, Nixdorf, Nokia, Olivetti, Philips, Siemens, Sun, and Unisys, and will shortly be joined by IBM. Add to that the active promotion of X-Windows by Sony, Apple, Tektronix, Data General, Microsoft, Prime, and Adobe, and the scale of this support becomes apparent. X-Windows is very important (although the full extent of this was not appreciated when Perihelion adopted it).

The core to X-Windows is a 'virtual terminal' server. What this means is that, instead of an application writing directly to the screen, or reading directly from the mouse or keyboard, the application writes to, and reads from, X-Windows. This has the advantage that programs need not concern themselves with the hardware it is running on. As far as an application program is concerned it does not matter whether it is addressing a Sun, an IBM RT, an Apollo, a Hewlett Packard, or a workstation operating under Helios – it all looks the same (The list of Hardware vendors is not exhaustive).

This portability has a price. As you can imagine, in order to write a dot to the screen an application has to write to X-Windows, which in turn writes to the screen. This affects performance, and is the reason why it has not appeared on anything but the high performance workstation so far. In the same way memory is used up – you need the 'server' to display the graphics and you need to embed the 'library' in your application program to use it. The core X-Windows simply provides routines for drawing points, lines, filled patterns, fonts, pixel maps (coloured bit maps), to the screen, and routines for taking input from the keyboard, mouse, and other input devices, and passing them onto the right application. This latter aspect is important. In a multi-program environment, spread across different windows, you only want mouse movements and keyboard input to affect specific applications.

The core X-Windows manages all of this for you. In addition X-Windows provides routines to query the optimum characteristics of the hardware. Along with this core system, X-Windows provides numerous utility programs. Among these are Window Managers, bit-map editors and toolkits. The difference between a 'windows system' and a 'windows manager' is important. A window manager dictates what commands are available and what the windows and icons look like to the user. All the windows system does is provide the tools to do this. Thus Microsoft Windows, Apple Windows, and any others, can all run using X-Windows but all look totally different to the user.

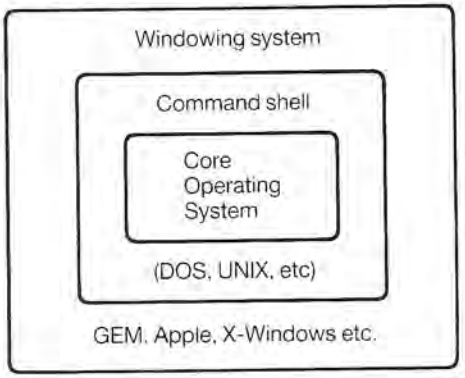
In addition to vast amounts of 'Contributed' software (22 MBytes of Source in addition to the 20Mbytes provided by X-Windows itself), X-Windows helps the applications designer by providing 'toolkits'. These toolkits provide higher level views over X-Windows and allow you to create menu bars, icons, scrolling text windows, etc at a much higher level. To those of you who have programmed using Apple's Hyper-Card, X-Windows gives to the C programmer what Hypertalk gives to the Hyper-Card programmer (in terms of graphics functions).

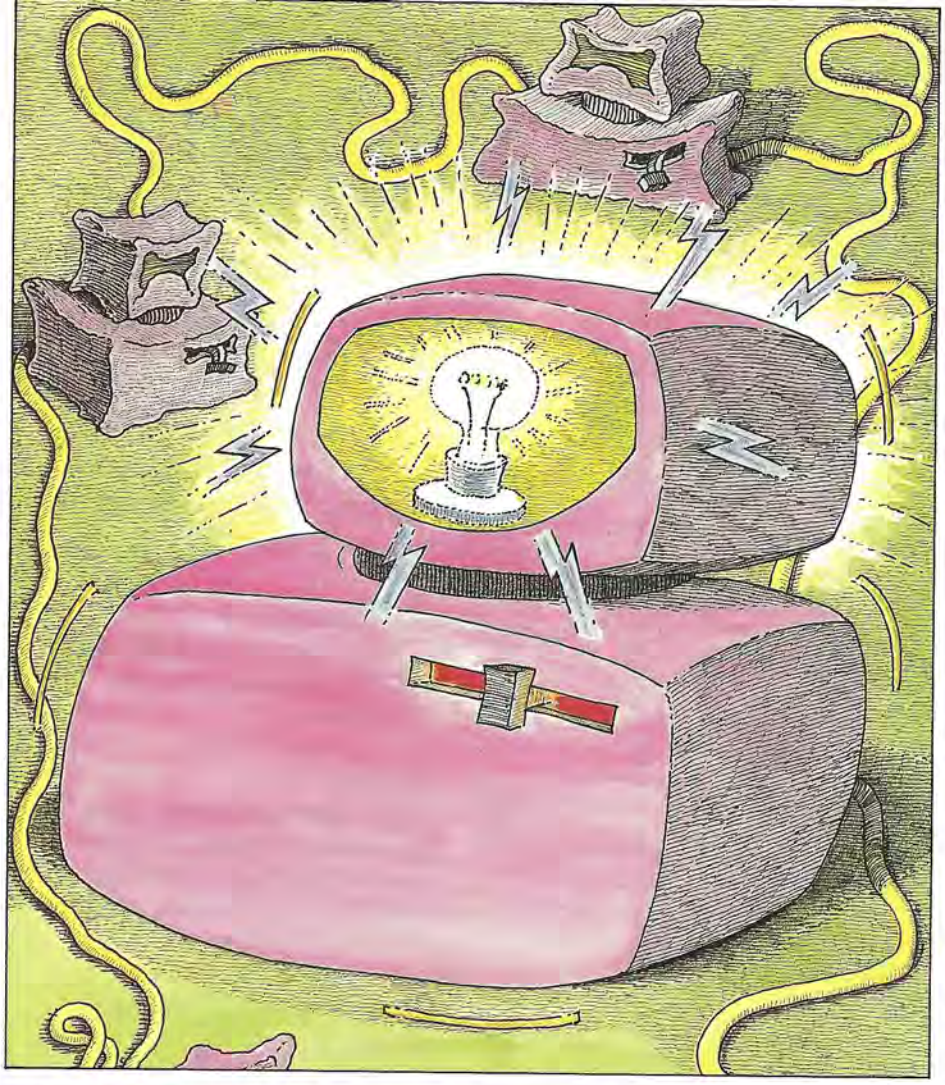
The key to X-Windows, and the reason it is ideally suited to networked systems, is that it is based around message passing. It is designed to talk to machines across a network, (and this message passing

POSIX Compatible System Calls

_exit	abort	access
alarm	cf_getispeed	cf_getospeed
cf_setispeed	cf_setospeed	chdir
chmod	close	closedir
creat	ctermid	cuserid
dup	dup2	execl
execle	execlp	execv
execve	execvp	exit
fcntl	fdopen	fileno
fstat	getcwd	getenv
getlogin	getpwent	getpwnam
getpwuid	isatty	kill
link	lseek	mkdir
mkfifo	open	opendir
pause	pipe	read
readdir	rename	rewinddir
rmdir	sigaction	sigaddset
sigdelset	sigfillset	siginitset
sigismember	siglongjmp	signal
sigpending	sigprocmask	sigsetjmp
sigsuspend	sleep	stat
tcdrain	tcflow	tcflush
getattr	tcsendbreak	tcsetattr
time	times	ttyname
unmask	uname	unlink
utime	vfork	wait
wait2	write	

Differences with the full POSIX standard include the following:-
* All user and group id functions excluded
* fork() replaced by vfork()
* job control omitted (possible future extension)





mechanism is available to the programmer). Like Helios, it can handle disruptions to the network.

Perhaps you can now begin to see how X-Windows fits hand in glove with Helios. With X-Windows it does not matter which processor you run your application on – it will always get routed to X-Windows, which in turn will always put it to the screen, and return any keyboard/mouse events.

Moreover, close examination of the X-Windows low-level routines indicates that the Charity chips are ideally suited to this environment (and in fact can do a lot more).

I could say a lot more about X-Windows but even after a week's intensive course (given by IXI) I still need a month to play around with it to get a reasonable grasp.

Of course, the use of X-Protocol underlying X-Windows should mean that Abaq's will be able to talk to Sun's, Apollos', DEC's, IBM's, Apple's.

X-Windows is constantly being extended. 3D extensions, based on the PHIGS standard are currently being implemented and of course the library of 'contributed' software grows with every release.

Program development tools

Helios comes with a 'C' compiler, a transputer assembler/linker, make, and various other development aids. The 'C' compiler is pleasant to use, certainly a lot nicer than the 'C' compiler I recently used on an Apollo Domain 3000. It is ANSI standard (well – very very close – one or two ANSI features have not yet been implemented). It is based on the well respected compiler developed for the Acorn ARM RISC chip. The reason for using this rather than a typical PC compiler is that the transputer, like the ARM, has very few registers (if you exclude on-chip RAM).

Provided you have the access rights, you can steal your neighbour's processor to run that heavy analysis program.

The assembler/linker is Perihelion's own, but it is able to handle INMOS standard code and uses the same mneumonics.

Debugging is a bit more of a problem because the Transputer has no low level debugging instruction. However, Perihelion provides a retrospective debugging facility through the server which is better than nothing. In practice I have not yet had real reason to use it – the C compiler is very helpful and picks out most errors (such as most incorrect uses of pointers – unlike that previous compiler I mentioned). However, a source level debugger is not out of the question in the future – it is very active in Perihelion's mind.

In addition to the ANSI standard library, the C compiler has access to most of the routines which the system uses. For example, there is access to semaphores (read that book on operating systems if you don't know what these are), message passing, linked lists, object and stream creation and manipulation, and other system related functions, such as getting at the Clock.

Although undocumented at this time, there is also access to the relatively fast block move instruction in the transputer, and there are routines to make the best of the transputer's fast cache.

In terms of other languages, the Top Express/Maiko Fortran compiler is being ported to run under Helios, INMOS are working on a version of OCCAM, there are three LISP compiler developments, two Prologs, and even a BASIC, somewhere in the pipeline. Although not strictly speaking a language, there is a company doing a PC emulator (although I rarely have recourse to use my MSDOS machine now!).

Of course, the fact that I am using MicroEmacs for the first draft of this, indicates that it is quite easy to port public domain languages such as XLISP over.

Stability

The lack of memory management hardware on the transputer has led some commentators to doubt the stability of Helios as a multi-tasking, multi-user operating system. Considering I have been using a pre-release version of Helios (Version 0.3) until very recently (I have just upgraded to a pre-release version 1.0) and regularly use the multi-tasking capabilities of Helios, I have to say I have had no real problems. By habit now, I have four copies of the Shell running at the same time (one of them running on my other processor). If I crash out of one, the others work quite happily. (Note that the standard PC also has no hardware memory management – what about all of those resident pop-up programs?).

The future

Helios is moving very quickly. Perhaps one of the more interesting areas at the moment is the development of 'parallelising' compilers. At the moment most of the language implementations, apart from Occam, handle parallelism at the task level (including the 'parallel' compilers!). Perihelion, like many others are researching this area. Fortunately they see that the C++ object oriented extension to 'C' could provide a handle on which to implement this – so a true C++ compiler may well appear next year. This has other benefits since it is quite feasible that the current popularity of C may result in C++ becoming the industry standard object oriented language (supplanting LISP perhaps?).