SAND2011-3681P

NIX, a prototype operating system for heterogeneous manycore CPUs

Ron Minnich
Francisco J. Ballesteros
Gorka Guardiola
Enrique Soriano
(want your name here? Just ask!)

ABSTRACT

This paper describes NIX, a prototype operating system for future manycore CPUs. NIX features a heterogeneous CPU model and a change from the traditional Unix memory model of separate virtual address spaces. NIX has been influenced by our work in High Performance computing, both on Blue Gene and more traditional clusters.

NIX partitions cores by function: Timesharing Cores, or TCs; Application Cores, or ACs; and Kernel Cores, or KCs. There is always at least one TC, and it runs applications in the traditional model. KCs are optional cores created to run kernel functions on demand. ACs are also optional, and are entirely turned over to running an application, with no interrupts; not even clock interrupts. Unlike traditional HPC Light Weight kernels, functions are not static: the number of TCs, KCs, and ACs can change as needs change. Unlike a traditional operating system, applications can access services by sending a message to the TC kernel, rather than by a system call trap.

Control of ACs is managed by means of active messages. NIX takes advantage of the shared-memory nature of manycore CPUs, and passes pointers to both data and code to implement the active messages.

1. Introduction

NIX is a new operating system, evolving from a traditional operating system in a way that preserves binary compatibility but also enables a sharp break with past practice. NIX is strongly influenced by our experiences over the past five years with running Plan 9 on some of the largest supercomputers in the world. It has been said that supercomputing technology frequently tests out ideas that later appear in general purpose computing, and certainly our experience is no exception.

This paper is more concerned with documenting the state of the NIX design, rather than rationalising why the decisions were made. That needs to be done too.

NIX is designed for manycore, heterogeneous processors. The manycore is probably obvious at this point. The idea of hetereogeneity derives from discussions we have had with several vendors. First, the vendors would prefer that not all cores run an OS at all times. Second, there is a very real possibility that on some future manycore systems, not all cores will be able to support an OS. We have seen some flavor of this with the Cell which, while a failed effort, does point toward a possible future direction.

NIX uses a messaging protocol between cores based on shared-memory active messages. The protocol uses a shared memory data structure, containing cache-aligned fields. In particular, it contains arguments, a function pointer, to be run by the peer core, and an indication for flushing TLB (when required).

2. Key NIX attributes

There are a few common attributes to todays manycore systems. The systems have a non-uniform topology. Sockets contain CPUs, and CPUs contain cores. Each socket is connected to a local memory, and can reach other memory only via other sockets, via one or more hops; access to memory is hence non-uniform. Core-to-core and core-to-memory communications time is non-uniform.

A very important aspect of these designs is that the varying aspects of the topology affect performance, but not correctness. We preserve an old but very important principle in successful computer projects, both hardware and software: things that worked will still work. We are trying to avoid the high risk of failure that comes with clean sheet designs. NIX thus begins life with a working kernel and full set of userland code.

The idea to keep a standard time-sharing kernel as it is now, but be able to exploit other cores for either user or kernel intensive processes. In the worst case, the time-sharing kernel will be able to work as it does now. In the best case, applications will be able to run as fast as permitted by the raw hardware.

Source code, compiled for Plan 9, also work son NIX. We make an even stronger guarantee:if a binary worked on Plan 9, it will work on NIX.

NIX partitions cores by function, hence implementing a type of heteregeneity. The basic x86 architecture has, since the advent of SMP, divided CPUs into two basic types: BSP, or Boot Strap Processor; and AP, or Application Processor[1]. In a sense, x86 systems have had heterogeneity from the start, although in keeping with the "things still work" principle, it was not visible for the most part at user level. NIX preserves this distinction, with a twist: we create three new classes of cores:

1 The first, TC, is a time sharing core. The BSP is always a TC. There can be more than one TC, and in fact a system can consist of nothing but TCs, as determined by the needs of the user.

2 The second, AC, is an Applications Core: only APs can be Application Cores. Application cores run applications, in a non-preemptive mode, and never field interrupts.

3 The third, KC, is a Kernel Core. Kernel Cores can be created under control of the TC. KCs do nothing but run OS tasks for the TC. A KC might, for example, run a file system. KCs never run user mode code.

Cores hence have roles. Part of the motivation for the creation of these roles comes from discussions with vendors. While cores on current systems are all the same −with a few exceptions, such as Cell− there is no guarantee of such homogeneity in future systems. In fact, heterogeneity is almost guaranteed by a simple fact: systems with 1024 cores will not need to have all 1024 cores running the kernel. Designers see potential for die space and power savings if, say, a system with N cores has only $sqrt(N)$ cores complex enough to run an operating system and manage interrupts and I/O. At least one vendor we have spoken with evinced great dismay at the idea of taking a manycore system and treating it as a traditional SMP system[2].

While future manycore systems may have heterogeneous CPUs, one aspect of them it appears will not change: there will still be shared memory addressable from all cores. NIX takes advantage of this property in the current design. NIX-specific communications for management are performed via *active* messages, called ‘‘inter-core-calls’’, or ICCs. An ICC consists of a structure containing a pointer to a function, an indication to flush the TLB, and a set of arguments. Each core has an ICC structure associated, and polls for a new message. The core, upon receiving the message, calls the function with the arguments. Note that the arguments can be pointers, because we are sharing memory. ACs sit in a simple command loop, waiting for commands, and executing them as needed. KCs do the same. TCs execute standard time-sharing kernels (the BSP also coordinates all other cores).

Cores can change roles, again under the control fo the TC. A core might be needed for applications, in which case NIX can direct the core to enter the AC command loop. The TC can also direct the core, via an active message, to exit the AC loop and re enter the group of TCs. At present, only one core −the BSP−

[1] It is actually a bit more complex than that, due to the advent of multicore. On a multicore socket, only one core is the "BSP"; it is really a Boot Strap Core, but the vendors have chosen to maintain the older name.

[2] Direct quote: ‘‘It would be a terrible waste to run Linux on every core of this processor, but that’s all most people are thinking of’’.

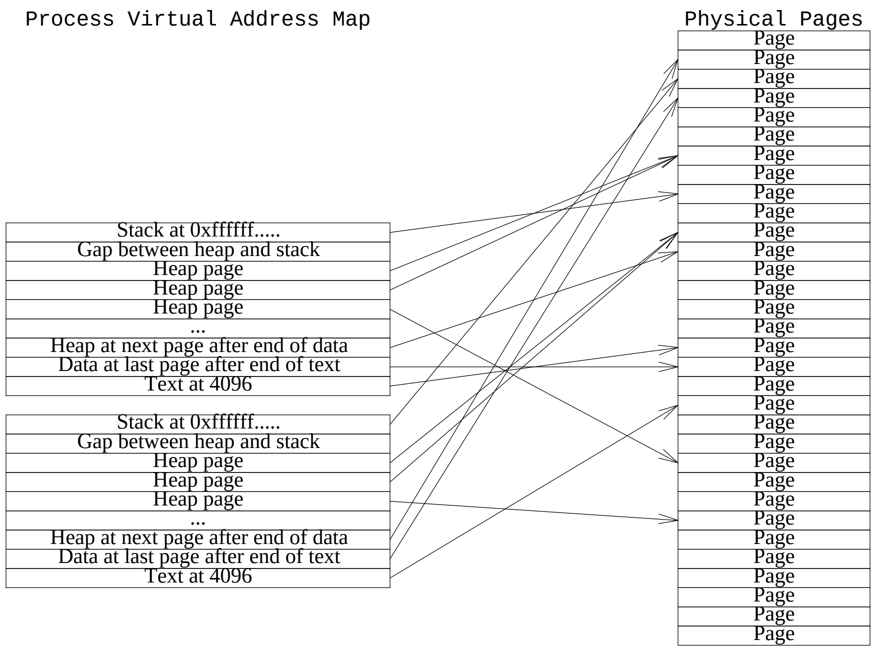
boots to become a TC; all other cores boot and enter the AC command loop. This will be fixed.

Processes can also change their mode. All processes are started on the TC and, depending on the type of *exec* being performed, can optionally transition to an AC. If the process takes a fault, makes a system call, or has some other problem, it will transition back to the TC for service. Once serviced, the process may resume execution in the AC.

Binary compatibility for processes is hence rather easy: old processes only run on a TC. If a user makes a mistake and starts an old binary on an AC, it will simply move back to a TC at the first system call and stay there until directed to move. If the binary is mostly spending its time on computation, it can still move back out to an AC for the duration of the time spent in a computational kernel. No checkpointing needed: this movement works because the cores share memory.

3. Things not done yet

NIX has a changed memory model from the old UNIX model used for the last forty years. The reader should recall that the early machines UNIX ran on had very tight memory constraints: 16 4096-byte pages. Further, the 16-bit memory address on those machines could not address all of memory once the 11/45 and later models were introduced. Finally, it made sense to provide a virtual address space that was the same for all binaries, since it made for more efficient code. Hence, we have the memory address scheme show below.


Process Virtual Address Map
Physical Pages
Stack at 0xffffff.....
Gap between heap and stack
Heap page
Heap page
Heap page
...
Heap at next page after end of data
Data at last page after end of text
Text at 4096
Stack at 0xffffff.....
Gap between heap and stack
Heap page
Heap page
Heap page
...
Heap at next page after end of data
Data at last page after end of text
Text at 4096
Page

As the figure shows, with a standard virtual address range, even though processes have a near-identical address layout, the same virtual address in different processes point to different pages (except in the special case of the code for shared libraries and binaries). This setup works well for code and data. Code and data can be linked to always start at the same virtual address, greatly simplifying the task of starting a process and managing the use of multiple processes sharing one image.[3] For dynamic addresses, such as the stack and the heap, such simplification is supported but not required. The stack address can always be derived from the stack pointer register, and the heap is determined from a system call (*brk*). Most systems that use threads already allow for a variable stack location, specified when the thread is started. Unlike the older model shown above, most newer Unix systems also use disjoint heaps; libraries use *mmap* to allocate data, not *brk*, and the mapped data areas need not be virtually contiguous with any other area.

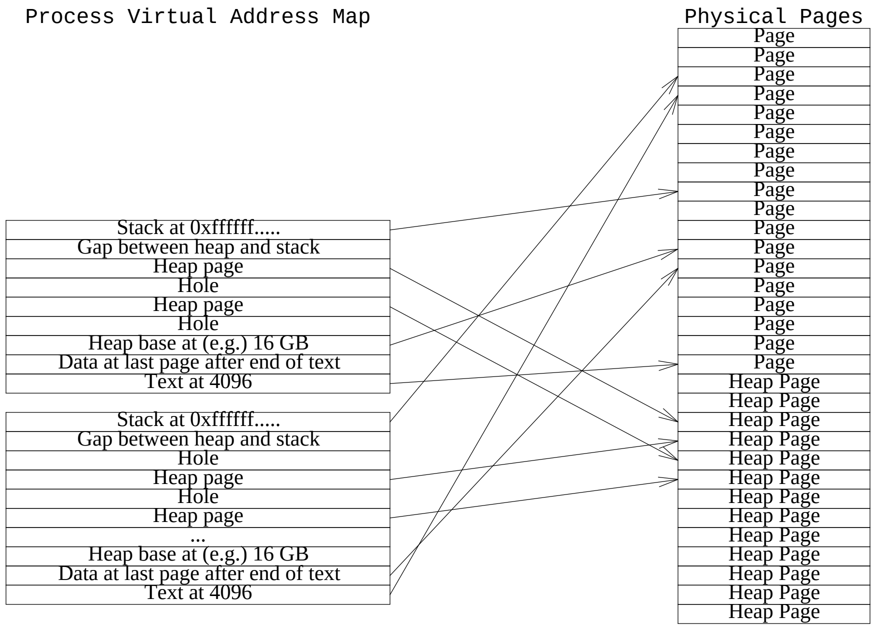
[3] On some older systems, the virtual address space of the code can change each time it is run, requiring support from the runtime, architecture, and even the operating system.

For example, /bin/cat on Linux has at least five different sets of heap pages, none of them adjacent to each other.

The simplification offered by a common virtual address range is not without problems. Processes can not share memory easily, because a heap address in one process does not point to the same data when passed to another process. This one limitation has felled more trees and spilled more ink at more conferences in the last 40 years than one might expect; it even led researchers at Berkeley to modify their PDP-11 hardware to support data transfer between user mode processes with the move from previous address space instruction[4]. Process addresses are not the addresses used by the kernel, which lives in its own address space. Still worse, interrupt code, which does not run in any process context, can not do direct I/O to process memory without a great deal of additional complexity. Pages must be "pinned", to make sure they don't move around, and data structures must be created and maintained to ensure that all the correct cleanup and maintenance is done. There is the further problem of I/O. Process heap virtual addresses have no meaning to I/O devices, most of which do not handle virtual addressing, and even further work must be maintained in, not only kernel structures, but structures managed by the operating system on the I/O cards. Quite a bit of work has been done to maintain the unique heap address space that is virtually contiguous; which, as we have seen, is not even needed any more.

In our work on Blue Gene we found a way to get rid of this limitation: we got rid of the fiction. Our kernel on Blue Gene sets up processes so that heap virtual addresses are not unique, and in fact are directly mapped from virtual to physical addresses. For heap, physical and virtual addresses are the same. The result is that a process heap address has the same meaning to the process, all other processes, the kernel, the interrupt drivers, and the I/O devices. A process can initiate I/O to an address without having to do any computations: the address has universal meaning. Processes can share memory without worrying about whether a given memory range in one process can fit in another.

A condensed version of the address space structure we are using on Blue Gene is shown below. We will be using some variant of this on NIX. One question: should heap be above or below the stack? For now, we leave the relative locations unchanged, although threaded programs intermix heap and stack pages without much trouble. As can be seen, heap pages are mapped to a corresponding physical page. Sharing between processes is easy given this design; a process need merely indicate which other process should be granted access to which part of its address space, and the kernel can install it in the other process should that other process attempt to access it.


Process Virtual Address Map
Physical Pages
Stack at 0xffffff.....
Gap between heap and stack
Heap page
Hole
Heap page
Hole
Heap base at (e.g.) 16 GB
Data at last page after end of text
Text at 4096
Stack at 0xffffff.....
Gap between heap and stack
Hole
Heap page
Hole
Heap page
...
Heap base at (e.g.) 16 GB
Data at last page after end of text
Text at 4096
Page
Heap Page

[4] Fortunately, it was a one-wire change.

Applications can now transparently share heap addresses as needed with each other. Heap addresses are now valid in all processor modes. One other change is that heaps are no longer virtually contiguous. Heaps have holes, which, as mentioned above, is acceptable. Some runtimes, e.g. Go in 32-bit mode, will not function correctly with discontiguous heaps. At the same time, Go only needs about 1 Gbyte of contiguous memory, so it is possible that this will not be a problem if the kernel can preserve reasonably large chunks of phsyically contiguous memory.

There are a few other things that have to be done. To name a few: Including more statistics in the `/proc` interface to reflect the state of the system, considering the different kind of cores in it; adding an interface to let the user code (or the kernel) request that a process should continue in the TC node (the transfer has been implemented and tested, but there is no interface yet to trigger it); deciding which interface to adopt for the user to call the new service, and to what extent the mechanism has to be its own policy; implementing KCs (which should not require any code, because they must execute the standard kernel); testing and debugging note handling for AC processes; more testing and fine tuning.

4. Things already done

This section is more a quick log of the current state of the implementation. As of today, the kernel is operational, although not in production. More work is needed in system interfaces, role changing, and memory management; but the kernel is functional enough to be used, at least for testing.

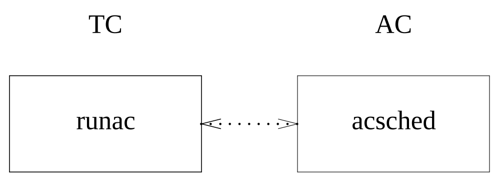
We have changed a surprisingly small amount of code at this point. There are about 400 lines of new assembler source, about 80 lines of platform independent C source, and about 350 lines of AMD64 C source code. To this, we have to add a few extra source lines in the start-up code, system call, and trap handlers. This implementation is being both developed and tested only in the AMD64 architecture.

A new system call

```
execac(int core, char *path, char *args[])
```

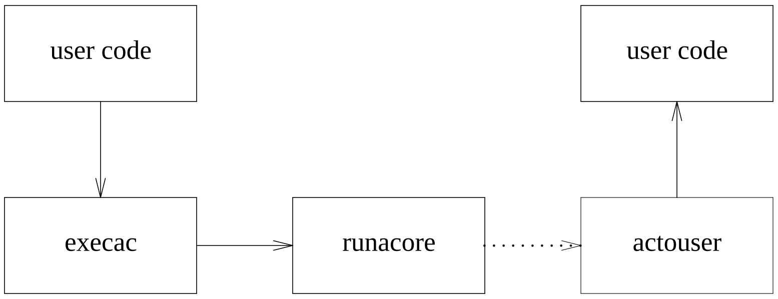
similar to *exec* is the primary interface. If core is 0, the standard *exec* system call is performed, except that all pages are faulted in before the process starts. We found that there seems to be no performance penalty for pre-paging, which is interesting on its own. If core is non-zero, and it is available, the process is dispatched for execution at that AC. We do this by using the inter core call, active message, mechanism.

The figure shows the inter-core-call or ICC mechanism. The function *acsched* runs on ACs, as part of its start-up sequence. It sits in a tight loop spinning on an active message function pointer. To ask the AC to execute a function, the TC sets in the ICC structure for the AC the arguments, indication to flush the TLB or not, and the function pointer and then changes the process state to *Exotic*, which would block the process for the moment. When the pointer becomes non-nil, the AC calls the function. The function pointer uses a cache line and all other arguments use a different cache line, so that polling can be efficient regarding bus transactions. Once the function is done, the AC sets the function pointer to nil and calls *ready* on the process that scheduled the function for execution. You think of this as a soft IPI.



While an AC is performing an action dictated by a process in the TC, its *Mach* structure points to the process so that *up* in the AC refers to the process. The process refers to the AC via a new field *Mach.ac*. Should the AC become idle, its *Mach.proc* field is set to nil.

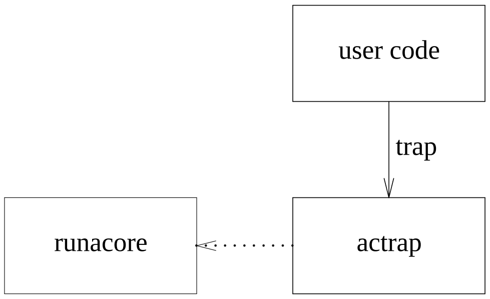
This mechanism is kind of like a vm_exit(). While we could think about implementing this with vm_enter(), it’s not quite clear we should take that step. VMs have overhead, at least when compared to the mechanism we use. This mechanism is used by NIX to dispatch processes to ACs, as shown:



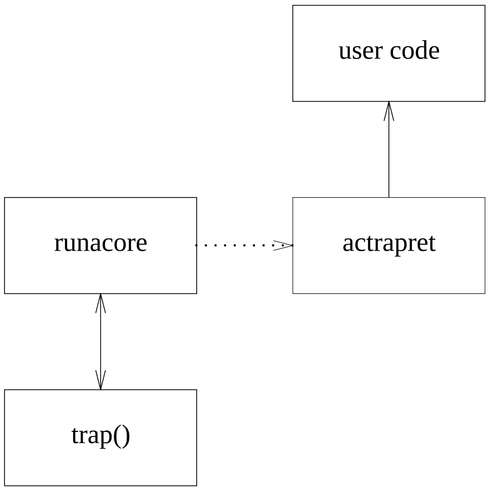
A process calls the new system call, *execac* and that makes the kernel call *runacore* in the context of the process. This function makes the process become a handler for the actual process, which would be running from now on on the AC. To do so, *runacore* calls *runac* to execute *actouser* on the AP selected. This transfers control to user-mode, restoring the state as saved in the *Ureg* kept by the process in its kernel stack. Both *actouser* and any kernel handler executing in the AC runs using the *Mach* stack.

The user code runs undisturbed in the AC while the (handler) process is blocked, waiting for the ICC to complete. That happens as soon as there is a fault or a system call in the AC.

When an AP takes a fault, the AP transfers control of the process back to the TC, by finishing the ICC, and then waits for directions. That is, the AP spins on the ICC structure waiting for a new call.



The handling process, running *runacore*, handles the fault and issues a new ICC to make the AP return from the trap, so that the process continues execution in its core. The trap might kill the process, in which case the AC is released and becomes idle.



Handling page faults requires the handling process to get access to the *cr2* register as found in the AC. We have virtualized the register. The TC saves the hardware register into a software copy, kept in the *Mach* structure. The AC does the same. The function *runacore* updates the TC software cr2 with the one found in the AC before calling *trap*, so that trap handling code does not need to know which core caused the fault.

When an AP makes a system call, the kernel handler in the AP returns control back to the TC in the same way it is done for faults, by completing the ICC. The handler process in the TC serves the system call and then transfers control back to the AC, by issuing a new ICC to let the process continue its execution after returning to user mode. Like in traps, the user context is kept in the *Ureg* in the handler process kernel stack, as it is done for all other processes.

The handler process, that is, the original time-sharing process when executing *runacore*, behaves like the red line separating the user code (now in the AC) and the kernel code (run in the TC). It is feasible to bring the process back to the TC, as it was before calling *execac*. To do so, *runacore* returns and, only this case, both *execac* and *syscall* are careful not do anything but returning to the caller. The reason is that calling *runacore* was like returning from *exec* to user code (only that in a different core). All things done while returning, were already done by *runacore*. Also, because the *Ureg* in the bottom of the kernel stack for the process has been used as the place to keep the user context, the code executed after returning from *syscall* would restore the user context as it was when the process left the AC to go back to the TC.

Although this mechanism is working, there is no interface as of now to reclaim a process back to the TC.

Interrupts are all routed to BSP. ACs should not take any interrupts, as they cause jitter. We changed the round-robin allocation code to find the first core able to take interrupts and route all to that. We assume that first core is the BSP. Also, no APIC timer interrupts are enabled on ACs. User code in ACs runs undisturbed, until it faults or makes a system call.

The AC requires a few new assembler routines to transfer control to/from user space, while using the standard *Ureg* space in the bottom of the process kernel stack for saving and restoring process context. The process kernel stack is not used (but for keeping the *Ureg*) in the ACs; instead, the *Mach* stack is used for the few kernel routines needed.

Because the instructions used to enter the kernel, and the sequence, is exactly the same in both the TC and the AC, no change is needed in the C library (other than a new system for using the new service). All system calls may proceed, transparently for the user, in both kinds of cores.

5. Conclusions

NIX presents a new model for operating systems on manycores. It follows a heterogeneous multicore model, which is the anticipated model for future systems. It uses active messages for inter-core control. It also preserves backwards compatibility. Programs will always work.

6. References (for goodness sake let's use refer!)

[1] ISC09 paper

[2] Sc11 paper

[3] KItten paper