Design and Implementation for the CiNIC Device Driver v2.0

A Senior Project Report Presented to the Computer Engineering Program

By

Max Roth

California Polytechnic State University, San Luis Obispo

Date Submitted: June 14, 2002
Advisor: Phil Nico
Abstract

The goal of the CiNIC project is to develop an intelligent network card that can offload the TCP/IP stack to provide better performance and increased functionality. We envision a network card that can support built-in firewalls, network caching, encryption services and intrusion detection. In this paper, I explain the design and implementation of the second generation CiNIC device driver. The new driver includes improvements to the protocol layer, shared memory segment, and thread system on the coprocessor.
Acknowledgements

I’d like to thank the faculty on the 3Com project: Jim Harris, Phillip Nico, and Hugh Smith. Without them, I’d probably have a lot more free time and a date with that girl in my Bio class. Boy, was she cuteeee! But in all seriousness, I am lucky to have had such wonderful people for encouragement and support. Their friendship and wise-cracks will be sadly missed.

I’d also like to give a shout out to the most mo-digidist-rock’n development team ever to hit the 3Com project: Billy Kwek on the kernel team, Scott-Ben-Billy-Joel-oh-whatever-your-name-is Melara, Jason-never-test-a-piece-of-code-before-committing-it-to-CVS Hatashita, Heather-heavens-max-is-a-sweet-coder-and-chicks-dig-him-too! Heiman, Clif-“where-do-you-want-to-go-today?”© Gordon, Daniel-sole-user-and-only-person-to-ever-download-the-ISO-for-Redmond-Linux Miller, Edison-Edison!-EDISON!-wake-up!-wake-up! David, and finally Eli-first-to-find-a-reverse-polish-notation-calculator-on-an-HP-logic-analyzer Snell. You guys have a special place in my heart. Thanks for all the support and the late night debugging festivals.

And finally, who can forget the very reason why I get up in the morning (err, early afternoon). I want to thank my Lord Jesus for giving me all these blessings. I stand amazed by His continued love and grace. May my life and works be a pleasing sacrifice.
# Table of Contents

Abstract .................................................................................................................................................................ii
Acknowledgements ...................................................................................................................................................iii
Introduction ...............................................................................................................................................................1
  Background and Problem Statement ..................................................................................................................1
  Hardware/Software Architecture .......................................................................................................................2
  Software Architecture ..........................................................................................................................................3
Second Generation Design .....................................................................................................................................6
  Overview ...............................................................................................................................................................6
  Detailed Flow Control .........................................................................................................................................8
Components ............................................................................................................................................................10
  Shared Memory ....................................................................................................................................................10
  Manager and Worker Threads ............................................................................................................................13
  Circular Buffer ..................................................................................................................................................14
  Interrupt Handler ...............................................................................................................................................15
  Proc File System ...............................................................................................................................................16
Conclusion .............................................................................................................................................................16
Future Work .........................................................................................................................................................17
Work Cited .............................................................................................................................................................18
Appendix A – Cache Coherency ...........................................................................................................................19
Appendix B – Source Code ......................................................................................................................................20

## Host Side

- bitmap.h .........................................................................................................................................................21
- bitmap.c ............................................................................................................................................................23
- com.h ...............................................................................................................................................................28
- com.c ...............................................................................................................................................................33
- proc_host_bitmap.h .........................................................................................................................................28
- proc_host_bitmap.c .........................................................................................................................................29

## Cohost Side

- cbuf.h ..............................................................................................................................................................38
- cbuf.c ..............................................................................................................................................................39
- thread.h ...........................................................................................................................................................40
- thread.c ...........................................................................................................................................................41
- com_e.h ............................................................................................................................................................46
- com_e.c ............................................................................................................................................................47
- manager_worker.h ..........................................................................................................................................51
- manager_worker.c ..........................................................................................................................................52
Introduction

Background and Problem Statement

With the proliferation of the Internet came a revolution in how people interacted with their computers. The general public, who once envisioned computers as a mere work tool that only added grief to their already melancholy lives, began seeing the computer as a vehicle for social interaction. Whole communities, by way of the “net”, began forming around common interests and hobbies. Before long, the computer was transformed from its previous subordinate role to being, arguably, the very center of modern society.

With this success came the frenzied development of applications to exploit “web” connectivity. Manufacturers and entrepreneurs alike scrambled to build the next “killer” application, or for that matter, anything that used the Internet. Everything from virus checkers to interactive first-person-shooter games began using the Internet to improve their functionality. Even the OS itself, began auto-updating itself over the net. All this, of course, demanded faster computers to keep up with the complex processing involved with networked applications.

In order to accommodate this increased demand for speed and efficiency in networking, the 3Com project has developed an intelligent network interface that is capable of performing all the TCP/IP processing itself. The CiNIC, CalPoly Intelligent Network Interface Card, uses hooks in the Linux kernel to move socket related system calls over to the NIC via a custom driver. In doing so, we reduce the processing for the PC and add functionality like built-in firewalls, encryption, and intrusion detection.

In winter of 2001, I joined the CiNIC project to continue the development of the CiNIC device driver that facilitated the offloading of the TCP/IP stack from a host PC. I was fortunate that the device driver was already in late development and there was already a working prototype. The hardware platform at this point, had already been standardized.
**Hardware/Software Architecture**

The three main components in our development environment are the host PC, the EBSA285, and the 21554 non-transparent bridge. The host PC is a garden variety Dell PC with an Intel Pentium III and 128 megabytes of SDRAM. RedHat Linux, version 7.1, is installed with the standard kernel configuration and set of application. The RedHat environment is purposely vanilla.

The cohost is an EBSA285 single board computer running a port of the Linux operating system. The EBSA285 is equipped with an Intel StrongArm processor at 233Mhz, an integrated MMU, 128 Megabytes of SDRAM, and a PCI bridge. The Linux kernel port, developed by Russell King, is currently at version 2.4.3 and is patched to support reserving 4 megabytes of physical memory at boot time.

In order to be able to insert the EBSA285 into the host PC, a 21554 Non-transparent PCI-to-PCI bridge is required. The bridge separates the EBSA285’s PCI bus from that of the host’s and allows a segment of memory to be mapped across both. The shared mapping of physical memory is used for inter-process communication between the two sides. The 21554 also provides a passive PCI
backplane for the EBSA285 board so that it can support a NIC. The EBSA285, 21554 bridge, and NIC make up the cohost. Figure 1 shows the host and the cohost platforms.

**Software Architecture**

The CiNIC driver is divided into two kernel modules. The first module, developed by Mark McClelland, is responsible for registering the driver with the Linux kernel and setting up the shared memory segment [1]. The shared memory is mapped by both the host and the cohost and is used to move data between the two platforms. The module also configures the 21554 bridge to send interrupts from either the host or the cohost side. Only after this module has successfully loaded, can the second module can be installed.

The second module, developed by Rob McCready, intercepts network related system calls and implements a protocol to transport them to and from the cohost. In order for the host’s TCP/IP stack to be completely bypassed, the module intercepts the network related system calls as soon as they enter kernel space. Each system call is then checked to see if it is network related. If so, the system call parameters are marshalled into a contiguous piece of memory (referred to in the rest of this document as a *packet*) and passed to the protocol layer of the CiNIC driver. The protocol layer handles moving the data into shared memory and notifying the cohost that there is a packet ready for service.

The first implementation of the protocol layer, as described in [2], used a two megabyte shared memory segment to transfer data between the host and cohost. In order for data to move in both directions simultaneously, the shared memory was divided into an upstream and a downstream segment. The upstream segment was used for moving data from the host to the cohost while, conversely, the downstream segment moved data from the cohost to the host. Each segment also contained a shared mutex that was used to signal either side when there was data present in shared memory. Controlling the movement of data in each direction was a system of kernel threads.
As depicted in figure 2, each memory segment was serviced by a kernel thread on either end. These threads were responsible for ensuring that only one process had access to the shared segment at any given time. When there were multiple processes that needed to transfer system calls to the cohost, the system call data was placed in a FIFO queue. The to_ebsa thread would then service the queue and move a packet into memory when it found that the upstream memory segment was available. The from_ebsa thread would then set the mutex to signal the from_host thread on the cohost side.

The from_host thread on the cohost, all the while, had been polling and saw the mutex change. It then responded by copying the packet to a working buffer and clearing the mutex. This event signaled the host to begin the process all over again. Likewise, the return path from the cohost to the host was identical except for using a different thread pair and the downstream shared memory segment.

This approach suffered from serious drawbacks. First, the shared memory layout only allowed for one packet to be transmitted at a time. In order to make the best use of the available memory, each side had to immediately copy the packet out of shared memory so that the next packet can be transmitted. This copying was processor intensive, especially on the host side where it had to traverse through the PCI bus and the 21554 bridge. Furthermore, in order to protect shared
memory from race conditions associated with multiple processes trying to write to shared memory, a complex system of locks and queues was required to enforce ordering. This added a great deal of overhead and caused a decrease in performance.

Second, the use of a binary mutex for signaling required each side to continuously poll. This meant that each thread had to repeatedly reschedule itself waiting for the mutex value to change. Each invocation of the scheduler meant that the kernel had to impede the thread’s processing, which led to increased latencies and CPU loads.
Second Generation Design

Overview

The new protocol layer remedies some of the inefficiencies associated with the previous implementation. First, it improves efficiency by replacing a polling protocol with one that uses interrupts for signaling. Second, the shared memory segment, used for communication between the PC and the CiNIC, is now compartmentalized to simultaneously handle multiple packets. Third, a thread pool system on the cohost side improves performance and better protects the CiNIC system from locking up.

Figure 3 gives a high level overview of the device driver. Unlike the previous version, there are no handler threads or multiple sleep states. We were able to remove these because the shared memory segment is divided into blocks that can be individually reserved through a bitmap API. The new memory layout allows for each process to access its own section of shared memory without worrying about interference from others. Thus, when a user-space application traps into kernel space by way of a network related system call, the CiNIC driver is able to execute without

Figure 3: State diagram for a typical system call on the host side.
interruption until it is ready to pass control to the cohost. Only after firing an interrupt to signal the cohost, will the process sleep.

The introduction of interrupts also alleviates the performance penalties incurred by using a shared mutex for signaling. The host and the cohost can now signal one another through interrupts without requiring a kernel thread to continuously poll the mutex. An interrupt handler is all that is required to service the interrupts on both sides. As depicted in figure 4, the top half on the cohost side obtains the packet from memory and dispatches it to the underlying thread system.

The thread system on the cohost side is used to ensure that blocking system calls cannot interfere with other system calls. The main feature in the system is the spawning of a worker thread for every open socket on the host. In other words, every socket created on the host has a corresponding worker thread on the cohost. It is the responsibility of the worker thread to unmarshall the system call parameters from shared memory, populate the requested system call, and execute it. At this point, control is passed to the cohost’s TCP/IP stack for processing. When the system call returns,
the thread will signal the host that the packet is ready with an interrupt. The host now responds to
the interrupt and wakes the appropriate sleeping process. After a releasing the shared memory slot
and copying the appropriate data back to the user, the process returns to user space.

**Detailed Flow Control**

This section will trace the flow control of a typical network related system call in order to better
understand the inner workings of the CiNIC driver. As mentioned previously, the driver is invoked
when a user space process makes a system call from the socket API family. When this event
occurs, known as a software interrupt, the kernel responds by invoking n_sys_socketcall().
Depending on the CALL value passed to it by the kernel, n_sys_socketcall() will execute the
appropriate handling routine, thereby serving a dispatch role. Since we want to bypass the kernel's
own methods, we call our own functions that will offload the packet onto the cohost device.
Functions that are handled by this method are:

```
SYS_SOCKET
SYS_CONNECT
SYS_BIND
SYS_LISTEN
SYS_SEND
SYS_ACCEPT
SYS_RECV
SYS_SENDTO
SYS_RECVFROM
SYS_GETSOCKNAME
SYS_SETSOCKOPT
```

It should be noted that not all functions typically used in socket programming are represented here.
This is because n_sys_socketcall() only handles device independent socket calls. For methods that
are device specific, like read() and write(), the kernel uses specific entries in the system call table to
handle them. Therefore, the CiNIC device driver also must redirect these entries and determine
whether the call being processed is destined for the cohost stack. The criterion is simple, if the file
descriptor being requested is mapped to a network socket, it will be processed by the CiNIC driver.
Otherwise, the driver will simply redirect it to the local kernel's own handling routines. [2]
Assuming the socket value is destined for the cohost, the device driver will first try to reserve a shared memory slot by calling the get_shrmem_slot(). This method is responsible for querying the bitmap structures to find consecutive slots large enough to hold all the system call parameters. Should a slot be found, the bitmaps will be updated to lock the memory from other processes, and a pointer to that block in shared memory will be returned. If a slot cannot be acquired, get_shrmem_slot() will schedule itself and try repeatedly up to a predefined limit. If at this point, it is still unable to obtain a shared memory slot, the protocol layer will return back to user-space with an appropriate error value.

From here, assuming that we obtained a slot, the device driver begins marshalling the system call parameters into the shared memory. This process can be arduous because of the complicated nature of socket parameters. Some functions, like recvmsg, are notorious for complex structures with scatter-gather type pointer structures. Therefore, these functions require a lot of overhead to arrive at a contiguous piece of memory (a packet) that can be transferred to the cohost.

After the packet is in shared memory, the host must update the shared memory header with the beginning block number of the packet in memory. Later, this will be used by the cohost to determine the offset of the packet in shared memory. The host then calls interrupt_cohost() to signal the cohost that there is a packet ready. In order to avoid data corruption and losing interrupts, this requires the host to ensure that the cohost is not in the middle of servicing an interrupt from a previous packet. Thus, iPendingInterrupt() is used to check the status bit on the 21554 bridge. The status bit gets cleared after an interrupt is serviced. Finally, the kernel process handling this packet will sleep while waiting for the cohost to process the packet.

When the interrupt arrives from the host, the Linux kernel executes the interrupt_handler(). This function is responsible for finding the packet in shared memory and passing the packet to the appropriate handling thread. If the packet is of type SOCKET or CLOSE, it will be dispatched to the manager thread for processing. The manager thread will make create the socket and spawn the worker thread that will handle packets for the new socket descriptor. The manager thread is also responsible for tearing down the thread and closing the file descriptor when a CLOSE arrives. All other packet types are sorted according to the socket descriptor and pushed to their respective
worker threads. Each network socket created on the host will be assigned a dedicated worker thread.

At this point, we can assume that the packet sent by the host has been passed to the corresponding worker thread. The worker thread then unpacks the system call parameters and uses them to make the appropriate socket API call. It is at this step that control is passed to the cohost's TCP/IP stack. When the socket API call returns, we call send_to_host() to signal the host that the cohost has finished and that the data is ready to return to the application layer. The procedure is very similar to that of the host, in that the shared memory is updated and an interrupt is fired to the host.

Back at the host, the kernel calls interrupt_handler() in response to receiving the interrupt from the cohost. This function finds the packet in shared memory and wakes the process that had been put to sleep previously. The awakened process now copies the relevant data back to user space, releases the bitmap slot that had previously been allotted, and returns to user space.

**Components**

In the following sections, we will examine the implementation of the various components that make up the CiNIC driver.

**Shared Memory**

The shared memory region is compartmentalized in 256 blocks. As shown in figure 5, there are two types of blocks: 128 byte small blocks and 8192 byte large blocks. The small blocks are allotted to system calls that require little space like bind(), connect(), and accept(). System calls with large payloads, like read() and write(), are typically handled with big blocks. The task of designating and reserving blocks is handled by the bitmap API that is described in the next section. Finally, the header structures that are located in the beginning of shared memory are used to tell the cohost where the packet resides in memory.
In order for shared memory to be simultaneously used by many processes, there has to be an arbiter to avoid collisions and data corruption. The method I implemented uses two bitmaps, one for the small blocks and the other for the large blocks, to represent shared memory. Each block in shared memory is represented with a bit in a bitmap structure shown below. If a block is in use by a process, the bit mapping that block is set to 1. Likewise, a free block is denoted with a 0.

```c
/* Bitmap Struct type*/
typedef struct{
    short int next_avail;
    unsigned long bits [SET_LONGS];
    unsigned short track_blocks_needed[BITMAP_SIZE];
}bitmap_t;
```

In order for a process to reserve a block, it must access the bitmap through the bitmap API. The available functions are get_free_block() and release_blocks(). get_free_block() is used to query the bitmap for a shared memory slot of size \( pkt_len \). In order to do this, the function must first determine which block size is best suited for storing the packet. The only criterion for this
The selection process is the packet length. If the packet length is smaller than 512 bytes, it is designated as a small block. Likewise, if the length is over 512 bytes, it is slated for large blocks. Once the block size is determined, the next step is to compute how many blocks are required to fit the packet. This is performed by dividing the block length by the block size and adding 1 to the result. The final step is to access the corresponding bitmap structure and begin searching.

The search algorithm is a simple linear traversal starting from the next_avail location up to the end of the bitmap. The next_avail value is always assigned to the bit immediately following the most recent slot allocation. This way, the likelihood of finding a free block is greater than if the traversal always began at the beginning of the bitmap. If the search arrives at the end of the bitmap and does not find a slot, it will begin from the beginning and traverse up to the next_avail location. If after this point the algorithm does not find a slot, it will return with an error code.

Should a slot be found, the next task is to reserve this slot by setting the appropriate bits to 1. To do this, an internal function call __mark_blocks() is used. This function invokes some assembly routines to quickly mark the specified bits. Finally, because get_free_slot() only returns the beginning block number (referred to as start_block) of the reserved slot, we need some sort of mechanism to record how many bits are associated with the reserved slot so we can later release them. We do this by storing the number of bits required for the slot in an array that is indexed with the beginning block number.
When the transaction with the cohost is done and the slot needs to be freed, the host will call release_block(). This function first retrieves the number of bits associated with the start block number. With this information, it then clears the appropriate number of bits which frees up the memory for other processes.

Manager and Worker Threads

The manager thread is responsible for controlling the various worker threads. Each time a user on the host requests a new socket with the SOCKET system call, the manager thread spawns a worker thread to handle all future requests made to that socket descriptor. This way we ensure that a blocked socket API call made to one socket does not inhibit any other socket calls. The manager thread is also responsible for handling CLOSE system calls by tearing down the worker thread associated with the request.

The manager thread is initialized when the CiNIC device is initially loaded. In order to provide it a “sane” environment to work from, one that does not inherit properties from the ins_mod process, the keventd daemon is used to launch the thread. To do so, we only need to register our start_manager() function with the kerneld daemon and call the scheduler. It should be noted that this step can take a considerable amount of time because we have to wait for the kernel to schedule keventd. Furthermore, because keventd is shared by many processes, we also have to wait for our turn to run on keventd’s run queue. This delay is only present when the manager spawns and thus only impacts the driver during the initialization. As a result, the driver is inoperable during this period.

After the manager thread gets initialized, the thread will sleep waiting for the interrupt handler to deliver a packet. When the manager thread receives a SOCKET packet, it handles the packet by first calling n_sys_socket() to create a network socket on the cohost platform. The socket descriptor value for this new socket is then used to index into a special array (referred to as thread_lookup[]) that contains pointers to structures that hold the worker thread’s state information. In this case, however, the thread structure pointer indexed by the socket descriptor value is NULL because the worker thread servicing the new socket descriptor does not yet exist. The manager
now allocates the thread structure and assigns the pointer to it. Finally, it calls
spawn_worker_thread() function which spawns the worker thread.

The spawn_worker_thread() function differs from the from start_manager() in that it does not use
the keventd daemon to spawn the thread. Instead, the function spawns the thread directly from the
manager thread. This ensures that the thread gets spawned immediately and the manager’s
environment parameters are transferred to the new child thread. This works to our advantage
because we inherit the open file descriptors created by the manager thread. The worker can now
access the sockets created by the manager thread.

In the future, whenever a packet arrives with the this particular socket value, the interrupt handler
will index into the thread_lookup[] array and get the state information for the worker assigned to
that file descriptor. Included in the state information is, most importantly, the circular buffer that
the worker thread services. The top half only needs to add the packet to the thread’s circular buffer
and wake up the worker thread by setting the thread’s sleep semaphore. When the worker awakens,
it responds by calling cbuf_remove() to obtain the packet. The worker thread unmarshalls the
system call parameter, makes the appropriate socket call, and signals the host when it is done.

Whenever a CLOSE socket arrives from the host, the manager thread will tear down the worker
thread assigned to that socket descriptor. This is accomplished by first closing the socket and then
sending the SIGKILL signal to the worker. When the worker thread receives the signal, it will
deallocate itself and signal the when it is about to exit. The manager will then reap the child and
deallocate the memory from the thread_lookup[] array.

Circular Buffer

Circular buffers are used by the cohost to pass packets from the interrupt handler to the thread
processes. Each thread gets allotted one circular buffer containing 1024 packet pointers. The
circular buffer is purposely very large because data is added to the circular buffer during interrupts.
Because we are in interrupt context, the process is forbidden to sleep or poll should there be no
room available in the circular buffer. By making the circular buffer large, we ensure that such a
scenario will likely not occur. Although this solution might seem rather crude, this is the only conceivable solution I can come up with short of developing a complicated subsystem to dynamically grow the arrays.

The circular buffer system is accessed through two functions: cbuf_add() and cbuf_remove(). The cbuf_add() function places a packet in the head position and increments head by one. The cbuf_remove() retrieves the packet pointed to by tail and increments tail. When tail+1 equals head, we know the thread is empty and an error should be returned if the user tries to remove a packet. Likewise, if head equals tail, we know that the circular buffer is full and we return an error should the user try to add a packet. As mentioned previously, this latter event would be catastrophic because there is no recourse for the interrupt handler to take. The packet would be inaccessible and freeze the user space application indefinitely. Short of unloading the modules on each side, there is no recovery.

Interrupt Handler

The interrupt handler is responsible for servicing interrupts. When the interrupt line attached to the 21554 PCI bridge is asserted, the Linux kernel responds by invoking do_irq(). Among its many duties, do_irq() will call the function registered to that interrupt. In our driver, the function registered to service the interrupt is actually just a wrapper that invokes, by way of an exported function pointer, the interrupt_handler() routine defined at the protocol layer. This function actually services the packet in shared memory.

It is important to realize that interrupt_handler() runs as a top half (a.k.a. fast interrupt handler) and expects only one packet. If interrupts are fired too quickly, thereby creating a scenario where the interrupt line is asserted despite it already being high, then the interrupt will be lost and the next top half will not be able to recover the packet. In order to guard against this possibility, iPendingCoHostINT() and iPendingHostINT() are used to verify that the interrupt pending bit on the 21554 bridge is, in fact, cleared. The interrupt pending bit is set by firing an interrupt but only gets reset after the top half has completely finished and is ready for another interrupt. If there is an
interrupt still pending, the protocol layer will respond by rescheduling and trying again at a later time.

**Proc File System**

Included in “path = /proc/cinic/host/” are two files names print_small_bitmap and print_large_bitmap. These files get created dynamically by the CiNiC modules and serve as a visual representation of the current state of the bitmaps. They also display basic information like percentage of blocks free. Furthermore, if the BITMAP_PARANOID flag is set in bitmap.h.h, this system will present information about collisions and number of times a process had to sleep when waiting for a free slot. In the case where there is only one packet in memory, the system will fetch it and print out some relevant information like function id, type and process id.

**Conclusion**

In this paper, I have explained the design and implementation of the second-generation CiNIC driver. Building from Rob McCready’s original version, various improvements were made to facilitate more efficient packet handling. We have examined how compartmentalizing the shared memory segment and adding interrupts remedied the bottleneck associated with the old polling protocol. A new method using worker threads on the cohost side was also introduced to alleviate the problem of having blocking system calls interfere with other processes.

Our preliminary performance measurement shows that the new architecture still performs inadequately when compared to a stand-alone machine. Both the host and cohost can independently download a 100Mb file in nearly 38.4 seconds (2.7e+03 kbytes/sec). When the CiNIC driver is loaded and the cohost offloads the TCP/IP stack, the performance decreases to 83.9 seconds (1.2e+03 Kbytes/sec). Although this is discouraging, I have great hope that we can further optimize the driver to provide nearly comparable performance.
**Future Work**

The main problem that needs to be resolved is how to recover from an application that does not properly close its file descriptors. When this event occurs, the kernel has to manually do all the clean up work. This clean up, as far as I can tell, occurs deep within the kernel and therefore is not visible to the cohost. As a result, the cohost is left in an unstable state. In order to solve this problem, we need to develop some kind of signaling mechanism to force the cohost to kill whatever threads were allocated for that process.

Once this bug is fixed, I suggest developing an automated test suite to run the code against whenever changes are introduced. A simple script that launches a fleet of threads that intelligently test the functionality of the device driver would suffice. Although testing by running Netscape and X Windows is encouraged, I feel that it is not sufficient in and of itself. This kind of just-see-if-it-works testing is not robust enough to identify all the difficult bugs early on. Furthermore, even if you do find a problem, debugging with these products is difficult because of their complexity and network socket abusing nature. Having a lot of simple tests that test for specific conditions will save a lot of heartache in the future.

I would also suggest that we rewrite the fd_map family of functions. These functions, as described in [2], are used to translate the file descriptor values on the host to those of the cohost. The main problem with the current implementation is that searching a large linked list is expensive. I suggest we use a hash table instead. I believe that this will greatly improve the speed of the driver because the linked list is traversed at least twice for every system call. Furthermore, functions like select() have to search the linked list numerous times per invocation. For example, if I call select once on 4 file descriptors, the linked list of translations will be searched a minimum of 8 times! A simple hash system would reduce the overhead associated in searching the list.

Finally, we should begin porting the source code over to the next generation architecture. As soon as the hardware platform is defined and the OS selected, the process of simulating the new environment should begin. Most likely, the new board will not be able to run a full-blown Linux distribution, so this will require some redesign.
Work Cited


Appendix A – Cache Coherency

The biggest problem I had when implementing the new protocol was ensuring cache consistency on the cohost side. Unlike the host side where the shared memory segment is mapped as I/O and thereby guarantees that all transactions are write-through, the cohost shared memory segment is mapped as regular system memory and cached. This leads to a visibility problem because the StrongArm will fetch and write data from cache while the 21554 bridge accesses physical memory directly. When the host tries to read a value from memory, it will retrieve an old stale value from memory instead the updated value in cache.

The most elegant fix would be to disable caching for the shared memory segment. As of this writing, there is no method for doing this on the EBSA285 platform. Russell King, the developer who ported Linux to the StrongArm processor, has disabled this feature on all memory mapping functions (see for example ioremap_nocache). However, Russell has kindly implemented a family of functions that can invalidate the cache lines themselves. Using these functions, I was able to resolve the problem in a satisfactory manner.

The solution has three parts. First, we clean and invalidate the cache lines at the beginning of the top half interrupt handler. This ensures that any data in cache is written back before our interrupt handler code reads values from shared memory. We also flush the cache before exiting the top half to ensure that any writes get posted immediately. The second part of this solution involves ensuring that the host does not try to write another packet number to the header value before the cohost has an opportunity to run the top half handler. Jason Hatashita and I implemented iPendingHostINT() and iPendingCoHostINT() to ensure that the host waits for the cohost to finish handling the interrupt before sending another interrupt. The third modification involved placing buffer regions around data that could be simultaneously accessed by both side. We had to implement this because the cohost’s CPU fills 16byte cache lines when reading data from memory. Thus, the cohost can corrupt data that is immediately adjacent to the data it is using. By putting spaces in shared memory, we remove this possibility.
Appendix B – Source Code

This section contains the source code for the major portions of the new architecture. For the entire CiNIC driver source, please see the project web site.
Host Side

bitmap_h.h

/*
/* bitmap_h.h- Cal Poly 3Com CiNIC project
/*
/* Provides bitmap constants and function defn. for
/* bitmap manipulation routines.
/*
/* 5Id: bitmap_h.h,v 1.0 -Max
/*
*/
 ifndef _BITMAP_H_
define _BITMAP_H_
#include "global.h"
#include "com.h"
#include "com_h.h"

/* If _BITMAP_PARANOID is defined, the bitmap routines will do */
/* simple bounds and collision reporting. This will add a few */
/* instructions per call to the bit maps. Better to disable */
/* this when you are sure everything is working properly */
define BITMAP_PARANOID

/* Seconds that the process will wait before trying to allocate */
/* free blocks */
/*
*/
define BITMAP_NO_FREE_BLOCK_DELAY (1/100)

/* Are you looking for these:
* SMALL_BITMAP
* LARGE_BITMAP
* SMALL_BLOCK_SIZE
* LARGE_BLOCK_SIZE
* see global.h
* although they pertain to bitmaps, they are
* needed by both the host and cohost.
*/

/* Seeing as the return value is not signed, we need a simple
* way for returning errors. I'll just make the error an
* extremely large value that is totally unreasonable and will
* never occur (knock on wood, eh?)
*/
define BITMAP_ERROR 30000

define BITMAP_SIZE 20 /*Number of bits in each bitmap*/
define BITS (8 * sizeof(unsigned long))
define SET_LONGS (BITMAP_SIZE/BITS) + 1

/* The following assembly routines were originally taken from */
/* include/asm-i386/posix_types.h. I made some modifications */
/* to handle my expanded bitmap_t struct. */
/*
*/
/* History: */
/* July 27, 2001 - Version 1.0 , Max CiNiC project */
/* 04/09/02 - TODO: Turns out there are */
/* architecture independant forms of these */
/* functions. I'll leave implementing them */
/* as an exercise to the user. See device */
/* driver book. */
define BITMAP_CLR(bit,set) 
  __asm__ __volatile__("btrl %1,%0":
    "=m" (*((bitmap_t*) (set->bits))):"r" {{int} (bit)})
define BITMAP_ISSET(bit,set) {{

unsigned char __result; \
__asm__ volatile("btl %1,%2 ; setb %0" \
:"=q" (__result) :"r" ((int) (bit)), \
"m" (*(bitmap_t*) (set->bits))); \
__result; ))

#define BITMAP_SET(bit,set) \
__asm__ volatile("btsl %1,%0": \
:"=m" (*(bitmap_t *) (set->bits)):"r" ((int) (bit))

#define BITMAP_ZERO(set) \
do {\
    int __d0, __d1; \
    __asm__ volatile("cld ; rep ; stosl": \
    :="m" (*(bitmap_t *) (set->bits)), \
    "=c" (__d0), "=d" (__d1) \
    :"a" (0), "l" (_SET_LONGS), \n    "2" ((bitmap_t *) (set->bits)) : "memory"; \
} while (0)

/* Bitmap Struct type*/
typedef struct{
    short int next_avail;
    unsigned long bits [SET_LONGS];
    unsigned short track_blocks_needed[BITMAP_SIZE];
}bitmap_t;
#define BITMAPSTRUCT_SIZE sizeof(bitmap_t)

/* Bitmap Struct type to keep track of collisions and missed 
* attempts at allocating blocks (eg. Bitmap is full) */
/* Only is used when BITMAP_PARANOID is defined. Debug purposes. */
/* Outputs to proc filesystem */
typedef struct{
    unsigned long collision_count;
    unsigned long sleep_count; /* Tracks how many times process' sleeps */
    unsigned long massive_packet_count;
}bitmap_error_stat;

bitmap_error_stat track_errors;

/* Bitmap manipulation functions */
cyclic_queue_t get_free_block(long pkt_len);
int release_blocks(compkt_t* pkt);
void bitmap_init();
void bitmap_cleanup();
void __debug_print_bitmap_info(bitmap_t* set, char* name, char* page);

#endif
bitmap_h.c
/*
 * bitmap_h.c (Cal Poly 3Com CiNIC project)
 *
 * Authors: Max
 *
 * Description:
 * Provides routines for manipulating bitmaps on the host side.
 *
 * History:
 * 04/10/02    Max Roth    Initial coding
 *
 * Version: $Id: bitmap_h.c,v 1.1.1.1 2002/06/13 08:28:38 max Exp $
 */

#include <linux/time.h>
#include <asm/semaphore.h>  /* up/down */
#include <linux/slab.h>     /* kmalloc */
#include "bitmap_h.h"

bitmap_t *small_bitmap;
bitmap_t *large_bitmap;

struct semaphore small_bitmap_lock;
struct semaphore large_bitmap_lock;

static cyclic_queue_t __mark_blocks(int bitmap_select, bitmap_t* bitmap, int bits_needed);

/* Description: Initializes bitmaps/error tracking/semaphores.
 * Notes: */
void bitmap_init()
{
    PRINT_DEBUG(3,"Bitmap Initializion\n");
    
    #ifdef _BITMAP_PARANOID
    track_errors.collision_count   = 0;
    track_errors.sleep_count          = 0;
    track_errors.massive_packet_count = 0;
    #endif

    small_bitmap = (bitmap_t*)kmalloc(BITMAP_STRUCT_SIZE, GFP_KERNEL);
    large_bitmap = (bitmap_t*)kmalloc(BITMAP_STRUCT_SIZE, GFP_KERNEL);

    /* Clear the bitmap arrays */
    small_bitmap->next_avail = 0;
    large_bitmap->next_avail = 0;
    memset(small_bitmap->bits, 0, SET_LONGS * sizeof(unsigned long));
    memset(small_bitmap->track_blocks_needed, 0, BITMAP_SIZE * sizeof(unsigned short));
    memset(large_bitmap->bits, 0, SET_LONGS * sizeof(unsigned long));
    memset(large_bitmap->track_blocks_needed, 0, BITMAP_SIZE * sizeof(unsigned short));

    /* Initialize Semaphores */
    sema_init(&small_bitmap_lock, 1);
    sema_init(&large_bitmap_lock, 1);
}

/*
 * Description: 
 * Notes: */
void bitmap_cleanup()
/* deallocate memory. */
kfree(small_bitmap);
kfree(large_bitmap);
small_bitmap = NULL;
large_bitmap = NULL;
}

/* Name: get_free_block
 * Description: This function searches a given bitmap for a user defined
 * number of sequential free blocks. If it finds them it will mark them
 * as used and return the beginning block number.
 * History:
 * July 27, 2001 - A tad ugly but this is the most efficient method of
 * searching the bit map that I can come up with.
 * Don't let any of the CSC professors see this! -Max
 * Note that speed is critical (critical section too!),
 * thus we return as soon as possible
 * +
 */
cyclic_queue_t get_free_block(long pkt_len){
    int bits_needed = 0;
    short temp_store = 0;
    int temp_found = 0;
    int bitmap_selected = LARGE_BITMAP;
    bitmap_t* bitmap;
    cyclic_queue_t start_block;
    struct semaphore *lock;

    if(pkt_len == 0){
        start_block.block_index = BITMAP_ERROR;
        return start_block;
    }

    /* TODO: This needs to be cleaned up. Use bitmasks
to speed the modulo*/
    /* Disregard case where pkt_size % 128 = 0 */
    if (pkt_len <= SMALL_BLOCK_SIZE){
        bits_needed = 1;
        bitmap = small_bitmap;
        bitmap_selected = SMALL_BITMAP;
    } else if (pkt_len <= (SMALL_BLOCK_SIZE * 4)){
        bits_needed = (pkt_len / SMALL_BLOCK_SIZE) + 1;
        bitmap = small_bitmap;
        bitmap_selected = SMALL_BITMAP;
    } else if (pkt_len % LARGE_BLOCK_SIZE){
        bits_needed = (pkt_len / LARGE_BLOCK_SIZE) + 1;
        bitmap = large_bitmap;
    } else {
        bits_needed = (pkt_len / LARGE_BLOCK_SIZE);
        bitmap = large_bitmap;
    }
    #ifdef _BITMAP_PARANOID
    if (bits_needed > 256){
        PRINT_DEBUG(3,"Bitmap error. Packet is enormous.\n");
        track_errors.massive_packet_count++;
        start_block.block_index = BITMAP_ERROR;
        return start_block;
    }
    #endif

    if (bitmap_selected == SMALL_BITMAP)
        lock = &small_bitmap_lock;
    else
lock = &large_bitmap_lock;
down(lock);
temp_store = bitmap->next_avail;

/* Search bitmap from next_avail pointer to end */
for(; bitmap->next_avail < BITMAP_SIZE ;){
  if (!BITMAP_ISSET(bitmap->next_avail, bitmap))
    temp_found++;
  else
    temp_found = 0;
  bitmap->next_avail++;

  if (temp_found == bits_needed){
    start_block = __mark_blocks(bitmap_selected, bitmap, bits_needed);
    bitmap->track_blocks_needed[start_block.block_index] = bits_needed;
    up(lock);
    return start_block;
  }
}

/* Search bitmap from beginning to next_avail pointer */
for(bitmap->next_avail = 0; bitmap->next_avail < temp_store;){
  if (!BITMAP_ISSET(bitmap->next_avail, bitmap))
    temp_found++;
  else
    temp_found = 0;
  bitmap->next_avail++;

  if (temp_found == bits_needed){
    start_block = __mark_blocks(bitmap_selected, bitmap, bits_needed);
    bitmap->track_blocks_needed[start_block.block_index] = bits_needed;
    up(lock);
    return start_block;
  }
}

/* Found nothing, return error code */
up(lock);
start_block.block_index = BITMAP_ERROR;
return start_block;

/* Name: release_blocks
 * Description: Used to quickly unmark bits.
 * Return: Number of blocks released or -1 for error.
 * July 27, 2001 - */
int release_blocks(compkt_t* pkt){
  int counter;
  short int num_of_blocks;

  bitmap_t* bitmap;
  struct semaphore *lock;

  if (pkt->start_block.block_index > BITMAP_SIZE){
    PRINT_DEBUG(1,"Bitmap Error! Block_index is > %d = actual index is %d\n", BITMAP_SIZE, pkt->start_block.block_index);
    return -1;
  }

  if (pkt->start_block.block_size == SMALL_BITMAP){
    bitmap = small_bitmap;
    lock = &small_bitmap_lock;
  }
} else {
    bitmap = large_bitmap;
    lock = &large_bitmap_lock;
}
}

down(lock);

num_of_blocks = bitmap->track_blocks_needed[pkt->start_block.block_index];

for(counter = 0; counter < num_of_blocks; counter++){
    #ifdef _BITMAP_PARANOID
    if (!BITMAP_ISSET(pkt->start_block.block_index + counter, bitmap)){
        PRINT_DEBUG(3, "CiNIC: Bitmap error! Bit expected to be set is clear.\n");
        track_errors.collision_count++;
        up(lock);
        return -1;
    }
    #endif

    BITMAP_CLR(pkt->start_block.block_index + counter, bitmap);
}

up(lock);
return counter;

/* Name: __mark_blocks  
* Description: Used to quickly mark bits. Returns cyclic_queue_t which will designate start_block number and the bitmap it belongs too. Remember, this gets called with a lock on the bitmaps, so we don't have to worry about critical sections/race cond.  
* Return: Returns shrmem_queue_t which will designate start_block number and the bitmap it belongs too.  
* July 27, 2001 - Internal function. -Max 
* August 5, 2001- cyclic_queue_t struct system has been implemented. 
*/

static cyclic_queue_t __mark_blocks(int bitmap_select, bitmap_t *bitmap, int bits_needed){

    int start_block;
    int counter;
    cyclic_queue_t return_block;

    start_block = (bitmap->next_avail - bits_needed);
    if(start_block < 0)
        start_block = 0;

    return_block.block_index = start_block;

    if (bitmap_select == SMALL_BITMAP)
        return_block.block_size = 0;
    else if (bitmap_select == LARGE_BITMAP)
        return_block.block_size = 1;
    else
        PRINT_DEBUG(1, "CiNIC: Error! Unspecified bitmap value passed to __mark_blocks\n");

    for(counter = 0; counter < bits_needed; counter++){
        #ifdef _BITMAP_PARANOID
        if (!BITMAP_ISSET(start_block + counter, bitmap)){
            PRINT_DEBUG(1, "CiNIC: Bitmap error! Collision. Bit is set.\n");
            PRINT_DEBUG(1, "---Questionable bit is: %d.\n", bitmap->next_avail);
            track_errors.collision_count++;
        }
        #endif
    
    return return_block;
}
#ifndef
BITMAP_SET(start_block + counter, bitmap);
#endif

return return_block;

/* Name: __debug_print_bitmap_info
 * Description: Internal function that will print out the bitmap.
 *              Useful for debug purposes.
 *              The percent figures do not work cause the logger
 *              has no float capabilities.
 * Return:      Void.
 */
void __debug_print_bitmap_info(bitmap_t* set, char* name, char* page){
    int counter           = 0;
    int total_iterations  = 0;
    int bits_set_stat     = 0;
    int bits_clear_stat   = 0;
    float percent_used;
    float percent_free;

    PRINT_DEBUG(3,"CiNIC: 

%s:
", name);
    for (counter=0; counter < 256; counter++){
        total_iterations++;
        if (!(counter%8))
            printk(" ");
        if (!(counter%32))
            printk("\n");
        if (BITMAP_ISSET(counter, set)){
            printk("1");
            bits_set_stat++;
        } else {
            printk("0");
            bits_clear_stat++;
        }
    }
    percent_used = ((float)bits_set_stat)/(float)BITMAP_SIZE * 100;
    percent_free = ((float)bits_clear_stat/(float)BITMAP_SIZE) * 100;
    printk("\n
Statistics:
");
    printk("^^^^^^^^^^^^^
");
    printk("Bitmap size:            %d
", BITMAP_SIZE);
    printk("Number of bits set:     %i
", bits_set_stat);
    printk("Number of bits cleared: %i
", bits_clear_stat);
    printk("Percentage used:        %.1f
", percent_used);
    printk("Percentage free:        %.1f
", percent_free);
    printk("Next available block: %i\n\n", set->next_avail);
    #ifdef _BITMAP_PARANOID
    printk("Collisions detected: %li\n", track_errors.collision_count);
    printk("Massive packets detected: %li\n", track_errors.massive_packet_count);
    printk("Sleeps required because bitmap was full: %li\n\n\n", track_errors.sleep_count);
    #endif
}
proc_host_bitmap.h

/*
 proc_host_bitmap.h - Cal Poly 3Com CiNIC project

 $Id: proc_host_bitmap.h,v 1.1.1.1 2002/06/13 08:28:39 max Exp $
 */

#ifndef _PROC_HOST_BITMAP_H_
#define _PROC_HOST_BITMAP_H_

#include "global.h"

int proc_host_bitmap_init(struct proc_dir_entry *root_entry);
int proc_host_bitmap_cleanup();

#endif
proc_host_bitmap.c

/*
 * proc_host_bitmap.c - Cal Poly 3Com CiNIC project
 *
 * Creates "bitmap status" file in proc system that
 * when read will graphically output the current state
 * of bitmap and include some basic statistical data.
 *
 * History:
 * 7/20/2001 - Initial coding - Max
 */

#include <linux/types.h>
#include <linux/module.h>
#include <linux/proc_fs.h>

#include "global.h"
#include "decode.h"
#include "syscalls.h"
#include "host.h"
#include "bitmap.h"
#include "fd_map.h"

extern fd_translation_table_t    *fd_tables;

/*
 * Error tracking structure
 */
#if defined BITMAP_PARANOID
extern bitmap_error_stat   track_errors;
#endif

/*
 * Bitmaps
 */
extern bitmap_t *small_bitmap, *large_bitmap;

/*
 * Proc functions and structures
 */

static int print_bitmap_info(bitmap_t* set, char* name, char* page);
static int read_small_bitmap_status(char *page, char **start, off_t off, int count, int *eof, void *
data);
static int read_large_bitmap_status(char *page, char **start, off_t off, int count, int *eof, void *
data);

struct proc_dir_entry *proc_host_bitmap_root;

int proc_host_bitmap_init(struct proc_dir_entry *root_entry)
{
    struct proc_dir_entry   *entry;
    struct proc_dir_entry   *second_entry;

    proc_host_bitmap_root = root_entry;
    entry = create_proc_entry("small_bitmap_status", S_IRWXO, proc_host_bitmap_root);
    if (entry)
        entry->read_proc = &read_small_bitmap_status;

    second_entry = create_proc_entry("large_bitmap_status", S_IRWXO, proc_host_bitmap_root);
    if (second_entry)
        second_entry->read_proc = &read_large_bitmap_status;

    if(entry && second_entry)
        return 0;
    else
        return -1;
}
int proc_host_bitmap_cleanup()
{
    remove_proc_entry("small_bitmap_status", proc_host_bitmap_root);
    remove_proc_entry("large_bitmap_status", proc_host_bitmap_root);

    return 0;
}

static int read_small_bitmap_status(char *page, char **start, off_t off, int count, int *eof, void *data)
{
    count = 0;
    count +=print_bitmap_info(small_bitmap, "Small Bitmap", page);
    return count;
}

static int read_large_bitmap_status(char *page, char **start, off_t off, int count, int *eof, void *data)
{
    count = 0;
    count +=print_bitmap_info(large_bitmap, "Large Bitmap", page);
    return count;
}

/* Name: print_bitmap_info
 * Description: Displays bitmap and stats to console.
 * History:
 * July 27, 2001 - Version 1.0, Max CiNiC project
 * July 28, 2001 - Removed destructive printing routine
 *               and replaced it with a cleaner version.
 *               No mallocs or bit shifts.-Max
 * July 30, 2001 - Added error tracking and proc functionality
 * */

int print_bitmap_info(bitmap_t* set, char* name, char* page){
    int count             = 0;
    int counter           = 0;
    int total_iterations  = 0;
    int bits_set_stat     = 0;
    int bits_clear_stat   = 0;
    int sitting_duck = 0;
    float percent_used;
    float percent_free;
    shrmem_t* mem = NULL;
    compkt_t* pkt;

    mem = (shrmem_t*)module_info.shared_mem_addr;

    count += sprintf(page+count,"\n\n%s:
", name);

    for (counter=0; counter < BITMAP_SIZE; counter++){
        total_iterations++;

        if(!(counter%8))
            count += sprintf(page+count,"    ");

        if(!(counter%32))
            count += sprintf(page+count,"\n");

        if(BITMAP_ISSET(counter, set)){
            count += sprintf(page+count,"1");
            sitting_duck = counter;
            bits_set_stat++;
        } else {
            count += sprintf(page+count,"0");
            bits_clear_stat++;
        }
    }
percent_used = (((float)bits_set_stat)/(float)BITMAP_SIZE * 100);
percent_free = (((float)bits_clear_stat/(float)BITMAP_SIZE) * 100);

count += sprintf(page+count,"\\n\\nStatistics: \n");
count += sprintf(page+count,"^^^^^^^^^^^^");
count += sprintf(page+count,"\nNumber of bits set: %i \n", bits_set_stat);
count += sprintf(page+count,"\nNumber of bits cleared: %i \n", bits_clear_stat);
count += sprintf(page+count,"\nPercentage used: %f \n", percent_used);
count += sprintf(page+count,"\nPercentage free: %f \n", percent_free);
count += sprintf(page+count,"\nNext available block: %i \n", set->next_avail);

/* Sitting duck hack.  */
/* If there is one packet in memory, it will find it and  */
/* print out some relevant data. VERY useful for debugging*/
/* in shared memory  */

if(bits_set_stat == 1){
    pkt = (compkt_t*) &mem->large_blocks[sitting_duck * LARGE_BLOCK_SIZE];
    count += sprintf(page+count, "(\%5i) Unkown func_id=\%i ret_val=\%li pkt_number=\%li\n",
    (int)pkt->pid, pkt->func_id, pkt->ret_val, pkt->pkt_number);
}

#ifdef BITMAP_PARANOID
    count += sprintf(page+count,"Collisions detected: %li\n", track_errors.collision_count);
count += sprintf(page+count,"Massive packets detected: %li\n", track_errors.massive_packet_count);
count += sprintf(page+count,"Sleeps required because bitmap was full: %li\n\n\n\n", track_errors.sleep_count);
#endif

return count;
}
com_h.h

/*
 * com_h.h - Cal Poly 3Com CiNIC project

 Header file for the Host side of the communications protocol.

 $Id: com_h.h,v 1.1.1.1 2002/06/13 08:28:39 max Exp $
 */

#ifndef _COM_H_H_
#define _COM_H_H_

#include "global.h"
#include "com.h"     /* protocol structs */

#define MAX_GET_BLOCK_ATTEMPTS 100

/* Initialize the protocol */
int protocol_init(void);

/* Shutdown and cleanup protocol */
int protocol_cleanup(void);

/* Return a pkt queue node pointing to a compkt of size pktsize */
compkt_t *get_shrmem_slot(unsigned long pktsize);

/* Release a pkt queue node, frees its memory and any other cleanup tasks */
void release_shrmem_slot(compkt_t *pkt);

/* TODO*/
void send_to_cohost(compkt_t *pkt);
void interrupt_handler(void);

#endif
/* com_h.c - Cal Poly 3Com CiNIC project */
/* Provides implementation of the protocol functions/threads on the host side. */
*/

#include <asm/semaphore.h>   /* up/down */
#include <linux/slab.h>      /* kmalloc     */
#include <linux/tqueue.h>    /* Task queues */
#include <linux/interrupt.h> /* mark_bh()     */
#include <asm/io.h>
#include "global.h"
#include "decode.h"
#include "host.h"
#include "com_h.h"
#include "bitmap_h.h"
#include "21554.h"

unsigned long pkt_counter = 0;

/* Bitmaps */
extern bitmap_t *small_bitmap, *large_bitmap;

/* Header semaphore for host->ebsa transmissions */
static struct semaphore host_header_lock;

/* Interrupt functions and task queue structs */
extern int interrupt_cohost(void);
extern void (*handler)(void); /* function pointer to interrupt handler */

int protocol_init(void)
{
    int err = 0;
    volatile shrmem_t* mem;

    mem = (shrmem_t*)module_info.shared_mem_addr;

    PRINT_DEBUG(3, "Initializing Protocol\n");

    /*
     * Remember that Mark exported the function pointer to the kernel's symbol table. This function pointer will get invoked, barring that we assign it to something (!NULL), as part of the top half. Effectively, my interrupt_handler() function is the top half.
     */
    handler = interrupt_handler;

    /*
     * Semaphore to protect critical header area from becoming a race condition. Not necessary, really, until this code becomes SMP/preemptible kernel safe
     */
    sema_init(&host_header_lock, 1);

    /* Initialize the bitmaps */
    bitmap_init();

    return err;
}
/* Shutdown and cleanup protocol */
int protocol_cleanup(void)
{
    int err = 0;

    bitmap_cleanup();

    return err;
}

/* Top half handler for host. */
void interrupt_handler(void)
{
    compkt_t *pkt;
    cyclic_queue_t start_block;
    volatile shrmem_t *mem;

    mem = (shrmem_t*)module_info.shared_mem_addr;

    PRINT_DEBUG(9, "CiNIC: Got packet back\n");

    start_block = mem->ebsa_header.pkt_index;

    /* Index into the appropriate shared memory array and set pointer. */
    if(start_block.block_size == 0)
    {
        pkt = (compkt_t*)&mem->small_blocks[start_block.block_index * SMALL_BLOCK_SIZE];
    }
    else
    {
        pkt = (compkt_t*)&mem->large_blocks[start_block.block_index * LARGE_BLOCK_SIZE];
    }

    /* This switch can be used as a crude filtering system. Thus, adapt it */
    /* to fit your needs. Right now, it does nothing */
    switch (readl(&pkt->func_id))
    {
        case SYS_SOCKET:
        case SYS_CONNECT:
        case SYS_RECV:
        case SYS_SENDTO:
        case SYS_CLOSE:
        case SYS_WRITE:
        case SYS_BIND:
        case SYS_LISTEN:
        case SYS_SETSOCKOPT:
        case SYS_READ:
        case SYS_ACCEPT:
        case SYS_SEND:
        case SYS_RECVFROM:
        case SYS_FSTAT64:
        case SYS_FCNTL:
        case SYS_GETSOCKNAME:
        {
            break;
        }
        default:
        {
            PRINT_DEBUG(1, "CiNIC: Error! in interrupt handler: Switch default hit!");
            Packet not recognized. [%i\n", pkt->func_id);
            /* oh well, we'll return -EINVAL */
            writel(-EINVAL, &pkt->ret_val);
            break;
        }
    }
    /* unlock the waiting pkt queue node, this should return */
    /* control to the system call that created the pkt */
    up(&pkt->process_lock);
}
/*  Returns a pkt_queue_node struct pointing to a compkt of size pktsize  */
compkt_t* get_shrmem_slot(unsigned long pktsize)
{
  long int i;
  long int sleep_count = 0;
  cyclic_queue_t tmp;
  compkt_t *pkt;
  shrmem_t *mem;

  mem = (shrmem_t*)module_info.shared_mem_addr;

  /* Somewhat dangerous. preliminary version */
  for(i = 0; i < MAX_GET_BLOCK_ATTEMPTS; i++)
  {
    tmp = get_free_block(pktsize);
    if (tmp.block_index != 1000)
      break;
    else
    {
      PRINT_DEBUG(1, "CiNIC: Error! Bitmap full. Sleeping. Try: %li\n",
                  sleep_count);
      track_errors.sleep_count++;
      set_current_state(TASK_INTERRUPTIBLE);
      schedule_timeout(BITMAP_NO_FREE_BLOCK_DELAY * HZ);
    }
  }
  if (i == MAX_GET_BLOCK_ATTEMPTS)
  {
    PRINT_DEBUG(1, "CiNIC: Error! Get free blocks failed. Returning error to app
                layer.\n"");
    return NULL;
  }

  /* Got block. Get pointer */
  if (tmp.block_size == 0)
    pkt = (compkt_t*)&mem->small_blocks[tmp.block_index * SMALL_BLOCK_SIZE];
  else
    pkt = (compkt_t*)&mem->large_blocks[tmp.block_index * LARGE_BLOCK_SIZE];

  /*
   * We know that every packet is linked with an application in userspace.
   * This kernel thread itself is actually running because the application
   * called it (trapped from user space) However, after we pass the packet to the
   * ebsa, this kernel thread will be put to sleep waiting for the
   * packet to return *from* the ebsa. So! We put it to sleep with a semaphore. This
   * semaphore is part of the packet and will be up'd by the interrupt handler
   * receiving the packet back from the ebsa.
   */
  sema_init(&pkt->process_lock,0);

  /*
   * Numbering the packets is a valuable tool for tracking what is occurring. I'd
   * leave this on all the time, regardless of whether you are debugging or not.
   */
  writel(pkt_counter++, &pkt->pkt_number);

  /* We are going to need the start block number later and on the cohost side */
  memcpy_toio(&pkt->start_block, &tmp, sizeof(cyclic_queue_t));

  return pkt;
}

/**************************************************************************
/*
void release_shrmem_slot(compkt_t *pkt)
{
  if (release_blocks(pkt) < 0)
  {
/*
 * No recovery. We'll lose the packet when pkt is
 * set to NULL.
 */
PRINT_DEBUG(1, "Error: Packet could not release bitmap locks!
"
);
}
pkt = NULL;
}

/*
 * Updates circular queue in shared memory with start block number
 * and signals for interrupt to be fired.
 * History: added crude mechanism for polling is queue
 * is full. This case is totally unlikely so
 * the algorithm will remain "crude"
 * Thanks to Jared Kwek for the idea.
 */
void send_to_cohost(compkt_t *pkt)
{
volatile shrmem_t*        mem;
volatile unsigned long dummy_variable;
mem  = (shrmem_t*)module_info.shared_mem_addr;
try_again:

down(&host_header_lock);
/*
 * Here comes a very important part! If we don't sync the
 * host and the cohost when they are accessing
 * the same data, then the ebsa will destroy the hosts
 * updates when the ebsa writes back its cache lines. See my senior
 * project for a detailed analysis. This is a very difficult
 * problem to track and will bring the entire system to
 * a crashing halt. Ideally, we will just disable the cache
 * on the mapped shared memory region. As of now, there
 * is no way to do this without disabling the ebsa's MMU.
 * So, we make we'll make sure that the no data that is share
 * with the ebsa's top half is touched. To implement this,
 * we ensure that the ebsa has finished handling the top
 * half and has flushed (clean and invalidated in ARM
 * terminology) its cache by checking IRQ status bits.
 */
if(iPendingCoHostINT()){
PRINT_DEBUG(9, "CiNIC: Cohost is busy. Retry interrupt later. Packet: %lu\n", pkt-
pkt_number);
    /* Ebsa still handling top half. Keep trying. */
    up(&host_header_lock);
    schedule();
    goto try_again;
}
switch (readl(&pkt->func_id)) {
    case SYS_SOCKET:
    case SYS_CONNECT:
    case SYS_BIND:
    case SYS_LISTEN:
    case SYS_ACCEPT:
    case SYS_RECV:
    case SYS_SEND:
    case SYS_SENDTO:
    case SYS_RECVFROM:
    case SYS_CLOSE:
    case SYS_WRITE:
    case SYS_READ:
    case SYS_FSTAT64:
    case SYS_Fcntl:
    case SYS_GETSOCKNAME:
    case SYS_SETSOCKOPT:

memcpy_fromio((void*)&mem->host_header.pkt_index, &pkt->start_block,
sizeof(cyclic_queue_t));

/*
 * We'll do a read to make sure all the posted writes on the 21554 are
flushed
 * before proceeding
 */
dummy_variable = pkt->pkt_number;
mb();

PRINT_DEBUG(9, "---->Packet Sending Number is %lu\n", pkt->pkt_number);

if (interrupt_cohost()){
    PRINT_DEBUG(1, "CiNIC: Interrupt did not fire!\n");
} else
    PRINT_DEBUG(5, "CiNIC: Interrupt fired successfully.\n");

    up(&host_header_lock);
break;

default: /* this should never happen */
    PRINT_DEBUG(1, "CiNIC Error! send_to_cohost switch default hit! [%i]\n", pkt->
>func_id);

    /* oh well we'll return -EINVAL */
    writeln(-EINVAL, &pkt->ret_val);
    up(&{pkt->process_lock}); /* unlock the waiting pkt queue node, this should
return */
    /* control to the system call that created the pkt
*/
    break;
}
}
Cbuf.h

/*
 * cbuf.h - Cal Poly 3Com CiNIC project

   Header file for circular buffer manipulation functions.

   History: 11/06/01 Max
*/

#ifndef _CBUF_
#define _CBUF_

#include "com.h"

#define CBUF_SIZE 1024

typedef struct circular_buffer_t{
    compkt_t *buf[CBUF_SIZE];
    unsigned long head;
    unsigned long tail;
    unsigned long pkt_count;
} cbuf_t;

void cbuf_init(cbuf_t *cbuf);
compkt_t* cbuf_remove(cbuf_t *cbuf);
int cbuf_add(cbuf_t *cbuf, compkt_t *pkt);

#endif
/* Initialize the circular buffer */
void cbuf_init(cbuf_t *cbuf)
{
    cbuf->head = 1;   /* reset queue */
    cbuf->tail = 0;
    cbuf->pkt_count = 0;   /*reset counter*/
}

/* Add to current head pointer and increment */
int cbuf_add(cbuf_t *cbuf, compkt_t *pkt)
{
    if (cbuf->head != cbuf->tail){
        /* not full */
        cbuf->buf[cbuf->head % CBUF_SIZE] = pkt;
        cbuf->head = (cbuf->head+1) % CBUF_SIZE;
        cbuf->pkt_count++;
        return 0;
    } else
    return -1;
}

/* remove tail element */
compkt_t* cbuf_remove(cbuf_t *cbuf)
{
    compkt_t *pkt = NULL;
    if((cbuf->tail + 1) % CBUF_SIZE != cbuf->head){
        /* not empty */
        pkt = cbuf->buf[cbuf->tail];
        cbuf->tail = (cbuf->tail + 1) % CBUF_SIZE;
        cbuf->pkt_count--;
        return pkt;
    } else {
        /* Trying to remove from empty circular buffer */
        return NULL;
    }
}
thread.h

/*
 * thread.h - Cal Poly 3Com CiNIC project
 * Header file for thread creation and destruction.
 */

#ifndef _THREAD_H_
define _THREAD_H_

#include <linux/types.h>      /* pid_t */
#include <linux/tqueue.h>
#include <asm/semaphore.h>    /* semaphore */

typedef struct thread_st{
    struct task_struct *thread;
    struct tq_struct tq;
    unsigned long flags;
    int (*function)(void*);
    struct semaphore lock;
    unsigned int sd;
    pid_t pid;

    /* Process name*/
    char name[16];
} thread_t;

/* Wraper functions. Use these to interface w/ thread subsystem */
void start_manager();
void stop_manager();

void start_worker(unsigned int sd);
int stop_worker(unsigned int sd, int option);
void stop_all_workers();

/*
 * Functions to start/stop threads
 */
void thread_launch(void *data);
void thread_daemon(int (*func)(void*), thread_t *tinfo, unsigned long flags);
void kill_thread(thread_t *tinfo);

/*
 * Functions called by threads to setup and exit
 */
void init_thread(thread_t *tinfo, char* thread_name, int option);
void thread_leave(thread_t *tinfo, int option);

#endif
thread.c
/*
 * thread.c - Cal Poly 3Com CiNIC project
 * Functions to streamline thread handling.
 * Original code from http://www.scs.ch/~frey/linux/kernelthreads.html
 * with some 'cought' a whole lot of modifications.
 * 05/27/2002 - Almost totally rewritten to use signals and fix
 * the zombie thread states.-max
 * 12/04/2001 - Rewritten to utilize keventd.
 * Mostly verbatim copy of web approach. Minor changes
 * to facilitate my plot to take over the world.
 * <humming> Pinky and the Brain, brain,... -max.
 * 06/07/2001 - Initial Coding by Rob
 */
#include <asm/signal.h>  /* Signal stuff */
#include <asm/uaccess.h> /* set_fs(KERNEL_DS) */
#include <linux/sched.h> /* current */
#include <linux/slab.h>  /* kmalloc */
#include <linux/kernel.h>
#include <asm/smplock.h>
#include "global.h"
#include "thread.h"
#include "com_e.h"
#include "manager_worker.h"
extern thread_lookup_t *thread_lookup[MAX_FILE_DESC];
extern thread_lookup_t manager_control;
/*
 * Wrapper to start manager kernel thread
 * Thread clones none of the parents memory, files, etc. (zilch).
 */
void start_manager()
{
    manager_control.thread = (thread_t*)kmalloc(sizeof(thread_t), GFP_KERNEL);

    PRINT_DEBUG(5, "CiNIC: Launching manager thread\n");

    /* Thread clones none of the parents memory, files, etc. (zilch)*/
    thread_daemon(manager_thread, manager_control.thread, 0);
}

/*
 * Wrapper function to kill the manager thread
 */
void stop_manager()
{
    kill_thread(manager_control.thread);
}

/*
 * Wrapper to start worker thread.
 * Give it the socket descriptor value it's responsible for
 * and off it goes! This nifty abstraction requires
 * the global thread_lookup array to be extern'd in.
 * Note that unlike the manager spawner, this baby has
 * none of the cute locks and protection devices. This
 * just basically tells the kernel to spawn thread and it
 * plows right on through. No waiting around for the thread
 * to start, brush its teeth etc.. This is intentionally done
 */
void start_worker(unsigned int sd)
{
    thread_lookup[sd]->thread = (thread_t*) kmalloc(sizeof(thread_t), GFP_KERNEL);
    thread_lookup[sd]->thread->sd = sd;
    mb();

    PRINT_DEBUG(5, "CiNIC: Launching worker thread: File descriptor : %u.\n", sd);
    kernel_thread(worker_thread, thread_lookup[sd]->thread, CLONE_FILES|CLONE_FS);
}

int stop_worker(unsigned int sd, int option)
{
    if(option == 0){
        /* Ensure no pending packets */
        if(thread_lookup[sd]->cbuf.pkt_count != 0){
            PRINT_DEBUG(1, "CiNIC: Error! Trying to kill thread with pending packets.\n")
            return -1;
        }
        /* Insert future tests here */
        kill_thread(thread_lookup[sd]->thread);
        kfree(thread_lookup[sd]->thread);
        thread_lookup[sd]->thread = NULL;
        return 0;
    } else if (option == 1) {
        kill_thread(thread_lookup[sd]->thread);
        kfree(thread_lookup[sd]->thread);
        thread_lookup[sd]->thread = NULL;
        return 0;
    } else {
        PRINT_DEBUG(1, "CiNIC: Error! Undefined option value passed to stop_worker");
        return -1;
    }
}

void stop_all_workers()
{
    unsigned long i = 0;

    for(i = 0; i < MAX_FILE_DESC; i++){
        if(thread_lookup[i] == NULL)
            continue;
        else {
            kill_thread(thread_lookup[i]->thread);
            kfree(thread_lookup[i]->thread);
            thread_lookup[i]->thread = NULL;
        }
    }
}

void thread_launch(void *data)
{
    thread_t *tinfo = (thread_t*)data;
}
void thread_daemon(int (*func)(void*), thread_t *tinfo, unsigned long flags)
{
    init_MUTEX_LOCKED(&tinfo->lock);
    /* Initialize the task queue structure */
    INIT_LIST_HEAD(&tinfo->tq.list);
    /* Setup task queue */
    tinfo->tq.sync = 0;
    tinfo->tq.routine = thread_launch;
    tinfo->tq.data = tinfo;
    /* assign thread function to be executed by kernel thread */
    tinfo->function = func;
    /* assign flag parameters for new thread */
    tinfo->flags = flags;
    /* and schedule it for execution */
    schedule_task(&tinfo->tq);
    /*wait till it has reached the setup_thread routine */
    down(&tinfo->lock);
}

void init_thread(thread_t *tinfo, char *thread_name, int option)
{
    /* No big kernel lock necessary because cohost will never be SMP */
    /* House keeping. Not used by anything. Might be handy in the future though */
    tinfo->thread = current;
    tinfo->pid = current->pid;
    sprintf(tinfo->name, "%15s", thread_name);
    /* Give a process name */
    strncpy(current->comm, thread_name);
    /* Not entirely sure what signals are inhereted, so we'll clear them */
    /* to make sure that nothing wierd happens */
    sigfillset(&current->blocked);
    if (option == 0){
        /*
         * Manager thread needs to receive signals from the dying children
         * and the death blow from the insmod process
         */
        siginitsetinv(&current->blocked, sigmask(SIGKILL)|sigmask(SIGCHLD));
    } else if (option == 1){
        /*
         * Workers only respond to one signal, the death blow from the
         * manager
         */
    }
siginitsetinv(&current->blocked, sigmask(SIGKILL));
} else {
    PRINT_DEBUG(1, "CiNIC: Error! Bad option parameter passed to init_thread!");
}

PRINT_DEBUG(S, "CiNIC: Thread launched: %s, pid=%i\n", thread_name, tinfo->pid);

/* tell creator that we are ready and let him continue */
up(&tinfo->lock);
}

/*
* Called by thread when dying
* Option flag is used to designate whether we are initializing
* the worker or the manager
*/
void thread_leave(thread_t *tinfo, int option)
{
    /*
    * No kfree(!) on purpose. Kernel will release it for us. Doing so
    * will kernel oops.
    */
    tinfo->thread = NULL;
    /* Memory barrier.
    * Break kernel etiquette by explaining a little theory. mb() will
    * ensure that the software/hardware do not reorder the read/write
    * or attempt to cache volatile memory so as to improve efficiency.
    * This is important when dealing with control registers etc. where
    * order is critical. However, for this applications,
    * it really doesn't do much because no other process will be
    * polling this and the up() will force a memory barrier anyways...
    * But, hey if the programmer I stole this code from wants one,
    * who am I to stop him.
    */
    mb();
    /* Notify kill_thread() that we are terminating */
    up(&tinfo->lock);

    if (option == 0)
    {
        /* We are killing the manager thread, notify keventd */
        kill_proc(2, SIGCHLD, 1);
    } else if (option == 1)
    { /* We are killing a worker, notify manager */
        kill_proc(manager_control.thread->pid, SIGCHLD, 1);
    } else {
        PRINT_DEBUG(1, "CiNIC: Error! Bad option parameter passed to init_thread!\n");
    }
}

/*
* Kills thread
*/
void kill_thread(thread_t *tinfo)
{
    if (tinfo->thread == NULL) {
        PRINT_DEBUG(1, "CiNIC: Error! kill_thread, thread non-existant: pid=%i\n", tinfo->pid);
        return;
    }
    init_MUTEX_LOCKED(&tinfo->lock);
    PRINT_DEBUG(5, "CiNIC: Killing thread, pid=%i\n", tinfo->pid);
    /*
* "Do you expect me to talk?"
* "No, mister thread, I expect you to DIE!"
 */
kill_proc(tinfo->pid, SIGKILL, 1);

/* block till thread terminates */
down(&tinfo->lock);

}
com_e.h

/*
 * com_e.h - Cal Poly 3Com CI NIC project
 * Header file for the Ebsa side of the communications protocol.
 * $Id: com_e.h,v 1.1.1.1 2002/06/13 08:28:39 max Exp $ */

#ifndef _COM_E_H_
#define _COM_E_H_

#include <linux/list.h>
#include "global.h"
#include "com.h"
#include "ebsa.h"
#include "thread.h"
#include "cbuf.h"

/* This is the max number of file descriptors that 
   we will deal with on the ebsa. Not sure if
   this can be extended. So, we might have a
   problem later if the host opens more than
   1024 sockets */
#define MAX_FILE_DESC 1024

typedef struct thread_lookup_entry{
    /* thread */
    thread_t            *thread;
    struct semaphore    rts;

    /* And the circular buf */
    cbuf_t              cbuf;
} thread_lookup_t;

/* Initialize/cleanup ebsa side threads */
int protocol_init(void);
int protocol_cleanup(void);

/* Place the queue node on the queue to be returned to the host */
void send_to_host(compkt_t *pkt);

/* TODO: Remove me */
int manager_thread(void *t);

#endif
com_e.c

/*
 * com_e.c - Cal Poly 3Com CiNIC project
 *
 * Provides implementation of the protocol functions on the ebsa side.
 *
 * $Id: com_e.c,v 1.1.1.1 2002/06/13 08:28:38 max Exp $
 */

#include <asm/semaphore.h> /* up/down */
#include <linux/slab.h>     /* kmalloc */
#include <linux/in.h>       /* sockaddr_in */
#include <asm/unistd.h>     /* _syscall? */
#include <asm/uaccess.h>
#include <linux/interrupt.h> /* mark_bh() */
#include <linux/interrupt.h>/* mark_bh() */
#include "com_e.h"
#include "global.h"
#include "thread.h"
#include "syscalls_e.h"
#include "ebsa.h"
#include "decode.h"
#include "21554.h"

int errno;
_syscall3(int,write,int,fd,const char*,buf,unsigned int,len);
_syscall3(int,read,int,fd,const char*,buf,unsigned int,len);
_syscall3(long,fstat64,long,fd,struct stat64*,st,long,flags);
_syscall3(long,fcntl,unsigned int,fd,unsigned int,cmd,unsigned long,arg);

/*
 * Static allocation of lookup array.
 * This contain control structures that
 * corresponding to all possible worker threads that
 * we might need. We index into the array with
 * the socket descriptor.
 */
thread_lookup_t *thread_lookup[MAX_FILE_DESC];

/*
 * Here is the control structure for the manager thread.
 * For lack of better name I will call it manager_control
 */
thread_lookup_t manager_control;

/*
 * Interrupt functions
 */
extern int interrupt_host(void);
extern void (*handler)(void);

/*
 * Function pointer to interrupt handler
 */
void interrupt_handler(void);

/*
 * Init function starts threads, inits semaphores, etc.
 */
int protocol_init(void)
{
    int err = 0;
    int i = 0;

    /*Semaphore initializations */
    sema_init(&manager_control.rts, 0);

    /* bloody thing needs to be all NULL. */
    for(i = 0; i < MAX_FILE_DESC; i++)
thread_lookup[i] = NULL;

    /* global thread control struct setup for manager thread */
    manager_control.thread = (thread_t*) kmalloc(sizeof(thread_t), GFP_KERNEL);

    if(manager_control.thread == NULL){
        PRINT_DEBUG(1, "CiNIC: Error! Unable to kmalloc memory to launch manager thread.
Exit.
");
            return -1;
    }

    /* Reset the circular buffer */
    cbuf_init(&manager_control.cbuf);

    /* Start manager thread */
    start_manager();

    /* Interrupt and bottom half initializations. */
    handler = interrupt_handler;

    return err;
}

    /* Kill threads, stop protocol. */
    int protocol_cleanup(void)
    {
        int err = 0;

        stop_manager();
        stop_all_workers();

        /* err doesn't do anything now. I'll keep it around for future use. */
        return err;
    }

    /* Interrupt handler */
    void interrupt_handler(void)
    {
        volatile cyclic_queue_t start_block;
        volatile shrmem_t* mem = NULL;
        compkt_t* pkt = NULL;

        mem = (shrmem_t*)g_shrmem;

        /*
        * We need to ensure that the cache is clean prior to reading. 
        * This will ensure we don't read a stale value from cache.
        */
        cpu_sa110_cache_clean_invalidate_all();

        start_block = mem->host_header.pkt_index;

        if(start_block.block_size == SMALL_BITMAP)
            pkt = (compkt_t*)&mem->small_blocks[start_block.block_index * SMALL_BLOCK_SIZE];
        else
            pkt = (compkt_t*)&mem->large_blocks[start_block.block_index * LARGE_BLOCK_SIZE];

        PRINT_DEBUG(9, "CiNIC: Packet arrived at Interrupt handler:
        ");

        //Packet Number: %lu Packet type: %ui and sd: %ui, and pid: %i
        switch (pkt->func_id) {
        
        }
case SYS_SOCKET:
    case SYS_CLOSE:
        /* Push packet to manager thread */
        if(cbuf_add(&manager_control.cbuf, pkt) < 0){
            PRINT_DEBUG(1, "CiNIC: Error! In Interrupt Handler: Unable to add packet to circular buffer.
            /* This leads to memory leak! And the host app. will freeze.
            * Nothing much that can be done. Disaster soon await us!
            */
            break;
        }
        /* Now that the packet is in the circular buffer, we signal the thread to start. */
        up(&manager_control.rts);
        break;
    case SYS_BIND:
    case SYS_CONNECT:
    case SYS_LISTEN:
    case SYS_ACCEPT:
    case SYS_GETSOCKNAME:
    case SYS_SEND:
    case SYS_RECV:
    case SYS_SENDTO:
    case SYS_RECVFROM:
    case SYS_SETSOCKOPT:
    case SYS_WRITE:
    case SYS_READ:
    case SYS_FSTAT64:
    case SYS_FCNTL:
        /* Push packet into corresponding worker thread. */
        if(thread_lookup[pkt->sd] == NULL){
            PRINT_DEBUG(1, "CiNIC: Error! In Interrupt Handler: Packet is destined to non-existant worker thread.
            goto end_interrupt_handler;
        }
        if(cbuf_add(&thread_lookup[pkt->sd]->cbuf, pkt) < 0){
            PRINT_DEBUG(1, "CiNIC: Error! In Interrupt Handler: Unable to add packet to circular buffer.
            goto end_interrupt_handler;
        }
        PRINT_DEBUG(7, "ABOUT TO SIGNAL WORKER THREAD
        /* Signal thread. */
        up(&thread_lookup[pkt->sd]->rts);
        break;

        default:
            PRINT_DEBUG(1, "CiNIC: Error! Switch default hit! Packet not recognized. [%i]
            goto end_interrupt_handler;
            break;
    }

    end_interrupt_handler:
        cpu_sa110_cache_clean_invalidate_all();
    }

    /* Updates circular queue in shared memory with start block number
    * and signals for interrupt to be fired.
    */
    void send_to_host(compkt_t *pkt)
    {
        volatile shrmem_t* mem;
mem = (shrmem_t*)g_shrmem;

try_again:

    /*
     * Here we do the same thing as we did on the host.
     * We will ensure that the top half on the host has
     * finished processing before firing off another
     * interrupt.
     */
    if(iPendingHostINT()){
        PRINT_DEBUG(9, "CiNIC: Error! Host is busy. Retry interrupt later. Packet: %lu\n",
                   pkt->pkt_number);
        /* Ebsa still handling top half. Keep trying. */
        schedule();
        goto try_again;
    }

mem->ebsa_header.pkt_index = pkt->start_block;

    /* Force write back to physical RAM */
    cpu_sa110_cache_clean_invalidate_all();

    if (interrupt_host()){
        PRINT_DEBUG(1, "CiNIC: Error! Interrupt did not fire!\n");
    } else {
        PRINT_DEBUG(9, "CiNIC: Interrupt fired successfully.\n");
    }
}
#ifndef _MANAGER_WORKER_
#define _MANAGER_WORKER_
#include "global.h"

int manager_thread(void *t);
int worker_thread(void *t);

#endif
manager_worker.c

#include <asm/semaphore.h> /* up/down */
#include <linux/slab.h> /* kmalloc */
#include <linux/unistd.h>
#include "global.h"
#include "com.h"
#include "com_e.h"
#include "syscalls_e.h"
#include "thread.h"
#include "cbuf.h"
#include "decode.h"

#define __KERNEL_SYSCALLS__

int errno;

extern thread_lookup_t manager_control;
extern thread_lookup_t *thread_lookup[MAX_FILE_DESC];

/*
 * This function is a bit wierd but this makes everything much
 * cleaner in the long run <trust me>. Atypically, we return NULL
 * regardless of what type of error happens. We will
 * determine why we exited by checking signals in the caller.
 * Prior to this, we passed in a double pointer and returned
 * the packet through the function parameters. This works but
 * makes for real ugly code and forces us to check signals
 * in this function also. As a result, we leave the signal
 * calling to the caller.
 * /

compkt_t* wait_on_cbuf(thread_lookup_t *cur)
{
    compkt_t *ret;
    if(down_interruptible(&(cur->rts)) == -EINTR)
        return NULL;
    ret = cbuf_remove(&cur->cbuf);
    if (!ret)
    {
        PRINT_DEBUG(1, "CiNIC: Error! Removed NULL pointer from cbufln");
        return NULL;
    }
    return ret;
}

/* Description: handles the creation of threads assigned to
 * specific socket/file_descriptor values.
 * Note3:
 * 1.Thread_lookup is static to avoid kmalloc'ing.
 * */
int manager_thread(void *t)
{
    long args[6];
    thread_t *tinfo = (thread_t*)t;
    compkt_t *pkt;
    struct task_struct *curtask = current;
    init_thread(tinfo, "CiNIC manager", 0);
    for(;;)
    {
        /* We need to reap our dead children. */

if (signal_pending(curtask) && sigismember(&(curtask->pending.signal), SIGCHLD)){
    while (waitpid(-1, (unsigned int *)0, __WALL|WNOHANG) > 0)
    {
        flush_signals(curtask);
        recalc_sigpending(curtask);
    }
}

/* Children are now reaped, check to see if we should die. */
if (signal_pending(curtask) && sigismember(&(curtask->pending.signal), SIGKILL)){
    /* Signals will be cleaned when the thread dies. No cleanup necessary. */
    break;
}

/* Now, wait for a packet to handle */
pkt = wait_on_cbuf(&manager_control);

if (!pkt){
    if(sigismember(&(curtask->pending.signal), SIGKILL)){
        /* We got a signal, while sleeping, to die. */
        break;
    } else {
        /* Two possibilities here:
        * 1) we removed a NULL pointer from
        *    the circular buffer. Obviously, something
        *    strange has happened. We'll recover by
        *    ignoring it and going back to waiting
        *    for another packet.
        * 2) We got a SIGCHLD signal. We recover by
        *    continuing this for() loop which will
        *    reap the children and then sleep waiting
        *    for another packet.
        */
        continue;
    }
}
/* Got packet! */
switch(pkt->func_id){
case SYS_SOCKET:
    args[0] = pkt->func.socket.family;
    args[1] = pkt->func.socket.type;
    args[2] = pkt->func.socket.protocol;
    pkt->ret_val = e_sys_socketcall(SYS_SOCKET, args);
    if(thread_lookup[pkt->ret_val] != NULL){
        PRINT_DEBUG(1, "CiNIC: Error! Thread structure is !NULL\n");
        /* TODO: how to recover. Thread, somehow, already exists! We
         * return to application on host saying that we can't
         * get a socket */
        sys_close(pkt->ret_val);
        pkt->ret_val = -1;
        send_to_host(pkt);
        break;
    }
    thread_lookup[pkt->ret_val] = (thread_lookup_t*)
        kmalloc(sizeof(thread_lookup_t), GFP_KERNEL);
    if(thread_lookup[pkt->ret_val] == NULL){
        /* Same as above error */
        PRINT_DEBUG(1, "CiNIC: Error! kmalloc failed. thread structure
        * not initialized.\n");
        sys_close(pkt->ret_val);
        pkt->ret_val = -1;
        send_to_host(pkt);
    }
    sema_init(&thread_lookup[pkt->ret_val]->rts, 0);
    cbuf_init(&thread_lookup[pkt->ret_val]->cbuf);
start_worker(pkt->ret_val);
send_to_host(pkt);
break;

case SYS_CLOSE:
  
  /*
   * Rather crazy checking this after we checked it at the
   * interrupt handler. It can happen though, if a CLOSE was
   * queued up and executed in the interrum
   */
  if(thread_lookup[pkt->sd] == NULL){
    PRINT_DEBUG(1, "CiNIC: Error! Trying to perform CLOSE on a
    thread that doesn't exist!\n"");
    /* TODO: Wow, this is an interesting one. I
       suppose we can fool the host into thinking the
       close actually worked? We'll try closing socket
       and see what happens.*/
    pkt->ret_val = sys_close(pkt->sd);
    send_to_host(pkt); /* return to host */
  }

  if(stop_worker(pkt->sd, 0)){
    /* TODO: Oh my! No friggin clue what to do.
       I've decided to keep socket open in this
       case. */
    PRINT_DEBUG(1, "CiNIC Error! Unable to stop stop worker
    thread. \n");
    pkt->ret_val = -1;
    send_to_host(pkt);
    break;
  }

  kfree(thread_lookup[pkt->sd]);
  thread_lookup[pkt->sd] = NULL;
  pkt->ret_val = sys_close(pkt->sd);
  send_to_host(pkt);
  break;

default:
  /* This should never happen.*/
  PRINT_DEBUG(1, "CiNIC Error! Switch default hit in
do_socket_close_thread. \n");
  return 0;
}

int worker_thread(void *t)
{
  compkt_t *pkt;
  thread_t *tinfo = (thread_t*)t;
  long args[6];

  init_thread(tinfo, "CiNIC worker", 1);
  PRINT_DEBUG(9, "------>Worker handling: socket %u\n", tinfo->sd);

  /*
   * Remember that we only accept one signal (SIGKILL). Therefore, we
   * don't need to check for the signal type like we did on the
   * manager.
   */
  while (!signal_pending(current)) {
/* Now, wait for a packet to handle */
pkt = wait_on_cbuf(thread_lookup[tinfo->sd]);

if (!pkt)
  if(!signal_pending(current)){
    /*
    * We got a signal, while sleeping. We know from the
    * thread init that this is a SIGKILL. Therefore,
    * there is no need to clear the signal etc.
    * "To be or not to be?"
    * ANSWER: not be.
    */
    break;
  } else {
    /*
    * One possibility:
    *  1) we removed a NULL pointer from
    *     the circular buffer. Obviously, something
    *     strange has happened. We'll recover by
    *     ignoring it and going back to waiting
    *     for another packet.
    */
    continue;
  }

/* Got Packet! */
switch (pkt->func_id) {
  case SYS_CLOSE:
    /* Fall through */
  case SYS_SOCKET:
    PRINT_DEBUG(1, "CiNIC: Error! Worker thread got SOCKET/CLOSE! Major
screw-up happened somewhere!\n"));
    break;
  case SYS_CONNECT:
    args[0] = pkt->sd;
    args[1] = (unsigned long)&pkt->func.connect.addr;
    args[2] = pkt->func.connect.addrlen;
    pkt->ret_val = e_sys_socketcall(SYS_CONNECT, args);
    break;
  case SYS_BIND:
    args[0] = pkt->sd;
    args[1] = (unsigned long)&pkt->func.bind.addr;
    args[2] = pkt->func.bind.addrlen;
    pkt->ret_val = e_sys_socketcall(SYS_BIND, args);
    break;
  case SYS_LISTEN:
    args[0] = pkt->sd;
    args[1] = pkt->func.listen.queuelen;
    pkt->ret_val = e_sys_socketcall(SYS_LISTEN, args);
    break;
  case SYS_ACCEPT:
    args[0] = pkt->sd;
    args[2] = (unsigned long)&pkt->func.accept.addrlen;
    args[1] = (unsigned long)&pkt->func.accept.addr;
    pkt->ret_val = e_sys_socketcall(SYS_ACCEPT, args);
    break;
  case SYS_RECV:
    args[0] = pkt->sd;
    args[1] = (unsigned long)&pkt->func.recv.msg;
    args[2] = pkt->func.recv.msglen;
args[3] = pkt->func.recv.flags;

pkt->ret_val = e_sys_socketcall(SYS_RECV, args);
/* set copy_len to header size plus number of bytes in message */
if (pkt->ret_val > 0) {
    pkt->copy_len = COM_PKT_HEADER_SIZE + COM_RECV_HEADER_SIZE +
    pkt->ret_val;
}
break;

case SYS_SEND:
    args[0] = pkt->sd;
    args[1] = (unsigned long)&pkt->func.send.msg;
    args[2] = pkt->func.send.msglen;
    args[3] = pkt->func.send.flags;
    pkt->ret_val = e_sys_socketcall(SYS_SEND, args);
    pkt->copy_len = COM_PKT_HEADER_SIZE + COM_SEND_HEADER_SIZE;
    break;

case SYS_SENDTO:
    args[0] = pkt->sd;
    args[1] = (unsigned long)(pkt->func.sendto.data + pkt->func.sendto.msg_off);
    args[2] = pkt->func.sendto.msglen;
    args[4] = (unsigned long)(pkt->func.sendto.data + pkt->func.sendto.addr_off);
    pkt->ret_val = e_sys_socketcall(SYS_SENDTO, args);
    pkt->copy_len = COM_PKT_HEADER_SIZE;
    break;

case SYS_RECVFROM:
    args[0] = pkt->sd;
    args[1] = (unsigned long)(&pkt->func.recvfrom.data + pkt->func.recvfrom.msg_off);
    args[2] = pkt->func.recvfrom.msglen;
    args[3] = pkt->func.recvfrom.flags;
    args[4] = (unsigned long)(&pkt->func.recvfrom.data + pkt->func.recvfrom.msglen);
    args[5] = (unsigned long)&pkt->func.recvfrom.addrlen;
    pkt->ret_val = e_sys_socketcall(SYS_RECVFROM, args);
    pkt->copy_len = COM_PKT_HEADER_SIZE;
    /* set copy_len to header size plus number of bytes in message */
    if (pkt->ret_val > 0) {
        pkt->copy_len = COM_PKT_HEADER_SIZE + COM_RECV_HEADER_SIZE +
    }
}
break;

case SYS_GETSOCKNAME:
    args[0] = pkt->sd;
    args[1] = (unsigned long)&pkt->func.getsockname.addrlen;
    args[2] = (unsigned long)&pkt->func.getsockname.addr;
    pkt->ret_val = e_sys_socketcall(SYS_GETSOCKNAME, args);
    break;

case SYS_SETSOCKOPT:
    args[0] = pkt->sd;
    args[1] = pkt->func.setsockopt.level;
    args[2] = pkt->func.setsockopt.optname;
    args[3] = (unsigned long)pkt->func.setsockopt.optval;
    pkt->ret_val = e_sys_socketcall(SYS_SETSOCKOPT, args);
pkt->copy_len = COM_PKT_HEADER_SIZE;
break;

case SYS_WRITE:
pkt->ret_val = write(pkt->sd,
pkt->func.write.msg,
pkt->func.write.msglen);
pkt->copy_len = COM_PKT_HEADER_SIZE;
break;

case SYS_READ:
pkt->ret_val = read(pkt->sd,
pkt->func.read.msg,
pkt->func.read.msglen);
pkt->copy_len = COM_PKT_HEADER_SIZE + COM_READ_HEADER_SIZE + pkt->ret_val;
break;

case SYS_FSTAT64:
pkt->ret_val = fstat64(pkt->sd,
pkt->func.fstat64.st,
pkt->func.fstat64.flags);
break;

case SYS_FCNTL:
pkt->ret_val = fcntl(pkt->sd,
pkt->func.fcntl.cmd,
(unsigned long)pkt->func.fcntl.arg);
break;

default:
PRINT_DEBUG(1, "CiNIC: Error! Default switch hit in worker thread!
\n");
    pkt->ret_val = -EFAULT; /* good enough error code */
    break;
} /*switch*/
send_to_host(pkt);
} /*while*/

thread_leave(tinfo, 1);
return 0;