Buenos on the ARM architecture

October 25, 2008

Contents

1 Introduction

Buenos is an educational operating system developed in 2003 at Helsinki University of Technology (TKK). The system runs on top of a MIPS emulator (YAMS) and is intended to be extendable by students. On our "Project In Embedded Systems" course we chose to port Buenos to the ARM architecture to get experience with ARM systems programming and also to see if Buenos really was that extendable. When we started the Buenos-ARM porting project we only had little knowledge of the details of the ARM processor and system architecture, so considerable time was spent on gathering information on the architecture.

This manual is organized into three sections. The first part documents the emulated hardware environment which consists of the softgun ARM emulator and our additions to it. The second part gives an overview of the changes we had to make to Buenos to support the ARM architecture. Finally, the third part is a short hands-on tutorial on working in the Buenos-ARM environment. It explains how to compile, configure, run and debug Buenos on ARM.

2 Softgun emulator and the ABC board

2.1 Introduction

For the purposes of porting Buenos to ARM we briefly surveyed three ARM emulators – Qemu, Gxemul and Softgun. Since Buenos depends on many devices in the YAMS MIPS emulator and because we wanted to keep the changes to Buenos minimal, we knew that we would have to add support for some of these YAMS devices also to our chosen ARM emulator. After spending a few hours reading the source code of the each emulators we got the impression that Softgun would be the best choice given our need to add new devices to it, since it is very simple. We were able to add a "hello world" device and a new board already during the initial survey.

Softgun is a simple, extensible ARM system emulator. Its CPU emulation is a simple interpreter, which fetches opcode from memory and interprets it mostly with clean C code. Some opcodes are optimized in x86 assembly but also C versions are available and are in use for other architectures. The system abstraction in Softgun is called a board and defined in a separate C file. A board definition is a set of devices and their interconnections. The board definitions in Softgun are fairly flexible, allowing arbitrary connections between I/O pins of emulated devices.

We named our imaginary ARM board "ARM Buenos Computer", or ABC in short. The board is based on an Atmel AT91RM9200 SoC, but some of the Atmel devices are disabled since there is no support for them in Buenos, and the YAMS devices have been added to the system to make porting easier. The board can be found under the boards directory of softgun (softgun/boards/abc.c), and the added YAMS devices under the yams directory of softgun (softgun/yams/*.c). Due to the modular design of Softgun very few other changes were necessary. It is important to note that since SMP is not common on embedded systems Buenos on ARM supports only one processor unlike on MIPS.

2.2 Physical memory device layout

Table 1 shows what is mapped in physical memory. MMIO indicates that it's Memory Mapped IO device. dram0 is mapped to two places because we wanted the kernel to run in `0x80000000` as in MIPS and dram0 must also be mapped to the beginning of memory because interrupt vectors reside there.

2.3 Physical memory address space layout

Table 2 shows how kernel uses it's address space. Because dram0 is mapped to two places, kernel is also found from the beginning of the physical memory. First level page table is always in the same address to ease modifying it. Notice that kernel's initial stack grows downwards really starting from `0x8000fffc`.

Table 1: ABC memory mappings

Address	Size	Description
0x00000000	0x10000000	dram0 (System RAM)
0x00000000	0x00100000	flash (Flash)
0x10000000	0x01000000	flash (Flash)
0x80000000	0x10000000	dram0 (System RAM)
0x90000000	0x00200000	dram1 (Ramdisk RAM)
0xffff0000	0x00001000	yams descriptor metadevice MMIO
0xffff1000	0x00001000	yams command line MMIO
0xffff8000	0x0000001c	yams device MMIO
0xffff8000	0x0000000c	yams tty MMIO
0xffff800C	0x00000004	yams meminfo MMIO
0xffff8014	0x00000004	yams shutdown MMIO
0xffff8018	0x00000008	yams cpustatus MMIO
0xffffff00	0x00000010	mc (Memory controller MMIO)
0xfffff000	0x00000200	aic (Advanced Interrupt Controller MMIO)
0xfffffc00	0x00000100	pmc (Power Management Controller MMIO)
0xfffffd00	0x00000100	st (System Timer MMIO)

Table 2: Buenos physical memory address space layout

Address	Size	Description
0x80000000	0x00004000	Interrupt vector and jump code
0x80004000	0x00004000	Main translation table (page table level 1)
0x80008000	0x00008000	Initial stack
0x80010000	0x00100000	Kernel code

2.4 Virtual memory map

Table 3 shows how memory works after virtual memory has been enabled. In Access column `Priv. R/W` means that the memory can only be accessed from kernel mode. `User R/W` can be accessed from user mode and from kernel mode. The page table may restrict access for reading only.

2.5 Non-YAMS devices

2.5.1 CPU

Softgun 0.16 emulates ARM CPU of the ARM9 family in both normal and thumb mode.

2.5.2 MMU

The MMU in ABC emulates the MMU in the ARM920T processor. In buenos drivers/_cp15.S provides a C level interface to the MMU.

Table 3: Buenos virtual memory mappings

Virtual Address	Size	Physical Address	Access	Description
0x00000000	0x00100000	0x00000000	Priv. R/W	Directly mapped interrupt vector
0x10000000	0x00100000	0x00100000	Priv. R/W	Directly mapped flash
0x30000000	0x50000000	in pagepool	User R/W	User space (processes)
0x80000000	0x01000000	0x80000000	Priv. R/W	Directly mapped kernel code
0x90000000	0x00100000	0x90000000	Priv. R/W	Directly mapped ramdisk
0xfff00000	0x00100000	0xfff00000	Priv. R/W	Directly mapped for I/O

2.5.3 Memory controller (MC)

2.5.4 Flash

ABC has a 16-megabyte AM29LV128ML NOR flash chip. Only the first megabyte of the chip is memory-mapped so that it can be read easily. Writing to the flash is rather complex. The chip has 64-kilobyte sectors. As it is NOR flash, changing ones into zeros is easy but to change zeros to ones you have to erase an entire sector. In practice most writes on TFS (the Buenos filesystem) requires erasing an entire block. In buenos drivers/amdflash.c handles reading and writing to flash.

2.5.5 DRAM

ABC has two DRAM chips. A 2-megabyte "dram1" holds a ramdisk we were forced to add due to slowness of NOR flash and time constraints. A 1-megabyte "dram0" is the actual system RAM. "dram1" is available through yams descriptor as "ramdisk" device. With more time, implementing a ramdisk in Buenos, using the Flash chip for persistent storage and system RAM for runtime access, would make the system more realistic.

2.5.6 Advanced interrupt controller (AIC)

The Advanced Interrupt Controller emulates the Atmel AT91AIC. It is used by Buenos simply to enable the necessary interrupts (timer most importantly), but allows rather fine-grained control of routing of interrupts from actual interrupt-generating devices to the CPU. The definitive source for AIC documentation is the Atmel AT91RM9200 [2] SoC documentation. at91.c initializes AIC by writing $1 << i$ to ith AIC Source Vector Register so that cswitch_arm.S can read the cause of the interrupt from AIC Interrupt Vector Register and pass it to interrupt_handle. For instance INTERRUPT_CAUSE_HARDWARE_5 is defined as $(1 << 15)$.

2.5.7 Power management controller (PMC)

The Power Management Controller emulates the Atmel AT91PMC. It derives clocks for all the devices of the system from two master oscillators. On ABC, it

is only explicitly connected to the System Timer device. The definitive source for PMC documentation is the ATMEL AT91RM9200 [2] SoC documentation. at91.c initializes PMC by writing to Main Oscillator Register (MOR) to enable the oscillator and set it to start after one slow cycle and by writing to Master Clock Register (MCKR) to select main clock as the clock source.

2.5.8 System timer (ST)

The System Timer emulates the Atmel AT91ST. It can be programmed to generate timer interrupts at specific intervals and to calculate elapsed time. On ABC the timer interrupts are connected to IRQ1 of AIC. at91.c initializes ST to raise IRQ pin approximately 1000 times per second and configures AIC to generate interrupts for ARM when the pin is high.

2.6 YAMS devices

The "descriptor" device implements a YAMS style device descriptor meta device that tells Buenos about devices that are present. On ABC the device has been mapped to a different address than under YAMS, but otherwise behaves identically. The memory address was changed mainly to keep memory mapped I/O devices closer to each other in memory space. The changes to the interface were two new device types:

- YAMS_DESCRIPTOR_TYPE_FLASH (0x302)
- YAMS_DESCRIPTOR_TYPE_RAMDISK (0x303)

Other YAMS-style devices that were implemented include "commandline", "cpustatus", "meminfo", "shudown" and "tty". In particular hard disk device was not implemented and tty device only supports polling at the moment.

2.7 Other changes to softgun

As expected we could not limit our changes to softgun to just adding new devices. This section documents changes we had to make to other parts of softgun.

2.7.1 GDB breakpoint fix

Softgun supports software breakpoints. When GDB asks softgun to set a breakpoint at address ADDR it replaces the instruction at ADDR with a break instruction and makes a backup of the original instruction. We noticed that with a simple

```
break ADDR
continue
continue
```

the first "continue" returns with PC set to ADDR+4 and not ADDR. The second "continue" then continues at ADDR+4 and the code at ADDR is never executed. As a temporary workaround we changed the implementation of the breakpoint instruction (i_bkpt() function in instructions.c) to decrement the value of PC with 4.

2.7.2 Command line option for setting buenos command line (-k)

The "commandline" YAMS-style device is not very useful unless there is some way to pass it the desired string from command line. To make it possible to set buenos boot command line easily we added a new command line option "-k" to parse_commandline function in softgun.c

2.7.3 The armdebug configuration option

We modified arm9cpu.c so that it calls the function debug_print_instruction() on every clock cycle if the configuration option "armdebug" is set to a non-zero value. This allowed us to easily execute temporary debugging and tracing code in the emulator.

2.7.4 Configuration file abc.sg

The board configuration in "buenos/softgun/abc.sg" can be used to change various settings of the ABC hardware. By default it contains some global settings like board name, path to the dynamic library that defines the board hardware configuration, and paths used by softgun to store temporary files. The global section also includes armdebug flag that can be used to enable custom debugging code in softgun (see end of debugging chapter in Tutorial section).

Some aspects of the hardware can also be configured using the board configuration file. On ABC board these are CPU clock rate, DRAM chip sizes and flash chip type. The regions section only contains aliases that can be used with the -l command line option to specify an address and range with a symbolic name instead of a hex number. If you want to e.g. change the address of ramdisk you need to also edit "softgun/boards/abc.c".

The final section of the configuration file is the remote debugger interface configuration. The "gdebug" section includes the host address and port to listen on.

3 ARM specific changes to Buenos

3.1 Introduction

This section is intended to give an overview of the changes we had to make to port Buenos to the ARM architecture. Our goal was to keep the changes minimal so that it would be possible to merge the ARM port later back to the official Buenos tree in the future.

3.2 Bootup

On MIPS the YAMS emulator parsers the ELF headers of Buenos binary and copies the loadable segments to RAM. Softgun does not have an ELF parser yet so we started to look for other bootup strategies.

A typical bootup approach on ARM is to map a tiny part of flash memory to the beginning of physical memory since that is where the CPU by default starts to execute instructions. We initially wrote a boot loader in assembler that copied rest of the Buenos to RAM, set stack pointer appropriately and jumped to the C code. Pretty soon we noticed that the RAM doesn't contain zeroes and the compiler assumes that ".bss" section should contain zeroes so we modified the boot loader to also zero rest of the RAM.

After a few bootups we realized that Softgun supports only software breakpoints. If we set a breakpoint to RAM before bootup code had copied Buenos to RAM the breakpoint got overwritten. Even worse, when the breakpoint was disabled Softgun restored the original contents of breakpoint location with the original opcode, which was uninitialized data at the time the breakpoint was set. Since we did not have time to improve GDB support in Softgun we were forced to change our bootup strategy. We modified the Makefile to produce "buenos.ramimage" that can just be loaded to "dram0" using Softgun's "-l" command line option. ARM processor is booted from address zero and the very first thing that the image does is to jump to the real kernel code at `0x80010000`.

3.3 Build system

Since our goal was to support both the original MIPS architecture and the ARM architecture we modified Makefile to architecture name as parameter named "ARCH". The default is currently "ARCH=arm" which includes code inside "#ifdef ARCH_ARM" in the build but it is easy to set the default back to"ARCH=mips" if we ever want to merge the changes back to the official Buenos tree.

Linker flags on ARM differ from MIPS in a various ways:

- We noticed that on ARM GCC generates assembler code that calls "__udivsi3" to handle division. In order to make this work we linked buenos against libgcc.
- Since "#ifdef" directives don't work in linker scripts we were also forced to split the linker script "ld.script" into two files ("ld_arm.script" and "ld_mips.script"). These differ in OUTPUT_FORMAT directive and ARM version also embeds jumper code to the beginning of boot image.
- Since interrupt vectors on ARM reside in the beginning of physical memory the kernel mode also maps the beginning of virtual memory directly there. We moved the load address of userland binaries from 0x0 to 0x30000000 for clarity. Later we noticed that ARM also supports "high" interrupt vectors that can reside elsewhere in the address space. Using "high" interrupts

would be useful to catch NULL pointer accesses but this is currently not done.

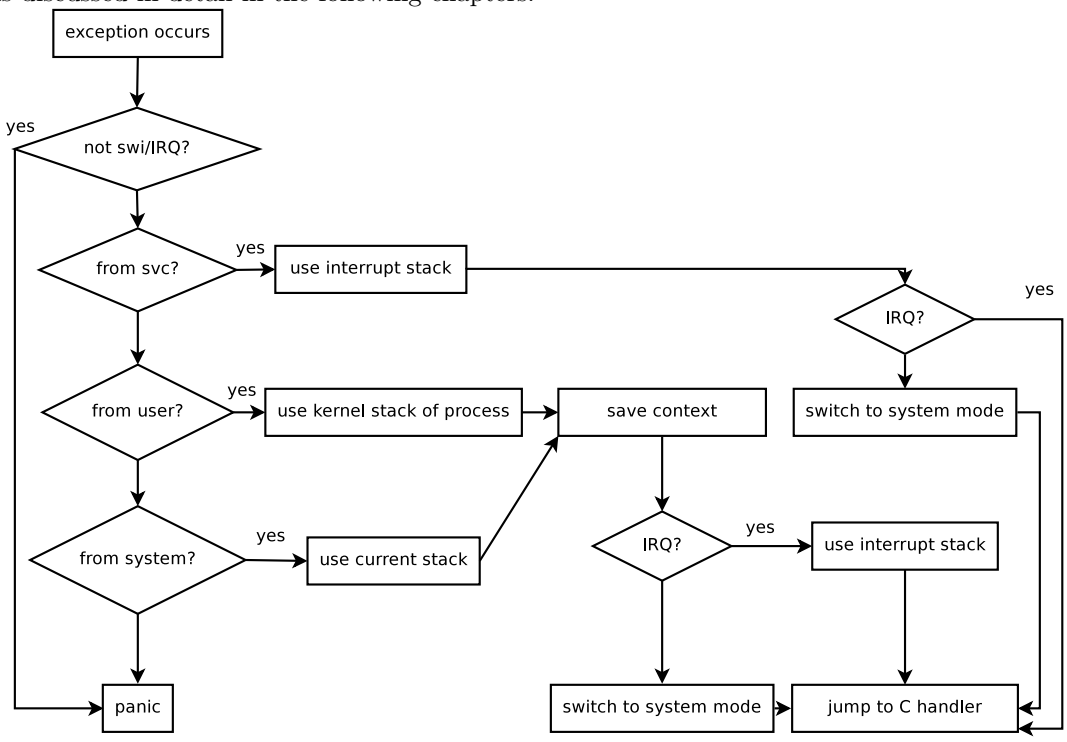
- For some reason GNU ld generated only one program header with "rwx" permissions on ARM. To get separate read-only and read-write segments on ARM we had to modify the userland linker script to explicitly specify two program headers.

3.4 Context switch

3.4.1 Introduction

Context switch is probably the most complex subsystem of Buenos. When the processor raises an exception it branches to a piece of code called interrupt vector area (IVA) and uses type of the exception as an offset. Just like on MIPS Buenos copies code to handle these exceptions to IVA using memorycopy(). This code just branches back to real handler routines that are not in the IVA. Since the branch is not expressable using a 26-bit immediate the code has to load the branch address first from memory to register. On ARM the only usable register at this point is unintuitively the stack pointer (sp).

The diagram above gives an overview of what happens during context switch. It is discussed in detail in the following chapters.



3.4.2 ARM processor modes

On MIPS the EXL bit distinguishes between privileged and unprivileged processor modes. On ARM the situation is more complex than that: the processor has seven different processor modes. Refer to the table 2-1 in [1] for a complete list. In Buenos we use four of the modes:

- User mode for userland programs
- System mode for regular kernel code
- IRQ mode for handling IRQs
- Supervisor mode (SVC) for booting and handling context switch

Of the other three modes, the undefined mode can currently be entered if the processor encounters an undefined instruction, and the abort mode upon illegal memory access. These exceptions are not handled properly, and will result in kernel panic. The FIQ mode for processing Fast IRQs cannot be entered, since the ABC system cannot generate Fast IRQs.

The user and system modes have access to 16 general purpose registers, and the Current Process Status Register (CPSR), which also determines the current mode. The other modes each have three additional registers. Stack pointer (SP) and link register (LR) shadow the corresponding user/system mode registers, and SPSR stores the mode the processor was in when the mode was entered. The Fast IRQ mode has even more additional registers, but since the mode is not used, they are not covered here.

3.4.3 Handling the first exception

Initially the system boots in SVC mode. When virtual memory and threading subsystems have been initialized the main() function calls thread_switch() which issues the "swi 0" instruction that causes a software interrupt. As a result the processor saves current state (this is not important, since we are not going to return to main()) and jumps to 0x00000008. Note that currently there is a race condition: a timer interrupt might get triggered before main() manages to issue "swi 0" but this should not be a problem since also that patch should properly switch to the initial thread.

The handler code first saves the first four registers to a scratch memory area to gain some working space. This is not needed on MIPS, since MIPS has two registers (k0 and k1) reserved for kernel code. Next the handler code checks the mode from which we entered SVC mode, and notices it was also SVC. This is a special case that only happens at bootup, at the first thread_switch() call.

In this special case, interrupt stack is selected. The previous state is not saved, and instead the handler code directly switches to system mode and calls scheduler_schedule. The scheduler then chooses to switch to init_startup_thread since it's the only non-idle thread in thread_able. When the C function returns the assembler context switch code figures out where the context to restore is

(by reading scheduler_current_thread[_FETCH_CPU_NUM]). It then switches to SVC mode so that it can use the special form (marked with caret in assembler syntax) of "ldmia" instruction which loads saved register contents from memory into user/system mode registers. Finally it restores user-mode PC from SVC-mode LR using movs, which also copies SPSR to CPSR thus restoring processor mode and completing context restore.

3.4.4 Handling system mode exceptions

After the first context switch we are in init_startup_thread() in system mode and using the kernel stack of the thread. If the C code creates a new thread (for example networking code does this) it may switch to this thread immediately using thread_switch() that causes an exception using "swi 0". The context switch handler then sees that the exception occured in kernel space and can thus use kernel stack of the current thread to save context. Since the exception was not an IRQ it switches to system mode and calls the C handler which schedules a new thread. Again when the C handler returns the context swith code restores this new thread.

If the exception was caused by timer and not by "swi 0" the context switch code is entered in IRQ mode instead of SVC mode and the handler uses the interrupt stack of the processor (although we only have one processor) for the C code. The interrupt stack is used for handling IRQs mainly because this was done on MIPS. The most obvious reason to use a separate interrupt stack on MIPS is to allow a thread to continue on a different CPU while the interrupt that interrupted it is being processed. Since there are no multiple CPUs on ARM (at the moment, at least), this problem does not exist here.

3.4.5 Handling user mode exceptions

When a userland process wishes to do a syscall it issues the "swi 0" instruction that causes a software interrupt. The handler code sees (from SPSR) that the previous mode was user mode and can thus use the kernel stack of the current thread for saving context. Just like in the kernel exception case it calls the C handler after it has switched to system mode. When interrupt_handle() notices that the interrupt was caused by a syscall it calls syscall_handle() that can access the syscall arguments through saved context and write the return value of the syscall there as well. When the C handler returns the assembler code switches back to the userland process. Note that syscalls are handled with interrupts enabled so a timer interrupt can interrupt a system call handler!

3.5 Virtual memory

(WORK IN PROGRESS)

3.5.1 Introduction

In AT91RM9200 board Translation Lookaside Buffer (TLB) is managed by MMU and not by the operating system. This means that instead of generating address fault when an address is not found in TLB, the MMU itself traverses the page table level to find the address. TLB must be flushed if pagetables are changed. ARM MMU has two page table levels, providing mappings with granularities ranging from 1 MiB to 1 KiB. This allows a rather complex system, but only 1 MiB sections and 4 KiB coarse small pages are used in Buenos. Coarse small pages map 4 KiB sequences of memory divided to smaller 1 KiB subpages for access privileges. In Buenos all 4 sub-pages of a small coarse page always have the same access privileges.

3.5.2 Mapping memory

Since the TLB is handled by the CPU, it needs to know where the level 1 translation table is. Buenos uses a single level 1 translation table at memory address 0x4000. When virtual memory is enabled during the bootup process, several kernel-accessible mappings are added directly to the level 1 translation table as 1 MiB section mappings (refer to table 3 for exact mappings). The virtual addresses from `0x30000000` to `0x7fffffff` are reserved for user processes, and are mapped separately for each process whenever the process is scheduled. User processes map virtual memory in 4 KiB pages.

Because the ARM MMU only offers 16 protection domains, either all processes cannot get their own protection domain, or the number of processes must be limited to 16. We currently only use one domain for all user processes, which means that we have to unmap all the pages of the previous process before we can map in the new process when switching processes.

There is a possibility for performance improvement here: we could dynamically assign protection domains to processes, and thus avoid having to unmap existing process if the new process already had a domain assigned to it. However, as mapping a process in or out takes only one memory write for each level 2 translation table of the process (each of which can address one sequential megabyte of memory), and for our test programs there are usually only three sections: program code, stack and heap, we decided not to pursue the dynamic assignment for now.

3.6 Drivers

3.6.1 Introduction

This section documents changes to drivers. In general these fall into two categories: either we added a new device or we failed to port a an existing driver to work on ARM. All drivers that have not been mentioned were not modified and should work on both MIPS and ARM.

3.6.2 Flash and ramdisk devices

ARM port of Buenos supports AMD Flash device and a simple ramdisk instead of YAMS disk device. This is because flash devices are much more common in ARM world and also there is also no support for YAMS disk devices in Softgun. As explained in the ABC board documentation writing to the flash is not very efficient so Buenos supports also a simple RAM disk so that we were actually able to run the filesystem tests in a reasonable time. We also modified VFS so that it tries to mount not only DISK devices but also FLASH and RAMDISK devices on bootup. Flash and ramdisk drivers in "drivers/amdflash.c" and "drivers/ramdisk.c" respectively.

3.6.3 Timer

MIPS supports a timer that generates an interrupt after a certain number of CPU clocks have passed. The timer on AT91RM9200 is different. It does have a System Timer (ST) device that uses Slow Clock (32768 Hz). ST supports Periodic Interval Timer (PIT) which allows interrupting at most on every Slow Clock resulting in 32768 Hz timer frequency. The timing is thus not as exact as compared to MIPS. In Buenos, CONFIG_TIMER_FREQUENCY controls PIT interval (100 Hz by default). ST has also Real-time Timer (RTT) to calculate elapsed time. In Buenos RTT is set to increment counter every 32 Slow Clocks resulting RTT to run on 1024 Hz. This allows counting milliseconds. ST initialization and handling is in "drivers/at91.c".

3.6.4 TTY

We did not implement full support for "yamst" style terminal emulation in Softgun. The current implementation supports only polling I/O so we had to change also the Buenos TTY driver to use only polling on ARM.

3.6.5 Processor sleeping

Modern operating systems always put the processor to sleep when there are no active threads. We had planned to implement this also in Buenos but it fails to put the processor to sleep in idle thread when running in ARM mode because a lack of support from Softgun. In AT91RM9200 board sleeping is done by telling the PMC (Power Management Controller) to stop giving clock to CPU and doesn't seem to have any effect in Softgun.

3.7 Test cases

We used homework written for the "Operating Systems Project" course as test cases for our port. This was a compromise, we would have liked to use a dedicated test suite but we really didn't have time to write one from scratch. Fortunately for us, we were already familiar with the test cases since two Buenos-ARM project members were also on the OS project course at the same time.

The Buenos-ARM tar ball currently contains solutions to phases 1 and 2 since later phases do not work yet on ARM.

The first phase covers locks and conditional variables and some simple kernel programming exercises. The second phase is mainly about implementing system calls. Solutions to both phases can have been carefully marked with "#ifdef CHANGED_n" where n is the phase number to make it possible to remove the homework solutions. The third phase was about filesystems and should not really have ARM related issues. The fourth phase covered paging (swapping) and could probably be ported to work on ARM. To get more information on the homework solutions consult the directory buenos/doc. Note that since pagetable format is different on ARM we had to make some changes to the test cases to make them work on ARM but the original homework documentation still assumes MIPS.

3.8 Miscellaneous fixes

This section documents miscellaneous changes that don't fit well under other sections:

- The ELF magic number of ARM architecture was added to ELF parser. We could not find this number from the ELF specification even though MIPS was mentioned there so we just assumed it would be "40" since that is what GCC generates.
- There were various endianness issues in TFS, the trivial file system. It copied buffers from disk to memory and then tried to access the data through C structures. We added byte-swap functions to lib/libc.c and fixed the endianness issues in TFS to be able to run our phase 2 test cases that depended on a working filesystem.
- The random number generator rand.S was written in assembler in Buenos. We did not understand the reason behind this and we rewrote the random number generator in C. The new version works on both ARM and MIPS without modifications.

4 Tutorial: working in the softgun environment

4.1 Building a cross compiler and utilities

To compile binaries for ARM on a non-ARM system you need a cross compiler. The upstream Buenos project gives useful instructions on building a cross compiler for MIPS at http://www.niksula.hut.fi/~buenos/cross-compiler.html. Luckily for us we only needed very small changes:

- use target "arm-elf" instead of "mips-elf" (Note that "arm-elf" is little-endian)

- build also GDB since softgun supports it

The following listing shows the exact commands we used to build binutils, GCC and GDB. The steps are available also as doc/compile_utils.sh and assume POSIX-compatible sh and that "make" is GNU make.

```
export PREFIX=$HOME/arm-gcc

wget http://www.niksula.hut.fi/~buenos/misc/binutils-2.16.1.tar.gz
wget http://www.niksula.hut.fi/~buenos/misc/gcc-core-4.0.2.tar.gz
wget http://ftp.gnu.org/gnu/gdb/gdb-6.6.tar.gz

tar zxf binutils-2.16.1.tar.gz
tar zxf gcc-core-4.0.2.tar.gz
tar zxf gdb-6.6.tar.gz

mkdir build-binutils
cd build-binutils
../binutils-2.16.1/configure --target=arm-elf --prefix=$PREFIX -v
make
make install
cd ..

export PATH=$PREFIX/bin:$PATH

mkdir build-gcc
cd build-gcc
../gcc-4.0.2/configure --with-gnu-ld --with-gnu-as --without-nls --enable-languages=c --
make
make install
cd ..

mkdir build-gdb
cd build-gdb
../gdb-6.6/configure --target=arm-elf --prefix=$PREFIX
make all
make install
cd ..
```

4.2 Building buenos, softgun and userland

If you followed the instructions in the previous section you should now have "arm-elf-ld" and "arm-elf-gcc" in your PATH. Now, extract the buenos-arm distribution tarball and move to "buenos" directory. In this directory a simple

```
make ARCH=arm
```

should compile buenos and produce "buenos.ramimage". If you still have MIPS cross compilation tools in PATH you can try

```
make clean
make ARCH=mips
```

but this is not guaranteed work reliably yet (some changes might have broken MIPS parts).

The buenos-arm project ships its own version of the softgun ARM emulator since we have added new devices and a new board (the ABC board). To build our modified softgun simply issue

```
make softgun
```

and verify that ../softgun/softgun was created. Finally, you can build userland binaries with

```
make ARCH=arm -C tests
```

and move them to a filesystem image just like with original Buenos. In particular, the testcases we used can be copied to "store.file" image using

```
tests/update.sh
```

4.3 Booting buenos

A normal way to boot Buenos on ARM is

```
../softgun/softgun -c softgun/abc.sg -l ram buenos.ramimage \
-l ramdisk store.file -k ‘‘initprog=[vol0]sh’’
```

which asks softgun to

- load board specific configuration parameters from softgun/abc.sg
- read contents of "buenos.ramimage" and place into a memory region named "ram" (completely fills the main DRAM chip)
- read contents of the filesystem image "store.file" to region named "ramdisk" (extra DRAM chip we use as a ramdisk).
- set "initprog=[vol0]sh" to be the Buenos kernel command line in the Yams compatible command line meta device (upstream softgun obviously does not support this).

If everything went well you should see the following messages in stdout:

```
LCA ‘‘ram’’ ‘‘buenos.ramimage’’
LCA ‘‘ramdisk’’ ‘‘store.file’’
Configuration file ‘‘softgun/abc.sg’’ loaded
IO-Thread started
```

```
Loading ABC Board module
MemMap and IO-Handler Hash initialized
Creating ARM9 CPU with clock 18432000 HZ
- Instruction decoder Initialized:  63424 2112 0 0 0 0 0 0 0
- Register Pointers initialized
GDB server listening on host ‘‘127.0.0.1’’ port 4711
- Create MMU Coprocessor
MMU: Byteorder is now LE
AT91RM9200 MC ‘‘mc’’ created
Flash bank ‘‘flash0’’ type AM29LV128ML Chips 1 writebuf 32
DRAM bank ‘‘dram0’’ with  size 1024kB
AT91RM9200 AIC ‘‘aic’’ created
AT91RM9200 Power Management Controller created
YamsDescriptorDev ‘‘desc’’ created
YamsTTY Device ‘‘tty’’ created
YamsDescriptorDev_Register type = 201 vendor = tty
YamsMemInfo Device ‘‘meminfo’’ created
YamsDescriptorDev_Register type = 101 vendor = meminfo
YamsShutdown Device ‘‘shutdown’’ created
YamsDescriptorDev_Register type = 103 vendor = shutdown
YamsShutdown Device ‘‘cpustatus’’ created
YamsDescriptorDev_Register type = c00 vendor = cpustatus
YamsDescriptorDev_Register type = 302 vendor = flash
DRAM bank ‘‘dram1’’ with  size 2048kB
YamsDescriptorDev_Register type = 303 vendor = ramdisk
desc[0] type 201 iobase ffff8000 iolen 12 irqcount 0 vendor tty reserved 0 0
desc[1] type 101 iobase ffff800c iolen 4 irqcount 0 vendor meminfo reserved 0 0
desc[2] type 103 iobase ffff8010 iolen 4 irqcount 0 vendor shutdown reserved 0 0
desc[3] type c00 iobase ffff8014 iolen 8 irqcount 0 vendor cpustatu reserved 0 0
desc[4] type 302 iobase 10000000 iolen 1048576 irqcount 0 vendor flash reserved 0 0
desc[5] type 303 iobase 90000000 iolen 2097152 irqcount 0 vendor ramdisk reserved 0 0
YamsHelper Device ‘‘helper’’ created
YamsCommandline Device ‘‘commandline’’ created
Loading buenos.ramimage to 0x80000000 flags 0
Loading store.file to 0x90000000 flags 0
Poll detector Sensivity 10
Starting CPU at 80010000
BUENOS is a University Educational Nutshell Operating System
============================================================

Copyright (C) 2003-2006  Juha Aatrokoski, Timo Lilja,
  Leena Salmela, Teemu Takanen, Aleksi Virtanen
See the file COPYING for licensing details.

Initializing memory allocation system
Kernel size is 0x000419ac (268716) bytes
```

```
System memory size is 0x000c8000 (819200) bytes
Reading boot arguments
Detected 1 CPUs
Initializing AT91RM9200 ARM-board
AT91Pmc: Enabled master clock
Main clock frequency is 18 MHz (18432000 Hz)
Initializing interrupt handling
Initializing threading system
Initializing sleep queue
Initializing semaphores
Initializing locks and condition variables
Initializing process table
Initializing device drivers
Device: Type 0x201 at 0xffff8000 irq 0x0 driver 'Console'
Device: Type 0x101 at 0xffff800c irq 0x0 driver 'System memory information'
Device: Type 0x103 at 0xffff8010 irq 0x0 driver 'System shutdown'
Device: Type 0xc00 at 0xffff8014 irq 0x0 driver 'CPU status'
Device: Type 0x302 at 0x10000000 irq 0x0 driver 'Flash'
amdflash_init: base 10000000 len 100000
Device: Type 0x303 at 0x90000000 irq 0x0 driver 'Ramdisk'
ramdisk_init: base 90000000 len 200000
Initializing virtual filesystem
VFS: Max filesystems: 8, Max open files: 512
Initializing scheduler
Initializing virtual memory
Pagepool: Found 200 pages of size 4096
Pagepool: Static allocation for kernel: 100 pages
Creating initialization thread
Starting threading system and SMP
Mounting filesystems
VFS: No filesystem was found on block device 0x10000000
VFS: TFS initialized on disk at 0x90000000
VFS: Mounted filesystem volume [vol0]
Initializing networking
Starting initial program '[vol0]sh'
Waiting for init to exit
0 $
```

Note that since we don't yet support yamst input is line buffered. To exit the emulator simply hit ctrl-c or run the halt program:

```
0 $ halt
Kernel: System shutdown started...
VFS: Entering forceful unmount of all filesystems.
VFS: Forcefully unmounting volume [vol0]
Kernel: System shutdown complete, powering off
shutdown_write value = 0badf00d
```

4.4 Debugging

4.4.1 Using GDB

Softgun has primitive support for GDB remote debugging protocol over a TCP socket. During our porting efforts we made extensive use of GDB and especially GUD (Grand Unified Debugger: an Emacs frontend for GDB) and eventually learned to avoid bugs in softgun's GDB support.

To get started, copy the following configuration options to your " /.gdbinit":

```
handle SIGPWR noprint nostop
handle SIGXCPU noprint nostop
handle SIGUSR1 noprint nostop

define xsi
si
x/4i $pc
end

define bb
target remote 0:4711
end

set remotetimeout 0
```

and then boot buenos using softgun with the "-d" option to have softgun wait for GDB before executing any instructions. Then run "arm-elf-gdb buenos" and type "bb" (short for "boot buenos"). GDB should print

```
_start () at init/_boot.S:81
81              ldr r0, .stack_address
Current language:  auto; currently asm
```

since we are in the beginning of bootup assembler code. Let's place a breakpoint at tfs_read and let softgun continue

```
(gdb) b tfs_read
Breakpoint 1 at 0x80021c24: file fs/tfs.c, line 540.
(gdb) c
Continuing.
```

After a short while we have arrived at tfs_read and can look at the backtrace

```
Program received signal SIGINT, Interrupt.
tfs_read (fs=0x80064000, fileid=443, buffer=0x8002d54c, bufsize=52, offset=0) at fs/tfs.
540         tfs_t *tfs = (tfs_t *)fs->internal;
Current language:  auto; currently c
(gdb) bt
#0  tfs_read (fs=0x80064000, fileid=443, buffer=0x8002d54c, bufsize=52, offset=0) at fs/
```

```
#1  0x80020628 in vfs_read (file=0, buffer=0x8002d54c, bufsize=52) at fs/vfs.c:730
#2  0x8001a238 in elf_parse_header (elf=0x8002d5a4, file=0) at proc/elf.c:70
#3  0x8001ca70 in process_exec (exec_i=2147817909) at proc/process_table.c:305
#4  0x800119dc in thread_goto_userland (usercontext=0x0) at kernel/thread.c:329
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

Note that backtrace does not extend to userland parts. Also note that softgun prints "softgun gdb interface does not support breakpoints" multiple times since softgun GDB support really is not at all complete. Because of these bugs we recommend that you limit your GDB usage to

- commands that only read or write memory (print, set)
- simple breakpoints (break and tbreak). Note that softgun breakpoints are software breakpoints: softgun overwrites address with a break instruction and tells gdb when it is executed. This does not work well with self-modifying code. In particular, do not ever place breakpoints to interrupt handlers in the beginning of RAM unless you know what you are doing!)
- simple "continue"
- single stepping with (stepi). In particular, the "xsi" macro is useful for stepping over assembler code. It steps one instruction and then disassembles the following four instructions.

4.4.2 Other debugging strategies

The syscall implementation we used as a testcase supports the boot option "strace" that causes it to print all executed system calls and their return values. For example

```
../softgun/softgun -c softgun/abc.sg -l ram buenos.ramimage \
-l ramdisk store.file -k ``initprog=[vol0]sh strace''
```

should print

```
syscall_WRITE(1, 2147478900, 4)
syscall_WRITE(1, 2147478900, 4) = 4
```

since the shell printed four characters ("0 $" and newline) to stdout (fd 1) and writing all characters succeeded.

It is possible to place custom debugging code at the function debug_print_instruction() in softgun's arm9cpu.c. When we were trying to fix subtle virtual memory bugs we printed the value of the program counter and contents of a few well-chosen memory addresses on every clock cycle to a file for later analysis. We could then use e.g.

```
$ arm-elf-addr2line -e buenos 0x80010400
kernel/cswitch_arm.S:345
```

to see line number information.

5 Appendix A: Educational story on debugging Buenos

In the demo session Buenos paniced unexpectedly. By trying earlier SVN versions we tracked the problem down to a commit that changed timer frequency from 1 tick/s to 100 ticks/s. The bug occured randomly so we suspected a race condition.

We modified arm9cpu.c to print

```
fprintf(stderr, ‘‘%08x: %08x: %s\n’’, REG_PC - 4, icode, instr->name);
```

and then set “armdebug: 1” in softgun/abc.sg so that we could run

```
../softgun/softgun -c softgun/abc.sg -l ram buenos.ramimage -l ramdisk store.file -k ‘‘i
```

to collect an execution trace.

We ran “arm-elf-objdump buenos — grep kernel_panic” to see that the address of kernel_panic function was 80010b54 and removed all addresses after this point from the trace. End of the trace then looked like

```
80016200: e5c23000: strb C: inside memoryset()
00000010: ea000008: bbl  C: data abort handler
00000038: e59fd02c: ldr
0000003c: e12fff1d: blx2,bx
80010138: e59f0028: ldr
8001013c: e1a0100e: mov
80010140: e59f20b4: ldr
80010144: ea000282: bbl
80010b54: e1a0c00d: mov  C: _kernel_panic
```

where “C: “ denotes a comment we added later.

Then we started softgun with

```
../softgun/softgun -c softgun/abc.sg -l ram buenos.ramimage -l ramdisk store.file -k ‘‘i
```

and connected to it with arm-elf-gdb. We first set a breakpoint at “memoryset” and let Buenos continue bootup to that point. At this point the interrupt vector has been copied into its place, and we can add soft breakpoints there. This was necessary because we wanted to break at data abort handler in the interrupt vector. The command we used to add the breakpoint was “break *0x00000010”.

With a debugger we saw that the offending instruction just before the jump to the data abort handler was

```
(gdb) x/i 0x80016200
0x80016200 <memoryset+72>:      strb    r3, [r2]
```

and register r2 contained value 0x40000000 which is where userland programs get their argv. Memoryset was called from process_exec, which seemed to create

all the appropriate mappings. Also, since the code worked with lower timer interrupt frequency, this was clearly a race condition and not a systematic error in process_exec. So why weren't the mappings there, then?

Next we added debug print to the main interrupt handler, which showed the cause of each interrupt as they happened. We also peppered process_exec with debug prints, telling what it was doing so that we could see where interrupts happened in relation to code. This was not enough to solve the mystery. Sure, there were timer interrupts during mapping of memory, but that should not have broken anything, since the pagetable of the new process was only activated later.

To solve the mystery, we added debug prints to vm_activate_pagetable to see when pagetables actually changed. This revealed that actually the pagetable was activated earlier than we thought, and we wrote the following hypothesis:

```
Shell does exec for echo. A new thread is started - it runs
process_exec. Process_exec creates a new pagetable for itself, and
also sets it to process_table. Then process_exec starts mapping things
into the new pagetable. At some point while mappings are being
performed, timer interrupt hits and kernel switches to some other
thread. Then it switches back to echo, and since it has a pagetable in
process_table, its pagetable is activated. Process_exec continues
mapping new entries to its process_table-pagetable. When it finishes
mapping things, it activates the process_table pagetable, which is now
ready to be used. However, since Echo's process_table-pagetable is
already active, vm_activate_pagetable does not do anything, and Bad
Things Happen. In this case command line arguments were mapped only in
process_table-pagetable, not in the global page table, and trying to
initialize them caused data abort.
```

This was what actually happened, and we fixed the issue as follows:

- We added dirty flag to the pagetable structure on ARM, which is set by vm_map and checked by vm_activate_pagetable in addition to checking if the pagetable is the same that is already active. If the pagetable is dirty, it is activated and the flag is cleared.
- We also disabled interrupts during vm_map, since the page table updating algorithm may be in an inconsistent state while it is updated. This does not happen on MIPS. On MIPS there is no race because of valid_count, which prevents access from tlb_fill to the entry that is being mapped. On MIPS vm_unmap complicates this more, but it is only accessed while holding global locks (in our MIPS Buenos, anyway), but it is not implemented on ARM at all.

[3]

References

[1] ARM Limited. ARM Architecture Reference Manual, 2000.

[2] Atmel corporation. AT91RM9200:: ARM920T-based Microcontroller, 2006.

[3] example author. Example title.