The Brown Simulator (version 2)

Jason Lango
Keith Adams
Michael Castelle
David Powell

Spring 1998

Revision

$Id: intro.tex,v 1.3 1998/09/28 10:37:36 jal Exp $
$Id: vm.tex,v 1.14 1998/09/28 10:37:37 jal Exp $
$Id: cpu.tex,v 1.20 1998/09/28 10:37:34 jal Exp $
$Id: format.tex,v 1.10 1998/09/25 16:46:01 jal Exp $
$Id: dev.tex,v 1.13 1998/09/28 10:37:34 jal Exp $
$Id: intr.tex,v 1.13 1998/09/28 10:37:36 jal Exp $
$Id: code.tex,v 1.3 1998/09/25 16:45:59 jal Exp $
$Id: boot.tex,v 1.14 1998/09/28 10:37:33 jal Exp $

Contents

Chapter 1

Introduction

1.1 Overview

The Brown Simulator is a program which simulates a machine upon which students and researchers can implement and test prototype operating systems.

The key goal of the Brown Simulator is to provide an environment in which a student or researcher may implement, compile, test, and debug the prototype operating system using the native operating system, hardware, and software tools on their desktop. The point is to explore operating systems concepts, but not to test an operating system which runs on an actual machine.

As such, the Brown Simulator does not simulate actual machine code, but is merely a library of functions, callable from C or C++ code, which provide the user with functionality which would be expected from actual machine hardware, such as privileged processor registers, device control registers, virtual memory hardware interfaces, etc. The actual code for the student's operating system is loaded by the Simulator as an executable shared library on the native operating system. Section 1.2 contains a technical overview of the simulated machine.

Since the Simulator does some pretty gruesome things in order to provide the illusion of virtual memory, the student's operating system should not make use of native operating system calls. Particularly, the only library and system calls supported are the standard string and memory functions (e.g. strcpy(), strcat(), memcpy(), etc.), printf(), sprintf() (and friends), and alloca(). All other library and system calls should be avoided (and will result in undefined behavior). Particularly, calls to malloc() and free() will not work under the Brown Simulator (though it would be cheating to use the native operating system's allocation routines anyway :-).

1.2 Machine model

The Brown Simulator allows the configuration of the simulated machine to be specified at runtime and provides functions so that the student's operating system can probe the machine to determine how many processors, devices, etc. are installed in the simulated machine. Chapter 2 talks about the process of booting the simulated machine and detecting the hardware configuration.

Each simulated processor has a set of registers which control how the processor behaves, such as whether interrupts can occur, whether the processor is executing in user or kernel mode, etc. Chapter 3 talks about the simulated processor

Sometimes an external event occurs that the processor must handle immediately, such as when the user presses a key. Device generated events are generally called *interrupts*. The code that the processor is executing can also generate certain internal events, or *exceptions*, such as referencing invalid memory, that the processor must deal with immediately. User code which needs to request a service from the kernel can generate a special event, known as a *trap*. Traps, exceptions, interrupts are covered in Chapter 4.

While the processor is executing a user program, the kernel may wish to present a *virtual memory* abstraction to the user process. The simulated hardware provides a mechanism wherein the kernel may request that read or write operations at certain locations in memory will result in an exception being generated, so that the kernel (which handles the exception) can control what physical memory the process has access to. Virtual memory is described in Chapter 5.

Devices, such as several disks or terminals, may be connected to your simulated machine. The simulator provides a mechanism where the user may specify the devices which are connected to the simulated machine in a configuration file. The kernel may probe to find out which devices are connected at run-time. Each device may use either a *direct memory access (DMA)* or *programmed I/O (PIO)* model of interaction. Chapter 6 describes configuring, probing, and using the simulated devices.

Chapter 2

Booting the simulator

2.1 Running a kernel on the Brown Simulator

The **simrun** executable is used to run your kernel on the Brown Simulator. The kernel image is passed as a command-line parameter to **simrun**.

The kernel image should be a shared object (or shared library) which has at least one global function: `kern_boot()`

2.1.1 simrun options

- -m *memory-in-Mb*
  Number of megabytes of physical memory for the machine.

- -p *ncpus*
  Number of simulated processors for the machine.

- -d *device-config-file*
  Configuration file, containing a list of devices and configuration options[1]. If this option is not specified on the command-line, a default file named **simconfig** is searched for in the config file search path `SIM_CONFIG_PATH`.

- -s *simulator-library*
  Specify a different simulator runtime library. Mostly for debugging purposes.

- -D *debug-flags*
  Specify a comma-separated list of debugging flags to use with the simulator. These flags will cause the simulator to print messages to standard-output. The list of currently supported flags can be viewed by typing `simrun -h`.

2.1.2 Simulator Environment Variables

The following environment variables alter the behavior of the Brown Simulator. Environment variable defaults, if any, can be displayed by running "simrun -h".

[1] See section 6.3.

`SIM_DEV_PATH` A colon-separated list of directories to search for simulator device files and the device server.

`SIM_CONFIG_PATH` A colon-separated list of directories to search for the default simulator configuration file, **simconfig**.

`SIM_DBG` A comma-separated list of debugging flags, same as the -D option to **simrun**.

2.2 Starting up...

`kern_boot()` will be the first function called in your kernel. It will be called on a special stack, which will be discarded once the first call to `SIM_set_context()` is issued[2]. Within this function, the kernel should set up the *trap vector block*[3] (TVB), including all interrupt and exception handlers.

At the point that `kern_boot()` is called, each processor will be at HIGH ipl (meaning that all interrupts are masked).

2.2.1 Multiple processors

If the machine is configured to contain multiple processors, `kern_boot()` will be called in parallel on each processor, with the processor number passed as a parameter.

2.3 Machine Configuration

At boot time, the kernel must determine certain configuration parameters for the simulated machine, such as the number of processors installed, the amount of physical memory available, the attached devices, etc.

`SIM_num_cpus()` returns the number of processors installed in the simulated machine.

`SIM_num_devs()` returns the number of devices configured for the machine. See section 6.6 for more information on detecting the parameters of configured devices attached to the machine.

`SIM_num_phys_pages()` returns the number of pages of physical memory installed in the simulated machine. See section 5.2 for more information on the usage of physical memory and chapter 5 for a general discussion of the simulator virtual memory model.

[2]See section 3.2.2.
[3]See section 3.3.2.

Chapter 3

The Processor

3.1 Modes

The processor has two major modes of operation: *user mode* and *supervisor mode*.

3.1.1 Supervisor mode

When the processor is in supervisor mode, the currently executing program has access to all of the machine's resources, including privileged processor registers. The kernel code executes in supervisor mode.

3.1.2 User mode

When the processor is in user mode, the following restrictions apply:

- Privileged processor registers cannot be read or written. This includes the PIR, PSR, IPL, and TVB.
- Physical memory cannot be read from or written to directly. All memory accesses must go through the TLB (see section 5.4).
- Device registers cannot be read or written.

A program executing in user mode can switch to supervisor mode only by executing a *trap instruction*. The function `SIM_trap()` simulates the trap instruction, causing the processor to switch to supervisor mode and the kernel's trap handler to be called[1]. `SIM_trap()` is the only simulator function which may be called while executing in user mode.

3.2 Processor contexts

A *processor context* can be considered a snapshot of the entire state of the simulated processor. The simulator provides functions for setting up and manipulating processor contexts, switching between processor contexts, etc. Another way of thinking of a processor context is as the snapshot of a *thread of execution*.

[1] See section 4.1.

All of the information for a processor context is encapsulated in the **sim_context_t** structure. The structure contains the following members.

sc_psr The *processor status register* (PSR). See section 3.3.1 for more information on the PSR and its associated values.

sc_ipl The *interrupt priority level* (IPL). See section 3.3.3 for more information on the IPL register and its associated values.

sc_kstack*, *sc_kstack_size The *kernel stack* for that context. See section 3.2.1 for more information on the kernel stack and when it is used.

sc_ucontext The opaque representation of the rest of the processor state.

3.2.1 Creating a context

`SIM_make_context(`*ctx, func, arg1, arg2, stack, stack_size, kstack, kstack_size, psr, ipl, return_ctx*) is used to initialize a **sim_context_t** structure.

The arguments to `SIM_make_context()` are as follows:

ctx Pointer to the context to initialize.

func, arg1, arg2 Pointer to the function which will be executed when this context is first run, via `SIM_set_context()` or `SIM_swap_context()`, and arguments to that function. The function pointer type is **sim_thread_func_t**.

stack, stack_size The stack on which this thread will execute. See the notes below for more information on where the memory for this stack should come from. The range of addresses from stack to stack + stack_size will be used as the stack for this context.

kstack, kstack_size The kernel stack for this thread. See section 3.2.1 for more information on when the kernel stack is used.

psr, ipl Values for the PSR and IPL registers. See section 3.3 for more information on the simulated processor's registers.

return_ctx Pointer to another **sim_context_t** which will automatically be executed when control returns from this processor context, i.e. when the originally executed function returns. This argument can only be specified for supervisor mode contexts (whose PSR has the `SIM_SUPERVISOR_MODE` bit set).

Note that whether the given stack should be in user space or in the physical memory space[2] depends on the value given for the PSR (whether the context will execute in user or supervisor mode). This is not to be confused with the kernel stack given, which is the stack on which traps will be handled. For a context which is executing in kernel mode, it is valid for both stacks to refer to the same memory (since the kernel mode code will not be executing any traps).

If the given stack is in user space, the call to `SIM_make_context()` will result in accesses to user space and possibly associated TLB exceptions[3].

[2] See chapter 5 for more information on virtual memory and address spaces.
[3] See section 5.4.2.

3.2.2 Context switching

`SIM_swap_context(`*old_ctx, new_ctx*) is used to atomically switch between two thread contexts. It will save the current machine context and start executing the new context.

`SIM_set_context(`*ctx*) immediately sets the current processor's state to the contents of the given **sim_context_t**. This function is rarely used, as `SIM_swap_context()` is the preferred method of switching contexts, although there are two special cases in which one *must* use `SIM_set_context()`:

1. When the machine first boots, the kernel will wish to start the first thread of control without saving the initial context. `SIM_set_context()` comes in handy in this case.

2. If the kernel wishes to preemptively switch threads while in the clock interrupt handler, `SIM_set_context()` must be used. The reason for this is that you don't want to save the current context when switching to the next, since that context is an *interrupt* context. The context which must be saved is that which the clock is interrupting. There is some example code below which illustrates how to use `SIM_set_context()` in an interrupt handler.

3.2.3 Examples

1. Preemptively switch between contexts during a clock interrupt. `next` and `current` are of type **sim_context_t**. *Note that the kernel is saving the context which is being interrupted,* **not** *the current interrupt context.*

```
void clock_intr(sim_device_t dev, sim_context_t *ctx)
{
        current = *ctx;

        /* Other context-switch code here, e.g.
         * SIM_tlb_flush() ...
         */

        SIM_set_context(&next);
}
```

3.3 Registers

3.3.1 PSR - Processor status register

The *processor status register* contains general information about the processor.

The only information currently supported is the mode of execution of the processor, which can be gotten at through the mode bits specified with the mask `SIM_PSR_MODE` and which will be one of `SIM_USER_MODE` or `SIM_SUPERVISOR_MODE`. The macros `SIM_PSR_USER_MODE()` and `SIM_PSR_SUPERVISOR_MODE()` test for these two conditions, given a PSR value.

The context switching functions `SIM_set_context()` and `SIM_swap_context()` might result in changes to the PSR, and thus to the processor mode.

3.3.2 TVB - Trap/interrupt vector block register

The *trap vector block* (TVB) register points to a structure in physical memory which in turn contains a set of function pointers (vectors) used by the kernel to handle exceptional conditions, interrupts, traps, etc.

The TVB contains the following members:

st_intr Points to a function which will handle device interrupts. Interrupt handlers are described in section 4.3.

st_ipi Points to a function which will handle interprocessor interrupts.

st_clock Points to a function which will handle clock interrupts, on a per-processor basis. The processor clock and associated interrupts are described in section 3.3.5.

st_illinst Points to a function which will handle illegal instruction exceptions. Illegal instruction exceptions are described in section 4.2.3.

st_trap Points to a function which will handle traps. Trap handlers are described in section 4.1.

st_break Points to a function which will handle breakpoint instructions placed in the code by an interactive debugger. Don't worry about this if you're not implementing a debugging system.

st_fpe Points to a function which will handle floating point exceptions, such as divide by zero, etc.

st_buserr Points to a function which will handle misaligned memory accesses (e.g. bus error or SIGBUS).

st_addrerr Points to a function which will handle invalid memory accesses (i.e. user accesses to addresses outside of user space; see chapter 5).

st_tlbrefill*, *st_tlbmodified*, and *st_tlbinvalid Point to functions which will handle tlb-related exceptions. The TLB is described in section 5.4 and its exceptions are described in section 5.4.2.

`SIM_set_tvb()` is used to set the value of the TVB register.

3.3.3 IPL - Interrupt priority level register

The *interrupt priority level* (IPL) register controls the set of interrupts which will be handled at a particular point in time.

When the IPL is set to a particular level, the processor will queue interrupts at that level and below. The handling of queued interrupts is deferred until a time when the IPL is later lowered, at which point those queued interrupts will be delivered in priority order.

The value of the interrupt priority level register is retrieved using the function `SIM_get_ipl()`. The interrupt priority level is set using `SIM_set_ipl()`. *Note that interrupt handlers*[4] *may be called as a direct result of lowering the IPL.*

[4] See section 4.3.

3.3.4 PIR - Pending interrupt register

The pending interrupt register marks which interrupts are queued to be executed when the IPL is lowered.

Bit N in the value of the PSR corresponds to IPL N, so if bit 6 is set then there is an interrupt at IPL 6 waiting to be delivered.

The value of the pending interrupt register is retrieved using the function `SIM_get_pir()`.

3.3.5 CLK - Clock register

Each processor has a real-time clock, which can be configured to generate interrupts at periodic time intervals. For more information on handling interrupts, see section 4.3. *Note that because the clock is not really a device, the* **sim_device_t** *passed to the interrupt handler will be* `NULL`.

The value of the clock register specifies the time interval (in milliseconds) between clock interrupts.

The clock always generates interrupts at ipl level `SIM_CLK_IPL` (usually the highest ipl supported by the machine).

The clock register is set using the function `SIM_set_clk()`.

3.3.6 Examples

1. Execute a critical section of code, disabling all interrupts.

```
int oldipl;
oldipl = SIM_set_ipl(SIM_HIGH_IPL);

        ... critical section code goes here ...

SIM_set_ipl(oldipl);
```

2. Set the clock to fire an interrupt every half second.

```
SIM_set_clk(500);
```

3. Run some code if there's an interrupt pending at IPL 5.

```
if (SIM_get_pir() & (1 << 5)) {
        ... some code ...
}
```

3.4 Multiple processors

3.4.1 Detecting the number of processors

`SIM_num_cpus()` will return the number of configured processors in the machine.

3.4.2 Atomic memory operations

In a multiprocessor kernel, certain sections of kernel code will need to be protected by critical sections and certain variables might need to be atomically updated such that no interleaving of operations between processors will cause the variables to be left in an inconsistent state.

The simulator provides one atomic memory operation, *atomic swap*, upon which others can be built.

`SIM_atomic_swap()` is a function which takes a variable of type **atomic_t** and atomically sets the value of the variable and returns its old value.

3.4.3 Examples

1. Use `SIM_atomic_swap()` to implement a spin lock.

```
typedef atomic_t spinlock_t;

void spin_lock(spinlock_t *lock)
{
        while (SIM_atomic_swap(&lock, 1) != 0)
                ;
}

void spin_unlock(spinlock_t *lock)
{
        *lock = 0;
}
```

Chapter 4

Traps, exceptions, and interrupts

4.1 Traps

When a user-mode program calls `SIM_trap(`*arg1, arg2*`)`, the kernel trap handler will be called with the same arguments passed to the user-mode function and a **sim_context_t*** which points to the context at the time of the trap. This **sim_context_t** will be stored on the kernel stack.

The trap handler will be executed on the current kernel stack.

The trap handler expects an integer return value, which will be the value returned from the user's call to `SIM_trap()`.

4.2 Exceptions

When an exception occurs the processor switches to supervisor mode and control is immediately passed to a particular handler function registered in the *trap vector block* (TVB, section 3.3.2). The handler function will execute on the current kernel stack and will be passed the processor context which was interrupted.

4.2.1 Floating Point Error Exceptions

When program code attempts to perform an illegal operation on floating point numbers, e.g. divide by zero, etc., in user or supervisor mode, a floating point error (FPE) exception will be generated. The *st_fpe* member of the TVB will be used to handle this exception.

4.2.2 Memory Access Exceptions: Bus error and address error

When program code attempts to load or store to an address which is incorrectly aligned, in user or supervisor mode, a bus error is generated. When user-mode program code attempts to load or store to memory outside of user space, an address error is generated. The *st_buserr* and *st_addrerr* members of the TVB will be used (respectively) to handle these exceptions.

4.2.3 Illegal Instruction Exceptions

When the processor attempts to fetch and execute an illegal instruction in user or supervisor mode, an illegal instruction exception will be generated. The *st_illinst* member of the TVB will be used to handle this exception.

4.2.4 TLB-related Exceptions: Refill, modified, and invalid

Programs executing in user mode may generate TLB-related exceptions, which are described in section 5.4.2.

4.3 Interrupts

Interrupts can be generated by devices whether in user or supervisor mode. The kernel may set up one interrupt handler per *interrupt priority level* (IPL) in the TVB (3.3.2).

The particular interrupt handler called will be given as its arguments the device handle[1] (of type **sim_device_t**) and the context[2] (of type **sim_context_t***) which was interrupted.

When an interrupt is delivered, the processor's IPL register is set to the IPL level of the current interrupt (meaning that interrupts at that priority or lower are queued/deferred until the current interrupt handler finishes executing). The kernel may choose to block interrupts temporarily at any point by setting the processor's IPL register[3]. The processor will switch to supervisor mode, if it is currently executing in user mode, and the handler function will execute on the current kernel stack.

[1] See chapter 6.
[2] See section 3.2.
[3] See section 3.3.3.

Chapter 5

Virtual Memory

5.1 Overview

The address space is partitioned into two main areas, *user space* and *physical memory space*. Figure 5.1 shows the partitioning of the simulator's virtual address space.

The virtual memory space is divided into fixed size chunks known as *pages*. The page is the fundamental unit of virtual memory. Each page is a region of bytes of memory, of size `PAGE_SIZE`.

Each byte of memory is said to exist at some *address*[1]. When an address points to the beginning of a page, we'll say that it is *page aligned*. The high order bits of an address will be known as the *page frame number*, that is they'll identify a particular page of memory. We'll refer to setting the value of memory at a particular address as *storing* or *writing* to that address, and getting a value from memory as *loading* or *reading*.

When we talk about *address spaces* we'll be referring to some range of addresses (which usually start and end on page boundaries). An *address space* should be considered distinct from actual memory in that all addresses in the space don't necessarily need to map directly to pages of real memory. An *unmapped* region of an address space consists of some number of *unmapped pages*. Typically when unmapped pages are accessed, they generate an exception (sometimes referred to as a *fault* or *page fault*). The real memory underlying a page (when it is mapped) is known as the *page frame*[2].

5.1.1 Constants and Macros

The simulator has some predefined constants and macros which pertain to virtual memory:

`PAGE_SIZE` Number of bytes in a page.

`PAGE_MASK` Mask which extracts the page frame bits in an address.

`PAGE_ALIGN()` Align the given address to the next page boundary.

`PAGE_SHIFT` Number of bits of the address which represent the offset into the address' page.

[1]The distinction between an *address* and a *pointer* is that typically we'll consider pointers to be addresses which point to an object of a particular type.

[2]Note the distinction between page frame and page frame *number*.

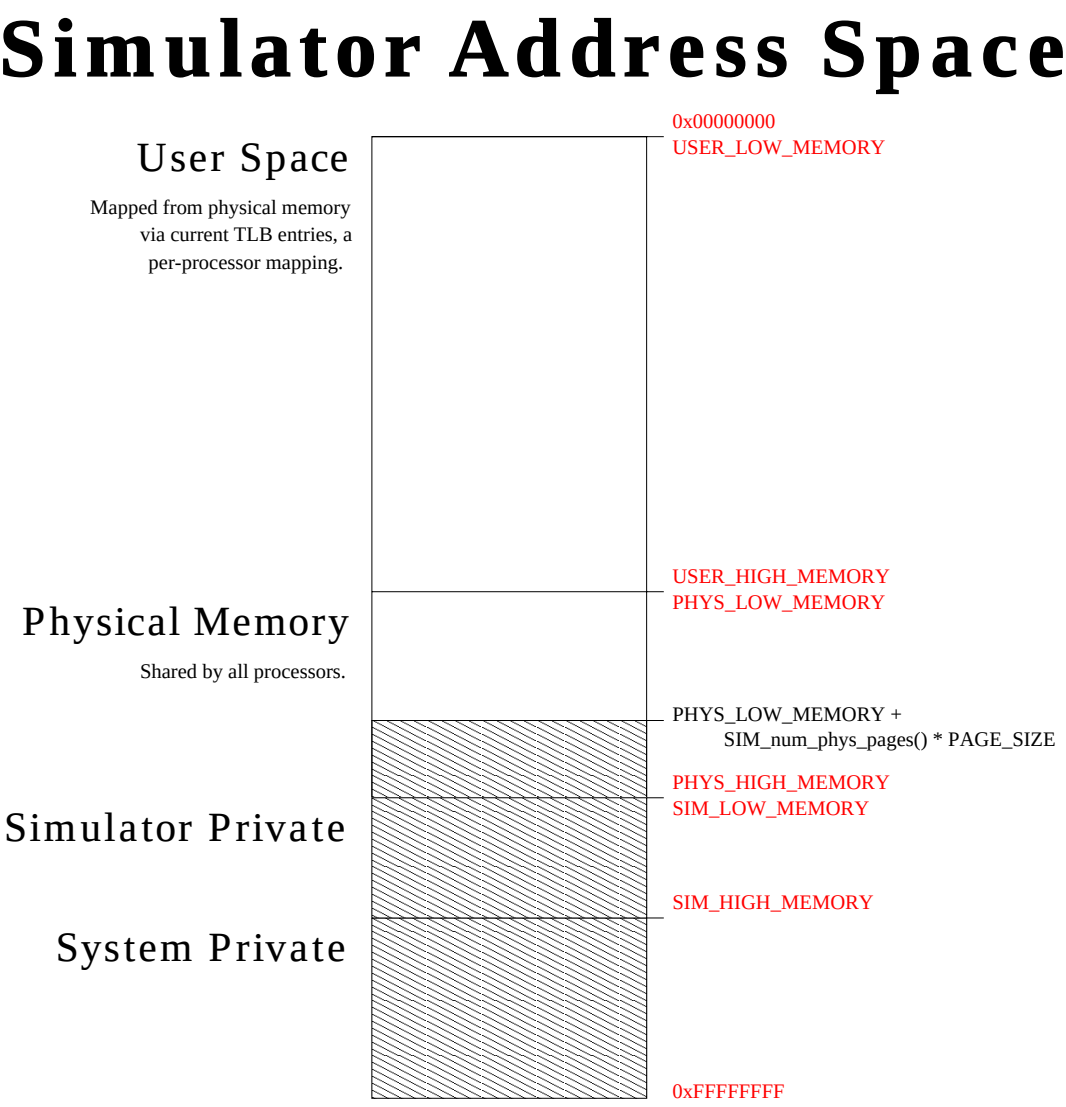

Figure 5.1: Simulator virtual address space

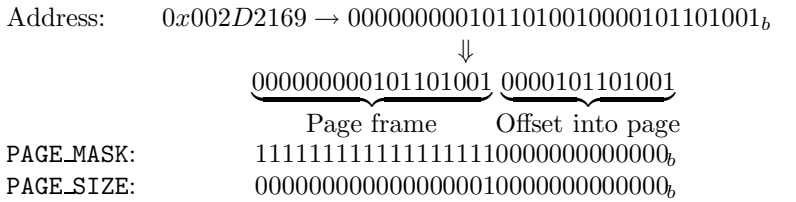

Figure 5.2: Breakdown of an address into page frame number, offset, etc.

Figure 5.2 shows an example of how an address is broken down into parts corresponding to some of the above definitions and constants. The figure shows the address and its corresponding binary representation. *Note that this example is for the particular case of a machine with* `PAGE_SIZE` *of 8192 and 32 bit addresses.*

5.1.2 Examples

1. Find the page frame number `pfn` for a given address `addr`:

   ```
   pfn = addr >> PAGE_SHIFT;
   ```

2. Given an address `addr`, find a pointer to it's page `pagep` and the offset of the address into that page `offset`:

   ```
   pagep = addr & PAGE_MASK;
   offset = addr & ~PAGE_MASK;
   ```

3. Two equivalent ways to determine the number of bytes in 8 pages:

   ```
   nbytes = 8 * PAGE_SIZE;
   nbytes = 8 << PAGE_SHIFT;
   ```

5.2 Physical Memory

Physical memory is the simulator's notion of the real memory installed in the simulated machine.

The range of addresses between `PHYS_LOW_MEMORY` and `PHYS_HIGH_MEMORY` is known as the *physical memory space* of the simulator. *The actual physical memory of the simulated machine doesn't actually take up all of this space.* Physical memory starts at the address `PHYS_LOW_MEMORY` and takes up as many pages as are actually installed in the machine.

The function `SIM_num_phys_pages()` returns the number of pages of physical memory which actually exist in the machine. These pages of physical memory start at the address `PHYS_LOW_MEMORY` and go up to the address `PHYS_LOW_MEMORY` + `PAGE_SIZE` * `SIM_num_phys_pages()`.

Dealing directly with physical memory is the kernel's responsibility. The kernel will generally allocate all of its data structures directly in physical memory, as well as control the allocation of pages of memory for individual processes.

Physical memory is not directly accessible when the machine is running in user mode[3]. The next section describes in general how physical memory is made available to user mode programs.

[3]See section 3.1.

5.3 User Space

User space is the area of memory which can be accessed by programs running in *user mode*. The kernel has explicit control over what memory user mode programs can access.

Pages of memory in user space are actually mapped to physical memory via a simulated piece of hardware known as the *TLB*. By controlling the TLB, the kernel can manipulate the virtual address space of user mode programs.

Essentially, when a user mode program accesses memory at a particular address, the memory access first proceeds through the TLB which typically translates the virtual address in user space directly to a physical address in physical memory.

User space is the range of addresses between `USER_LOW_MEMORY` and `USER_HIGH_MEMORY`. In contrast to the physical memory space, any and all of the pages in user space may refer to valid locations in physical memory (granted that this will imply a many-to-one mapping between user space and physical memory).

5.4 The Translation Look-aside Buffer (TLB)

The translation look-aside buffer (or *TLB*) keeps a cache of virtual-to-physical page translations for recently accessed pages of memory. The TLB is the only memory management hardware in the simulator. The simulator TLB is a simplified version of the TLB found in the *MIPS* architecture.

In order to provide a full virtual address space abstraction to user processes, the kernel must keep its own notion of the layout of user space in a set of *page tables*. The format of these page tables is up to the particular kernel implementation. Using the TLB and a set of page tables, the kernel can implement arbitrary virtual memory mappings of user space.

The TLB contains a predefined, fixed number of *TLB entries*, each representing the translation of one virtual page. Each TLB entry has a virtual frame number, physical frame number, and a set of flags. The number of entries in the TLB is defined by `SIM_TLB_NENTRIES`.

5.4.1 TLB Entries

Each TLB entry represents the translation of one page of addresses from the virtual address space (*user space*) to the physical address space (*physical memory*). All addresses which lie on that page in the virtual address space will be converted by the TLB to the same offset within the associated physical page. The TLB will swap the high-order bits of the address (specified by the `PAGE_MASK`) containing the virtual page number with the associated physical page number.

TLB entries contain the following three elements.

VFN The *virtual frame number*, the page number in user space of the page to be mapped. This could be considered the address to convert *from*.

PFN The *physical frame number*, the page number in physical memory of the page to be mapped. This could be considered the address to convert *to*. Note that in some cases this field is ignored, as described below.

Flags The flags affect how that address translation takes place (or if it takes place at all). The flags can be some combination of the following bit values:

`SIM_TLB_UNUSED` This TLB entry is empty. It is not used for address translation and the VFN and PFN are ignored.

`SIM_TLB_VALID` This TLB entry is valid, therefore addresses on the given virtual page will be translated to addresses on the given physical page. If the valid bit is not set, the PFN is ignored and all accesses to the given virtual page cause *tlbinvalid* exceptions.

`SIM_TLB_DIRTY` This TLB entry represents a *dirty page*, that is a page of memory which can be modified. The TLB entry should also have `SIM_TLB_VALID` set. If the dirty bit is not set, then all write accesses to that virtual page cause *tlbmodified* exceptions.

5.4.2 TLB-related exceptions

The TLB generates three exceptions, which must be handled by the kernel implementation:

- *tlbrefill* is called when a virtual page is accessed for which no TLB entry exists. The kernel implementation must provide an appropriate TLB entry for this memory reference to proceed. Typically, the kernel implementation will look through the page tables of the current process to determine what page (if any) should be mapped at that address, then call `SIM_tlb_wr()` to add an appropriate entry to the TLB or raise an exception if that section of the address space is meant to be unmapped.

- *tlbinvalid* is called when a virtual page is accessed for which an invalid TLB entry exists. The handling of this exception depends on the kernel implementation, since the kernel will have placed the invalid entry in the TLB for a reason, typically to represent a page which should not be accessed. An invalid TLB entry is an entry whose flags don't have the `SIM_TLB_VALID` bit set.

- *tlbmodified* is called when an attempt is made to modify a page whose TLB entry isn't marked as `SIM_TLB_DIRTY` (meaning that the page can be modified).

Some of the TLB exceptions are passed a `cause` argument, which specifies how the page was accessed in order to generate the exception. This argument can be one of the following values: `SIM_ACCESS_READ`, `SIM_ACCESS_WRITE`, and `SIM_ACCESS_EXEC`, for reading, writing, and executing memory, respectively.

5.4.3 TLB-related functions

The following functions are used to interface with the TLB.

`SIM_tlb_p(`*vfn*`)` Probes the TLB for an entry matching the given virtual page number. Returns the index of the entry, or -1 if no matching entry was found.

`SIM_tlb_wr(`*vfn, pfn, flags*`)` Writes to a random entry in the TLB, given a virtual page number, a physical page number, and a set of flags.

`SIM_tlb_wi(`*index, vfn, pfn, flags*`)` Writes to a particular entry in the TLB, given an entry index, a virtual page number, a physical page number, and a set of flags.

`SIM_tlb_ri(`*index, *vfn, *pfn, *flags*`)` Reads a particular entry from the TLB, given an entry index, and pointers to a virtual page number, physical page number, and flags.

`SIM_tlb_flush()` Clears out all entries in the TLB, setting their flags to `SIM_TLB_UNUSED`.

5.4.4 Examples

1. Map the virtual page containing the address *vaddr* to the physical page containing the address *paddr*, allowing the page to be modified:

```
SIM_tlb_wr(vaddr >> PAGE_SHIFT, paddr >> PAGE_SHIFT,
        SIM_TLB_VALID | SIM_TLB_DIRTY);
```

2. Flush the TLB, essentially unmapping all pages in user space efficiently:

```
SIM_tlb_flush();
```

3. Flush mappings in the virtual range [*start*, *end*) (useful if the kernel is changing attributes or removing an existing mapping):

```
sfn = start >> PAGE_SHIFT;
efn = end >> PAGE_SHIFT;
for (index = 0; index < SIM_TLB_NENTRIES; index++) {
        unsigned long vfn, pfn;
        int flags;
        SIM_tlb_ri(index, &vfn, &pfn, &flags);
        if (vfn >= sfn && vfn < efn)
                SIM_tlb_wi(index, 0, 0, SIM_TLB_UNUSED);
}
```

Chapter 6

Devices

6.1 Introduction

6.1.1 Programmed I/O vs. Direct Memory Access

All devices use either a *programmed I/O* (PIO) model or a *direct memory access* (DMA) model. PIO devices transmit and receive data entirely through special device registers. In the case of DMA, data is transmitted and received directly through physical memory.

6.2 Device registers

6.2.1 Device control register

All devices have a control register, which is used by the kernel to issue commands to the device. `SIM_dev_ctl(`*dev, value*`)` sets the value of this register. Setting the value of this register may have side-effects which depend on the type of the device. It is not possible to read a device's control register.

6.2.2 Device status register

All devices have a status register, which is used by the kernel to detect the status of the device, such as whether the device is ready to receive data, etc. `SIM_dev_sts(`*dev*`)` gets the value of this register. Reading a device's status register generally has no side-effects. It is not possible to write to a device's status register. The contents of the device's status register may change at any time, depending on the operation of the device.

6.2.3 PIO device registers

The kernel interacts with a PIO device using two registers: a *read* register and a *write* register. The function `SIM_dev_rreg(`*dev*`)` gets the value of a PIO device's read register and the function `SIM_dev_wreg(`*dev, value*`)` sets the value of a PIO device's write register.

Reading and writing these registers generally have no side-effects, although manipulating the registers while the device is not ready usually has undefined results.

Typically, when a device is ready to receive data (signaled by an interrupt or change in device status bits), the kernel should first set the value of the write register, then set the appropriate

bits in the control register to allow the write operation to proceed. A similar strategy is used for reading from a device, except that the kernel waits for the device to have data ready to read (signaled by an interrupt or change in device status bits), then reads from the device's read register and signals that the device can provide more data by setting the appropriate bits in the control register.

6.2.4 DMA device registers

DMA devices have two registers: a *memory address* register and a *device address* register. The functions `SIM_dev_maddr(`*dev, value*`)` and `SIM_dev_daddr(`*dev, value*`)` set the values of these registers, respectively.

Reading and writing these registers generally have no side-effects, although manipulating the registers while the device is not ready usually has undefined results.

The memory address register is set to a value in physical memory, which the device will either read from or write to (depending on the operation the kernel specifies in the device's control register).

The device address register is set to a device-specific address, which the device will either read from or write to (again, depending on the operation being performed).

The procedure for using these registers is similar to that of PIO, except values go directly to and from memory.

6.2.5 Examples

For specific examples, look at the disk and terminal sections, below.

6.3 Configuration files

The simulator supports a flexible scheme for configuring devices. The configuration of devices is specified in a configuration file, specified on the command-line. If none is specified, the simulator will use a default configuration file.

The device configuration file is specified as a command-line option to **simrun** or defaults to the file **simconfig** in the config search path, as described in section 2.1.

The configuration file is a series of lines of the form
`<OBJECT_FILE> <IPL> [<INIT_STRING> ... ]`, where `<OBJECT_FILE>` is the shared object for the device, `<IPL>` is the interrupt priority level at which the device interrupts, and `<INIT_STRING>` is a device-dependent initialization string. Anything to the right of a hash mark ('#') is considered a comment and ignored by the Simulator.

The simulator includes shared objects for terminal devices and disk devices.

6.4 Terminal device

The terminal is a PIO device which is capable of reading and writing characters sequentially from the user and to the display (respectively). The terminal currently emulates a DEC VT102 (actually an XTerm, of course), so VT100 code sequences should produce highlighting, clearing the screen, moving the cursor, etc. Simple device drivers can treat the device as a dumb terminal and leave fancy output to higher level (e.g. tty) drivers.

6.4.1 Control

The terminal device has several meaningful bits in the control register which the kernel may set in order to control the terminal. The bit masks are as follows:

`SIM_TERM_RGO` Begin a read operation. The terminal will enter the `SIM_TERM_RREADY` state when the user has pressed a key (and therefore the terminal has a character for the kernel to read).

`SIM_TERM_RENABLE` If set, the terminal will generate an interrupt when it enters the `SIM_TERM_RREADY` state.

`SIM_TERM_WGO` Begin a write operation. The terminal will write the character in its write register to the display. When the terminal is ready to write another character, it will enter the `SIM_TERM_WREADY` state.

`SIM_TERM_WENABLE` If set, the terminal will generate an interrupt when it enters the `SIM_TERM_WREADY` state.

Note that not setting the `SIM_TERM_RENABLE` *bit is meaningful even if you are beginning a write operation, since there might be a read in progress and setting the bit to zero disables read interrupts! The same goes for not setting the write enable when starting a read operation.*

6.4.2 Status

The terminal device has several meaningful bits in its status register which determine the state of the terminal device. The bit masks for accessing these bits are as follows:

`SIM_TERM_RREADY` Is the terminal ready to give you a character? If this bit is set, then the terminal's read register contains the next character that the user has typed.

`SIM_TERM_WREADY` Is the terminal ready to accept a character? If this bit is set, then the terminal is ready to write a character to the display. The terminal's write register is clear and waiting to be set, then begin a write operation.

Note that the terminal may be both RREADY *and* WREADY *at the same time, since they are both bits in the status register. A properly written device driver needs to handle these cases independently.*

6.4.3 Configuration file parameters

The `<INIT_STRING>` for the terminal is of the form:
`<NUM_ROWS> <NUM_COLS>`, where the parameters are the number of rows and columns (respectively) which control the size of the terminal.

6.4.4 Device-specific parameters

The **sim_term_params** structure allows the kernel to discover how many rows and columns of characters the terminal is capable of displaying, as the members *tp_rows* and *tp_cols* respectively.

6.4.5 Examples

1. Configure an 80 by 24 character terminal which generates interrupts at IPL 5, in your **simconfig** file:

   ```
   term.so 5 24 80
   ```

2. Write a character to the terminal, keeping both read and write interrupts enabled. `ch` is the character to write and `dev` is the device.

   ```
   SIM_dev_wreg(dev, ch);
   SIM_dev_ctl(dev, SIM_TERM_WGO | SIM_TERM_WENABLE | SIM_TERM_RENABLE);
   ```

6.5 Disk device

The disk is a DMA device which is capable of reading or writing blocks in a random-access fashion by first seeking to a location then performing the read or write operation.

6.5.1 Control

The disk device has several meaningful bits in the control register which the kernel may set in order to control the disk. The bit masks are as follows:

`SIM_DISK_GO` Begin an operation. Must be combined with one of the `SIM_DISK_OP_*` operations.

`SIM_DISK_ENABLE` If set, the disk will generate an interrupt when it enters the `SIM_DISK_READY` state.

`SIM_DISK_OP_WRITE` Begin a write operation. The disk will take the values stored at the physical memory location in its memory address register and write those values to the current disk location. When the write operation is finished, the disk will enter the `SIM_DISK_READY` state.

`SIM_DISK_OP_READ` Begin a read operation. The disk will take the values stored at the current disk location and write those values to the physical memory location in its memory address register. When the read operation is finished, the disk will enter the `SIM_DISK_READY` state.

`SIM_DISK_OP_SEEK` Begin a seek operation. The disk head will move such that the current disk location is that which is specified in the device address register. When the seek operation is finished, the disk will enter the `SIM_DISK_READY` state.

Note that the `SIM_DISK_GO` *bit is required to be set to begin any operation. Also note that the disk must* SEEK *to a disk location each time before a* READ *or* WRITE *operation.*

6.5.2 Status

The disk device has one meaningful bit in its status register, specified by the bit mask `SIM_DISK_READY`. When this bit is set, the disk has completed its requested operation and is ready for the next operation.

6.5.3 Configuration file parameters

The <INIT_STRING> for the disk is of the form:
<NUM_BLOCKS> <BLOCK_SIZE> <SEEK_DELAY> <DISK_FILE> [<DIFF_FILE>], where

<NUM_BLOCKS> Number of blocks on the disk.

<BLOCK_SIZE> Size of each disk block in bytes.

<SEEK_DELAY> Delay of disk seeks, in milliseconds.

<DISK_FILE> Name of file used to store disk contents.

<DIFF_FILE> Optional parameter, specifying the name of a file used to store modifications to the disk contents. If this option is specified, the file named by <DISK_FILE> is not modified by the simulator (and will be opened read-only). This option can be used to save disk space if many individuals wish to share a large disk file whose contents will remain mostly unmodified.

6.5.4 Device-specific parameters

The **sim_disk_params** structure allows the kernel to discover the configuration of a particular disk device. The members of the structure are as follows:

dp_num_blocks The number of blocks in the disk.

dp_block_size The size of each block in the disk.

dp_seek_delay The amount of time, in milliseconds, that it takes to move the head to a particular location on disk (the disk seek latency).

6.5.5 Examples

1. Configure a disk (whose file is named "silly") which generates interrupts at IPL 7, with 128 8k blocks and a seek latency of 20 milliseconds, in your **simconfig** file:

   ```
   disk.so 5 128 8192 20 silly
   ```

2. Seek to block `blocknum`, keeping interrupts enabled.

   ```
   SIM_dev_daddr(dev, blocknum);
   SIM_dev_ctl(dev, SIM_DISK_GO | SIM_DISK_OP_SEEK | SIM_DISK_ENABLE);
   /* ... wait for interrupt ... */
   ```

3. Read the current block into `buffer`, keeping interrupts enabled. (Note that the data in `buffer` won't be valid until the interrupt occurs.)

   ```
   SIM_dev_maddr(dev, buffer);
   SIM_dev_ctl(dev, SIM_DISK_GO | SIM_DISK_OP_READ | SIM_DISK_ENABLE);
   /* ... wait for interrupt ... */
   ```

6.6 Detecting devices and configurations

The simulator has an interface for the kernel to detect configured devices and get their configuration parameters.

The **sim_devconfig_t** structure is used to store the configuration parameters of a particular device. This structure contains the following members:

- *dc_type* - Device type, which may be one of `SIM_TYPE_DISK`, `SIM_TYPE_TERM`, or `SIM_TYPE_FB`, or a user-defined type.
- *dc_dev* - Device handle, which should be passed to all `SIM_dev_*()` functions.
- *dc_ipl* - IPL[1] at which this device generates interrupts, if any.
- *dc_params* - Device-specific parameters.

6.6.1 How many devices are configured?

`SIM_num_devs()` will return the number of configured devices in the machine.

6.6.2 Finding a device's configuration

`SIM_dev_config(`*dev, *cfg*`)` returns a device's configuration parameters, given a device handle of type **sim_device_t**.

`SIM_dev_config_n(`*n, *cfg*`)` returns the configuration parameters of the *n*th device, where *n* can be in the range 0 . . . (`SIM_num_devs()` - 1).

[1] See section 3.3.3.

Appendix A

Simulator C interface

A.1 sim.h

```
/* The Brown Simulator.
   Copyright (C) 1998 Jason Lango, Keith Adams, Michael Castelle, David Powell

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2, or (at
   your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.  */

/*
**  sim.h - Simulator public interface.
**  Jason Lango <jalcs.brown.edu>
**  Keith Adams <kmacs.brown.edu>
*/

#ifndef __SIM_H__
#define __SIM_H__

#include <ucontext.h>
#include <limits.h>
#include <stdarg.h>

#ifdef __cplusplus
extern "C" {
#endif

#if defined(sun)
#define SOLARIS 1
#include "sim_arch_solaris.h"
#elif defined(sgi)
#include "sim_arch_irix.h"
```

```
#endif

#define SIM_MAJOR_VERSION 1
#define SIM_MINOR_VERSION 1
#define SIM_PATCH_VERSION 0
#define SIM_VERSION_CODE ((((SIM_MAJOR_VERSION << 8) + \
                            SIM_MINOR_VERSION) << 8) + \
                            SIM_PATCH_VERSION)

#define SIM_MAX_CPUS   256
#define SIM_MAX_DEVS   32

#define PAGE_SIZE      (1 << PAGE_SHIFT)
#define PAGE_MASK      (~(PAGE_SIZE-1))
#define PAGE_ALIGN(addr) (((addr)+PAGE_SIZE-1)&PAGE_MASK)
#define PAGE_SAME(a1,a2) (((a1)>>PAGE_SHIFT) == ((a2)>>PAGE_SHIFT))

#define WORD_SIZE      (sizeof(long))
#define DWORD_SIZE     (sizeof(long long))
#define WORD_MASK      (~(WORD_SIZE-1))
#define DWORD_MASK     (~(DWORD_SIZE-1))
#define WORD_ALIGN(addr) (((addr)+WORD_SIZE-1)&WORD_MASK)
#define DWORD_ALIGN(addr) (((addr)+DWORD_SIZE-1)&DWORD_MASK)

#define USER_LOW_MEMORY 0
#define PHYS_LOW_MEMORY USER_HIGH_MEMORY

#define SIM_ACCESS_READ 0
#define SIM_ACCESS_WRITE 1
#define SIM_ACCESS_EXEC 2

#define SIM_NUM_IPL    16
#define SIM_LOW_IPL    0
#define SIM_HIGH_IPL   (SIM_NUM_IPL - 1)
#define SIM_CLK_IPL    SIM_HIGH_IPL

#define SIM_CPU_ALL    -10
#define SIM_CPU_NOTME -11

#define SIM_PSR_MODE   0x0001 /* mask for psr mode */
#define SIM_USER_MODE 0
#define SIM_SUPERVISOR_MODE 1
#define SIM_PSR_USER_MODE(psr) (((psr) & SIM_PSR_MODE) == SIM_USER_MODE)
#define SIM_PSR_SUPERVISOR_MODE(psr) (!SIM_PSR_USER_MODE(psr))

/*
 * device types
 */
#define SIM_TYPE_TERM SIM_TYPE_INTERNAL(1)
#define SIM_TYPE_DISK  SIM_TYPE_INTERNAL(2)
#define SIM_TYPE_FB    SIM_TYPE_INTERNAL(3)
#define SIM_TYPE_RESERVED 0x80000000
#define SIM_TYPE_INTERNAL(x) (SIM_TYPE_RESERVED | (x))

#define SIM_DEV_MAXPARAMS (10)
#define SIM_DEV_INVALID ((sim_device_t)NULL)

/*
 * disk constants
 */
/* control reg constants */
```

```
#define SIM_DISK_GO 0x100
#define SIM_DISK_ENABLE 0x10
#define SIM_DISK_OP_WRITE 0x2
#define SIM_DISK_OP_READ 0x4
#define SIM_DISK_OP_SEEK 0x8
/* status reg constants */
#define SIM_DISK_READY 0x1

/*
 * terminal constants                                                    110
 */
/* control reg constants */
#define SIM_TERM_RGO 0x100
#define SIM_TERM_WGO 0x200
#define SIM_TERM_RENABLE 0x10
#define SIM_TERM_WENABLE 0x20
/* status reg constants */
#define SIM_TERM_RREADY 0x1
#define SIM_TERM_WREADY 0x2
                                                                         120
typedef unsigned long sim_addr_t;

typedef struct sim_term_params {
        unsigned long       tp_rows;
        unsigned long       tp_cols;
} sim_term_params_t;

typedef struct sim_disk_params {
        unsigned long       dp_num_blocks;
        unsigned long       dp_block_size;                               130
        unsigned long       dp_seek_delay;
} sim_disk_params_t;

typedef struct sim_fb_params {
        unsigned long       fbp_width;    /* Width of the framebuffer */
        unsigned long       fbp_height;   /* Height of the framebuffer */
        unsigned long       fbp_padding;  /* Bit padding per scanline */
        unsigned long       fbp_bits;     /* Bits per pixel */
        unsigned long       fbp_rmask;    /* The red mask */
        unsigned long       fbp_gmask;    /* The blue mask */              140
        unsigned long       fbp_bmask;    /* The green mask */
} sim_fb_params_t;

typedef struct sim_context {
        ucontext_t          sc_ucontext;
        int                 sc_ipl;
        int                 sc_psr;
        sim_addr_t          sc_kstack;
        size_t              sc_kstack_size;
} sim_context_t;                                                         150

typedef void *sim_device_t;
typedef void (*sim_intr_func_t)(int ipl, sim_device_t dev, sim_context_t *ctx);
typedef int (*sim_trap_func_t)(unsigned long arg1, void *arg2,
        sim_context_t *ctx);
typedef void (*sim_thread_func_t)(unsigned long arg1, void *arg2);

typedef struct sim_devconfig {
        int                 dc_type;
        sim_device_t        dc_dev;                                      160
        int                 dc_ipl;
```

```
        unsigned long       dc_params[SIM_DEV_MAXPARAMS];
} sim_devconfig_t;

typedef struct sim_tvb {
        /* Interrupts.
        **
        ** intr - device interrupt.
        ** ipi - interprocessor interrupt.
        ** clock - clock interrupt.                                          170
        **
        **    dev is the device causing the interrupt, if any.
        */
        void (*st_intr)(int ipl, sim_device_t dev, sim_context_t *ctx);
        void (*st_ipi)(int ipl, sim_context_t *ctx);
        void (*st_clock)(sim_context_t *ctx);

        /* Traps and exceptions.
        **
        ** illinst - illegal instruction exception.                          180
        ** trap - trap instruction.
        ** break - trace/breakpoint instruction exception.
        ** fpe - floating point error exception.
        */
        void (*st_illinst)(sim_context_t *ctx);
        int (*st_trap)(unsigned long arg1, void *arg2, sim_context_t *ctx);
        void (*st_break)(sim_context_t *ctx);
        void (*st_fpe)(sim_context_t *ctx);

        /* Memory access errors.                                             190
        **
        ** buserr - when an address has invalid alignment.
        ** addrerr - when a user-mode program tries to access non-user memory.
        **
        **    cause is one of SIM_ACCESS_{READ,WRITE,EXEC}.
        */
        void (*st_buserr)(unsigned long vaddr, int cause, sim_context_t *ctx);
        void (*st_addrerr)(unsigned long vaddr, int cause, sim_context_t *ctx);

        /* TLB-related exceptions.                                           200
        **
        ** tlbrefill - no TLB entry matches a given address.
        ** tlbinvalid - address referenced which is marked INVALID in TLB.
        ** tlbmodified - attempted store to address not marked DIRTY in TLB.
        **
        **    cause is one of SIM_ACCESS_{READ,WRITE,EXEC}.
        */
        void (*st_tlbrefill)(unsigned long vaddr, int cause,
                sim_context_t *ctx);
        void (*st_tlbinvalid)(int index, unsigned long vaddr, int cause,  210
                sim_context_t *ctx);
        void (*st_tlbmodified)(unsigned long vaddr, sim_context_t *ctx);
} sim_tvb_t;

/*****************************************************************/
/* Machine Interface                                             */
/*****************************************************************/

/* num_phys_pages - Get the number of pages of physical memory in the
 * machine.                                                          220
 */
unsigned long SIM_num_phys_pages(void);
```

```
/* abort - Same as abort(3c), but simulator-safe.  Dumps core for the
 * current simulated processor.
 */
void SIM_abort(void);

/* halt - Halts the machine, effectively exiting the simulator
 * cleanly.                                                          230
 */
void SIM_halt(void);

/* printf - Simulator-safe printf() function. */
int SIM_printf(const char *fmt, ...);
int SIM_vprintf(const char *fmt, va_list ap);
#ifndef SIM_NO_OVERRIDE_PRINTF
#define printf SIM_printf
#define vprintf SIM_vprintf
#endif /* SIM_NO_OVERRIDE_PRINTF */                                  240

/*************************************************************/
/* Processor Interface                                       */
/*************************************************************/

/* num_cpus - Get the number of configured processors. */
int SIM_num_cpus(void);

/* cpu_id - Get the processor number on which we're executing. */
int SIM_cpu_id(void);                                             250

/* make_context - Setup/modify the given sim_context. */
void SIM_make_context(sim_context_t *ctx,
        sim_thread_func_t func,
        unsigned long arg1, void *arg2,
        sim_addr_t stack, size_t stack_size,
        sim_addr_t kstack, size_t kstack_size,
        int psr, int ipl, const sim_context_t *return_ctx);

/* set_context - Switch to sim_context, discarding current processor state. */  260
void SIM_set_context(const sim_context_t *new_ctx);

/* swap_context - Switch between sim_contexts. */
void SIM_swap_context(sim_context_t *old_ctx, const sim_context_t *new_ctx);

/* set_tvb - Set the trap vector block. */
void SIM_set_tvb(sim_tvb_t *tvb);

/* trap - Trap instruction, transfers control to the kernel trap
 * handler, on the interrupt stack.                                  270
 */
int SIM_trap(unsigned long arg1, void *arg2);

/* set_ipl - Set the interrupt priority level.  Interrupts at this
 * level and lower will be masked until the ipl is lowered.  Returns
 * the old value of the ipl.
 */
int SIM_set_ipl(int ipl);

/* get_ipl - Get the interrupt priority level. */                     280
int SIM_get_ipl(void);

/* get_psr - Get the value of the processor status register. */
```

```
int SIM_get_psr(void);

/* get_pir - Get the value of the pending interrupt register. */
int SIM_get_pir(void);

/* set_clk - Set the processor clock register.  This will set the
 * clock on the current processor to generate interrupts every msecs
 * milliseconds.  Setting the clock register to zero will disable
 * clock interrupts.  The clock generates interrupts at SIM_CLK_IPL.
 */
void SIM_set_clk(int msecs);

/* idle - Cause the processor to idle, saving "power".  :-)
 * In reality, this causes the simulator to use less compute time
 * during idle loops.
 */
void SIM_idle(void);

/* ipi - Send an interprocessor interrupt to a set of processors.
 *       cpu is one of:
 *           - a processor number, to send to a particular processor
 *           - SIM_CPU_ALL, to send to all processors
 *           - SIM_CPU_NOTME, to send to all but this processor
 *       ipl is any valid ipl.
 */
void SIM_ipi(int cpu, int ipl);

/*************************************************************/
/* Device Interface                                          */
/*************************************************************/

/* num_devs - Get the number of configured devices. */
int SIM_num_devs(void);

/* dev_config_n - Get the configuration of the Nth device. */
void SIM_dev_config_n(int n, sim_devconfig_t *cfg);

/* dev_config - Get the configuration of this device. */
void SIM_dev_config(sim_device_t dev, sim_devconfig_t *cfg);

/* dev_sts - Get the value of this device's status register. */
unsigned SIM_dev_sts(sim_device_t dev);

/* dev_ctl - Set the value of this device's control register. */
void SIM_dev_ctl(sim_device_t dev, unsigned val);

/* dev_rreg - Get the value of this device's read register.
 * Only valid for certain programmed I/O (PIO) devices.
 */
unsigned SIM_dev_rreg(sim_device_t dev);

/* dev_wreg - Set the value of this device's write register.
 * Only valid for certain programmed I/O (PIO) devices.
 */
void SIM_dev_wreg(sim_device_t dev, unsigned val);

/* dev_maddr - Set the value of this device's memory address register.
 * Only valid for certain direct memory access (DMA) devices.
 */
void SIM_dev_maddr(sim_device_t dev, unsigned long addr);
```

```
/* dev_daddr - Set the value of this device's device address register.
 * Only valid for certain direct memory access (DMA) devices.
 */
void SIM_dev_daddr(sim_device_t dev, unsigned long addr);

/*************************************************************/
/* Translation Look-aside Buffer (TLB) Interface             */
/*************************************************************/

/* PFN - page frame number, the upper bits of the physical page
 * address
 *
 * VFN - virtual frame number, the upper bits of the virtual page
 * address
 *
 * Flags:
 * V - entry is valid
 * D - entry is dirty (page is modifiable)
 */

#define SIM_TLB_NENTRIES    64

/* TLB entry flags */
#define SIM_TLB_VALID       (0x0001)
#define SIM_TLB_DIRTY       (0x0002)
#define SIM_TLB_UNUSED      (0x0004)

/* tlb_p - Probe TLB for an entry matching the given VFN.
   Returns the index, or -1 if no entry found. */
int SIM_tlb_p(unsigned long vfn);

/* tlb_wr - Write into a random entry in the TLB. */
void SIM_tlb_wr(unsigned long vfn, unsigned long pfn, int flags);

/* tlb_wi - Write into a particular indexed entry in the TLB. */
void SIM_tlb_wi(int index, unsigned long vfn, unsigned long pfn, int flags);

/* tlb_ri - Read from a particular indexed entry in the TLB. */
void SIM_tlb_ri(int index, unsigned long *vfn, unsigned long *pfn, int *flags);

/* tlb_flush - Clear out the entire tlb. */
void SIM_tlb_flush(void);

/*************************************************************/
/* Kernel Entry Point                                        */
/*************************************************************/

void kern_boot(int cpu_num);

#ifdef __cplusplus
}
#endif

#endif /* __SIM_H__ */
```

A.2 sim_arch_solaris.h

```
/* The Brown Simulator.
   Copyright (C) 1998 Jason Lango, Keith Adams, Michael Castelle, David Powell

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2, or (at
   your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.  */

/*
**  sim_arch_solaris.h - Architecture-dependent code for SPARC/Solaris.
**  Jason Lango <jalcs.brown.edu>
**  Keith Adams <kmacs.brown.edu>
*/

#ifndef _SIM_ARCH_SOLARIS_H
#define _SIM_ARCH_SOLARIS_H

#ifndef _ASM
#include <sys/frame.h>
#endif /* !_ASM */

/* The following constants are determined experimentally, based on
 * where ld.so locates the shared libraries within the process address
 * space.
 */
#define USER_HIGH_MEMORY (0xC0000000UL)
#define PHYS_HIGH_MEMORY (0xD0000000UL)
#define SIM_HIGH_MEMORY (0xE0000000UL)

/* The log base 2 of the page size, as returned by getpagesize().
*/
#define PAGE_SHIFT      (13UL)

/* The maximum number of mappings in a shared library.
*/
#define SIM_MAX_SHLIB_MAPPINGS 4

#define SIM_IS_LOADSTORE(x) ((x) & 0xC0000000UL)
#define SIM_IS_STORE_INSTR(x) (SIM_IS_LOADSTORE(x) && (x) & (1 << 21))
#define SIM_IS_LOAD_INSTR(x) (SIM_IS_LOADSTORE(x) && !SIM_IS_STORE_INSTR(x))

#define SIM_CONTEXT_PC(uc) \
        ((void*)(uc)->uc_mcontext.gregs[REG_PC])
#define SIM_CONTEXT_SP(uc) \
        ((void*)(uc)->uc_mcontext.gregs[REG_SP])
#define SIM_CONTEXT_EXEC_PAGE(uc) \
        (((unsigned long)(uc)->uc_mcontext.gregs[REG_PC]) >> PAGE_SHIFT)
#define SIM_CURRENT_INSTR(uc) \
        (*(unsigned long*)SIM_CONTEXT_PC(uc))
```

```
/* Architecture-dependent mode switch code.                                    60
*/

#define SIM_ARCH_SET_USER() \
        do { \
        } while(0)

#define SIM_ARCH_SET_SUPER() \
        do { \
        } while (0)
                                                                              70
/* Architecture-dependent post-makecontext context initialization.
*/
#define SIM_UCONTEXT_POSTINIT(uc) \
        {                                                   \
                struct frame *f;                            \
                /* Clear out the saved frame pointer. */    \
                f = (struct frame*) (uc)->uc_mcontext.gregs[REG_O6]; \
                f->fr_savfp = 0;                            \
                f->fr_savpc = 0;                            \
                /* We never want to return.  Ever. */       \                 80
                (uc)->uc_mcontext.gregs[REG_O7] = 0;    \
        }

/* Since it is architecture dependent whether stacks grow upwards,
 * downwards, sideways, etc. we'll use an architecture-dependent macro
 * for setting the stack_t members.
 */
#define SIM_SET_STACK(s, stack, stack_size) \
        {                                                   \
                (s)->ss_sp = (void*) (stack + stack_size - DWORD_SIZE); \     90
                (s)->ss_size = stack_size;                  \
                (s)->ss_flags = 0;                          \
        }

/* Get the arguments to a trap.
*/
#define SIM_TRAP_ARGS(uc,a1,a2) \
        {                                                   \
                struct frame *f;                            \
                f = (struct frame*) (uc)->uc_mcontext.gregs[REG_O6]; \        100
                (a1) = (unsigned long) f->fr_arg[0];        \
                (a2) = (void*) f->fr_arg[1];                \
        }

/* Advance the PC past a simulated trap instruction.
*/
#define SIM_TRAP_ADVANCE(uc) \
        {                                                   \
                (uc)->uc_mcontext.gregs[REG_PC] += 4; \
                (uc)->uc_mcontext.gregs[REG_nPC] += 4; \                      110
        }

/* Set the return value for a trap.
*/
#define SIM_TRAP_RETURN(uc,retval) \
{ \
        (uc)->uc_mcontext.gregs[REG_O0] = (retval); \
}

#ifndef _ASM                                                                  120
```

```
typedef unsigned atomic_t;
extern atomic_t SIM_atomic_swap(atomic_t*, atomic_t);

void sim_flush_windows(void);

#endif /* !_ASM */

#define SIM_TRAP_START (0xdfffe000) /* magic */
#define SIM_TRAP  SIM_TRAP_START                                        130
#define SIM_TRAP_END (SIM_TRAP_START + (1UL << PAGE_SHIFT))

#endif /* _SIM_ARCH_SOLARIS_H */
```

Index