SYSTEMS II — PROJECT 6
A virtual memory system
(revision D)

1 Overview

For this project, we will implement a *paged virtual memory system*. The virtual addressing mode of our ISA will no longer assume a *base* and *limit* for each process, but rather a page-based allocation of virtual space, physically backed by main memory and a backing store. Each process will see its own, complete virtual address space. As it allocates and uses virtual pages, the CPU and kernel will conspire to map these virtual pages with real page frames, supporting the illusion that each process has main memory to itself.

2 Updates to the ISA, compiler, assembler, and simulated system

In order to implement a virtual memory system, we need some further changes from the ISA first presented as part of Project 1 and then later modified for Project 4. This project includes modifications to the ISA, the system simulator, and the assembler, all of which are described below:

2.1 The ISA

2.1.1 Pages and page tables

The newly modified ISA will employ a virtual address mode that divides the $2^{16}\ bytes$ available in the virtual address space into $2^8 = 256\ pages \times 2^8 = 256\ \frac{bytes}{page}$. Therefore, each 16-bit virtual address can be divided in half, where the upper 8 bits is the *page number* and the lower 8 bits are the *page offset*.

A *page table* will therefore contain $2^8 = 256\ entries$, indexed by virtual page number. Each entry would comprise a one-word value, implying that each page table would occupy 2 page frames. where the **upper byte** would contain the *physical page frame number* to which the virtual page is mapped, and the **lower byte** would contain a number of status bits for the given virtual page. Specifically, these status bits are:

- `Bit 0`: **Valid:** `1` if the page table entry represents a validly mapped virtual page; `0` otherwise.
- `Bit 1`: **Resident:** `1` if this virtual page is cached in main memory; `0` otherwise.
- `Bit 2`: **Indirect:** `0` if the page frame number component of this PTE indicates the back frame that backs this virtual page; `1` if the page frame number component indicates a page frame that contains another page table that should be consulted to perform this translation.

- `Bits 3/4/5`: **User read/write/execute permission:** `1` if this virtual page may be read/written/executed (respectively) in user-mode; `0` otherwise.

2.1.2 Clocks and alarms

The CPU now how two new register:

- *Clock register*: A 32-bit register that stores the number of cycles/instructions that have passed since the system was activated. This value is updated on every cycle/instruction.
- *Alarm register*: A 32-bit register that stores the clock value at which to send the CPU (from within) with a `CLOCK_ALARM` interrupt.

The clock register is automatically maintained by the CPU. The alarm is set in software by the `SETALM` instruction (see Section 2.1.4). When these two registers are equal, the `CLOCK_ALARM` interrupt is raised.

2.1.3 Kernel/user space conventions

With a change of virtual addressing comes a change in how kernel- and user-level code share space. Previously, the kernel was loaded at physical address `0x0000`, and ran in physical address mode. Now, a different model must be used.

The kernel should be mapped into every virtual address space. Specifcally, the upper $\frac{1}{4}$ of each virtual address space should be dedicated to the kernel. That is, addresses `0x0000` to `0xbfff` (inclusive) are for use by the user-level application code, while `0xc000` to `0xffff` are addresses for the kernel.

2.1.4 Instructions

Some instructions from the previous ISA have been eliminated, while other have been added. These instructions are available only in supervisor mode. Specifically, these two instructions have been **depricated**—that is, they will now cause `INVALID_INSTRUCTION` interrupts:

- `0x14:  SETBS` *[source A]*
- `0x15:  SETLM` *[source A]*

The following instructions have been **added**:

- `0x19:  SETPT` *[source A]*

  Copy *source A* into the *page table register*. This register should point to the current page table that defines the virtual address space of the current process.

- `0x1a:  GETCLK` *[destination]*

  Copy the current value of the 32-bit *clock register* into double-word space *destination*.

- `0x1a:` `SETALM` *[source A]*

  Set the 32-bit *alarm register* to the double-word value given as *source A*. When the *clock register* equals the *alarm register*, the CPU will raise a `CLOCK_ALARM` interrupt.

Finally, the following instructions are themselves unchanged, but elements associated with them are **changed**:

- `0x16:` `SETIP` *[source A]*

  Copy *source A* into the *instruction preserve register*. This value is taken as a pointer to the *instruction preserve region*—a physical address to a main memory space into which the CPU will copy critical values upon an interrupt.

  Previously, this space was a single word in length, and the CPU copied the PC into this space before jumping into the interrupt handler specified by the trap table.

  Currently, this space must be two words. In the first, the CPU will place a copy of the PC before vectoring to the interrupt handler. Into the second, the CPU may place auxiliary data associated with the interrupt. For example, in virtual addressing mode, upon an `INVALID_ADDRESS` interrupt, the CPU will place into this auxiliary space a copy of the virtual address whose translation failed.

- `0x17:` `SETVA` *[source A]*

  If *source A* = *0*, then place the CPU in *physical addressing mode*; otherwise, place the CPU in *virtual addressing mode*.

  Previously, virtual addressing applied only to user mode, and not to supervisor mode.

  Currently, virtual addressing is orthogonal to user/supervisor mode selection.

2.2 Assembler

The assembler has not been modified by much. However, you will notice that, from Section 2.1.4, the new instructions have been added, and the depricated ones will cause assembly errors. Additionally, the `SHFTL` and `SHFTR` instructions, although defined previously by the ISA, had not been supported by the assembler, and now they are.

Critically, the assembler now also requires that you provide, at the command line, a starting address at which the executable image will be loaded. For normal programs, you should pass `0x0000`, since those images will be loaded at the beginning of the process's virtual address. For the kernel, however, its code will be loaded at `0xc000`, and so that value should be passed as the starting address to the assembler. You will note that the newly provided script, `compile-kernel.sh` (see Section 3.1), correctly passes this value to the assembler.

2.3 Compiler

The compiler provided performs complete code generation. In fact, the lastest version of the assembler corrects a number of critical code generation errors. While there may still be some errors, the code generated by the assembler does seem to work correctly a substantial fraction of the time. Help me improve it by carefully examining the code you generate with it.

The most essential addition to the definition of the *k-code* itself is that of *comments*. Specifically, the *pound sign (#)* is the comment marker: any characters that follow a pound sign on a line are ignored by the compiler. So, a Fibonacci procedure could now look like this one:

```
# Calculate and return the n-th Fibonacci number.
procedure 2 fib (2 n)
                []
{

  # Check for the base cases.
  ifthenelse (or (== n 0) (== n 1)) {

    # Base case: fib(0) = fib(1) = 1
    return 1

  } otherwise { # n > 2

    # Recursive case:
    #   fib(n) = fib(n-1) + fib(n-2)
    return (+ (fib (- n 1)) (fib (- n 2)))

  }

} # end fib()
```

Additionally, the compiler now allows procedure names to be referenced. For example, in the following line, the variable `ptr` is assigned the *address of the first instruction* of the procedure `main()`.

```
(= &ptr &main)
```

Finally, the compiler now takes some command line arguments to control which stub assembly and which standard library code it reads. By default, it will take its *stub* (the assembly that begins a program by setting up the stack and then calling `main()`) from the file whose pathname is `default-stub.asm`. Similarly, the compiler will include source code from a secondary file as a standard library, defaulting to the file whose pathname is `default-standard-library.k`. Both of these default files are included with the compiler.

However, on the command line, different files can be specified, and need to be for the kernel. For example, the provided script `compile-kernel.sh` (see Section 3.1) compiles the kernel like so, using the `-stub` and `-stdlib` switches to tell the compiler to use different files for the stub and standard library code.

```
$ java Compiler -stub ../system/kernel-stub.asm \\
                -stdlib ../system/kernel-standard-library.k \\
                ../system/kernel
```

3 Your assignment

3.1 Obtaining the tools

To obtain the new compiler, assembler, and simulator, login to the CS systems and follow these steps, preserving any existing work that you may have done on previous versions of this code.

```
$ cd cs26
$ cp -r project-6 old-project-6
$ tar -xzvpf ~sfkaplan/public/cs26/project-6-v4.tar.gz
$ cd project-6
$ ./build.sh
$ ./compile-kernel.sh
```

You now have the completed tools. As always, feel free to explore and modify any part of the system.

3.2 Your assignment

Parts of the kernel that you must write: Your task is, overall, to create a basic virtual memory system. Below are the steps that I recommend you follow (more or less) to lead yourself incrementally toward a completed system:

1. **Write bit-manipulator procedures:** You are going to need to control specific bit values in order to alter the status bits in your page table entries (PTE's). Therefore, it would be handy to write procedures that *get*, *set*, and *clear* an individual bit from a word. You will likely need to use the `SHFTL` *(shift left)* and `SHFTR` *(shift right)* instructions, along with the bitwise logical operators, in order to perform this task. That is, expect to write some k-code using assembly injections.

2. **Write PTE manipulator procedures:** You will need to read and write the various status bits of the PTE's. Make that easier by writing specific procedures for things like `get_status`, `set_valid`, etc. Also provide yourself procedures for things like `set_page_frame_number`.

3. **Write mapping and unmapping procedures:** Given a virtual page number and a physical page frame number, write a procedure that maps the former to the latter, setting some desired status bits, by manipulating some page table.

4. **Write a page table creator:** Each new procedure will need its own page table. Write a procedure that:

   (a) Obtains two free, consecutive page frames (see the `allocate_page_frame` procedure described below).

   (b) Maps those page frames into the kernel's virtual address space.

   (c) Clears the entries for virtual page numbers `0x00` through `0xbf` (the user processes's pages).

(d) Sets the entries for virtual page numbers `0xc0` through `0xff` to be *indirect*, with the page frame number set to that of the *kernel page table*. (See the `_kernel_page_table_pa` variable described below.)

(e) Returns both the virtual and physical addresses of the new page table: the former for in-kernel manipulation of its entries, the latter for setting the page table register.

5. **Write a procedure to create page frame table:** When the kernel begins, a procedure will be called to create a table, indexed by page frame number, that provides critical information about that page frame's status. Specifically:

   - `procedure 0 create_page_frame_table (2 device_table_ptr)`: Find RAM in the device table (which is already mapped into the virtual address space). Create an array that holds **one byte** per page frame. This array will be a table, indexed by page frame number, that stores the state of each frame.

     Then scan the kernel page table (only the kernel's portion), and for each valid mapping, grab the corresponding page frame number. Use that page frame number as the index into the page frame table, and mark the page as *used* by setting the least significant bit (that is, bit 0) of the entry.

     Finally, set the global variable `page_frame_table` to the virtual address at which this array begins.

6. **Write a procedure to allocate page frames:** Any time part of the kernel needs to allocate a page frame, it should call this method that you will write:

   - `procedure 2 allocate_page_frame ()`: Use the page frame table to find a free page frame. Mark that page frame as begin *used* by setting the page frame table entry's bit 0. Return the physical address at which that page frame begins.

There are **likely to be other parts of the kernel you must write**, and I will add descriptions of those to this document later. In the meantime, the above tasks are a good place to start.

What you do *not* need to do: I am providing a number of components to this system on which you should build. Specifically:

1. **Kernel initialization:** You do **not** need to writing the initialization portion of the kernel. Starting with version 2 of the code, the beginnings of a kernel are provided. Currently, the kernel compiles and boots, and then goes into an infinite loop. Its *stub code*, which gets the kernel going and can be seen in `system/kernel-stub.asm`, creates the kernel page table, creates a stack, maps these into their correct virtual locations, and then jumps to `main()`. That procedure is compiled k-code that begins the "normal" part of the kernel.

2. **Kernel page table addreses:** You do **not** need to do anything special to obtain the virtual or physical address of the kernel page table. Just use the global variables `kernel_page_table_va` and `_kernel_page_table_pa`. They will already be set to the correct values.

3. **Initializing the trap table:** I will create a procedure that initializes the trap table. You are likely, however, to need to modify this procedure if you add new interrupt handlers of your own.

4. **Loading executable images, starting processes, and scheduling:** I will provide the code that will map the executable image ROM's into the kernel's virtual address space, copy their contents into available newly mapped RAM pages, create a new *process object*, and then start that process running. I will also handle clock interrupts to perform pre-emptive scheduling. *Warning: This feature, sadly, is still not implemented.*

This list is not at all exhaustive, and more pieces will be added in later revisions.

Running the simulator: Note that running the simulator is a bit different than it used to be. Specifically, you specify, as a RAM size, the number of 256 byte pages that RAM should store, rather than the raw number of bytes used. So, when you run the simulator now, it may well look something like this:

```
$ java VirtualSystem 64 bios.vmx kernel.vmx
                        application-1.vmx application-2.vmx
```

4 How to submit your work

Use the `cs26-submit` command to turn in your programs. Specifically, move into your `cs26` directory and submit **your entire project directory**, thus allowing me to test your code with any modifications you may choose to make to the assembler or simulator:

```
$ cd ~/cs26
$ cs26-submit project-6 project-6
```

This command will submit, for this project, everything from your `project-6` directory.

This assignment is due at **5:00 pm** on **Friday, May 15**—the final day of final exams.