Notes for cs452
The Context Switch

Bill Cowan
University of Waterloo

This document was originally intended to list the sequence of steps by which I create context switches. But I have now made three successful, and different context switches.

- The first was done in a single assembly language file, so that I could get an intimate view of programming in assembly language for the ARM, and did nothing but context switch back and forth between different parts of the code.
- The second was done in several functions that started their life as C functions, were compiled into assembly language and the assembly language edited. This gave me a good feel for how the stack frames produced by the compiler have to be integrated into assembly code. It also improved my understanding of how much can be done in C, which I think should be maximized. For example, I initially created and initialized a task descriptor in assembly, then created C that would give me exactly the same code.
- The third was done as I expect you are doing, minimizing the amount of assembly language you need to write. This is a smart move: assembly language is hard enough to write, and even harder to debug.

I don't expect you to do anything like what I did: it's what is necessary to have enough confidence to answer questions in class.

Many of you will find this description insulting in its detail. So be it: most of you are much better programmers than I am.

0.1. Conventions.

1. All programming words, ones you put in a program or type at the shell are displayed in `courier`, with the capitalization you will use when programming.

A. Initial State

You should verify the initial state to make certain that it is exactly as I describe.

A.1. Using Assembly Code

The initial state you care about, such as the state of the caches and memory, require information stored in the co-processor which, as far as I know, requires assembly code. Here is how I do it.

First I start with a C program like the one following.

```
/*
* mmutest.c
*/

#include <bwio.h>
#include <ts7200.h>

int main( int argc, char* argv[] ) {
     int data, *junk;

     data = (int *) junk;
     bwsetfifo( COM2, OFF );
     bwputstr( COM2, "Page table base: " );
     bwputr( COM2, data );
     bwputstr( COM2, "\n" );
     return 0;
}
```

Then I compile it into assembly. The part I care about is the following, where the value of `data` is acquired and printed.

```
main:
     ...
     ldr  r3, [fp, #-24]
     str  r3, [fp, #-28]
     ...
     mov  r0, #1
     mov  r1, r3
     bl   bwputr(PLT)
```

This shows me how to print out what I get in hexadecimal. I substitute the second mov with a move from the co-processor into `r1`, followed by a second `bwputr` because I want to see the contents of another register, to get the following

```
     mov  r0, #1
     mrc  p15, 0, r1, c1, c0
     bl   bwputr(PLT)
     mov  r0, #1
     mrc  p15, 0, r1, c2, c0
     bl   bwputr(PLT)
```

Then I compile the assembly file, and run the result on the ARM. The result is printed out on the RedBoot display.

A.2. The State of the Memory

When RedBoot hands the processor over to your program the memory is mapped to the following virtual addresses.

```
RAM:
     0x00000000 - 0x01ffffff
ROM:
     0x60000000 - 0x607fffff
Device registers:
     0x80800000 - 0x808fffff
```

You can quite successfully develop a kernel without changing this mapping. The physical RAM is in four banks.

```
Physical RAM:
     0x00000000 - 0x007fffff
     0x01000000 - 0x017fffff
     0x04000000 - 0x047fffff
     0x05000000 - 0x057fffff
```

A.3. The State of the Cache Memory

The cache memory is disabled, both for instructions and for data.

B. Preliminaries

B.1. kernel.c

First I create a program called `kernel.c`, which does nothing but print `Hello, world.`

B.2. MKFILE

Next I create a `Makefile` to compile it. I want to create one that would expand easily as the kernel increased in complexity.

Before a hour is out I am totally fed up with `gmake`, and create a Plan9 program development environment so that I can use `mk`, `rc` and `acme`. Many of you will find this perverse, but it's easiest to work with what we are used to. Suffice it to say, fifteen minutes later I have a `Mkfile`.

B.3. KEYBOARD

I also thought that I would probably be doing a lot of typing, so I spent some time finding a better keyboard and adjusting the distance to the monitor. I am still looking for a three-button mouse without a scroll wheel. (Update. I have found one and am much happier and more productive.)

C. BABY STEPS

In what follows I build up the context switch in steps. I find most parts of the process to be non-trivial, but some work the first time I try them. To find out what is going on I mainly rely on three lines of assembly code that I stick in wherever I want to know the value of a register or memory location.

```
mov  r0, #1
mov  r1, sp
bl   bwputr
```

This example prints out the stack pointer, and because the io is busy-wait, does so even if the following instruction completely trashes the state of the system.

One thing to be careful about, however. `gcc` is not aggressive about preserving lots of registers on entering a function, so you should look at the assembly code of `bwputr` to find out what you can count on being preserved across the output. `gcc` with optimization off is quite good about separating its register use into discrete blocks that read from memory at the beginning and write to memory at the end. This is a benefit for you; the extra machine cycles are unimportant compared to your debugging time.

I wrote a busy wait IO function that showed me a task descriptor, and the top of its stack, and found it very helpful. It was possible to write it entirely in C, based on my C subroutine to make and initialize new tasks.

C.1. Crawling

I am trying to make enough code to support a non-trivial context switch, but no more, so I start with a slight modification of the kernel I keep writing on the board:

```
int main( int argc, char *argv[] ) {
     // declare kernel data structures

     initialize( tds );       // tds is an array of TDs
     for( i = 0; i < 4; i++ ) {
          active = schedule( tds );
          kerxit( active, req );   // req is a pointer to a Request
          handle( tds, req );
     }
     return 0;
}
```

These functions are mostly placeholders. `initialize` initializes a single task descriptor (`TD`) for a task called `first`, and puts the kernel entry point into the `swi` jump table; `schedule` returns a pointer to this `TD`; `handle` does nothing. I only go around the loop four times because it is marginally quicker if I can avoid rebooting the CPU all the time. (I am a scientist, not a mathematician. If a thing works four times it's correct… until it's incorrect.)

C.2. Toddling

All the action is in kerxit. I start out with a C function like this one.

```
void kerxit( TD *active, Request *req ) {
     bwprintf( COM2, "kerxit.c: Hello.\n\r" );
     bwprintf( COM2, "kerxit.c: Activating.\n\r" );
     kerent( );
     bwprintf( COM2, "kerxit.c: Good-bye.\n\r" );
}
```

It's a function pointer given `kerent` as its value that I put into the `swi` jump table. The `bwprintf` output is used to help me interpret the many lines of output I end up generating. `kerent` is an empty function.

When I compile this into assembly code I care about one line, which is

```
bl   kerent
```

This is where the context switch will occur. Before the context switch I must, in order,

1. push the kernel registers onto its stack;
2. change to system state;
3. get the `sp`, `spsr` and return value of the active task from its `TD`;
4. pop the registers of the active task from its stack;
5. put the return value in `r0`;
6. return to svc state;
7. install the `spsr` of the active task; and
8. install the pc of the active task.

After the context switch I must, in order,

1. acquire the arguments of the request, which are in registers that might be over-written;
2. acquire the `lr`, which is the pc of the active task;
3. change to system state;
4. overwrite `lr` with the value from 2;
5. push the registers of the active task onto its stack;
6. acquire the `sp` of the active task;
7. return to `svc` state;

8. acquire the `spsr` of the active task;
9. pop the registers of the kernel from its stack;
10. fill in the request with its arguments; and
11. put the `sp` and the `spsr` into the `TD` of the active task.

The order of these operations can be varied a little, but not much. For example, reading or writing the `TD` of the active task is only possible when the fp of the kernel is available, and it is unavailable from step 4 before the switch until step 9 after it.

To get these important sequences correct I find pairs of before and after operations that exactly undo each other, and then add them in pairs while leaving the branch-and-link to kerent.* At each point in the process the 'kernel' should function without change. I use lots of output to make certain that every detail is correct; you are probably less cautious.

C.3. Running

I now replace the body of kerent with the instructions in kerxit following

```
bl   kerent
```

Simultaneously, I change

```
bl   kerent
```

to

```
b    kerent
```

If all is well, the behaviour of the kernel is unaltered.

C.4. Walking

At this point we have completely debugged the save/restore code of the context switch without going off the sidewalk. My first user task looks something like this

```
void first( ) {
     bwprintf( COM2, "first.c: initializing\n\r" );
     FOREVER {
          bwprintf( COM2, "first.c: good-bye\n\r" );
          bwprintf( COM2, "first.c: hello\n\r" );
     }
}
```

where the execution starts at `first` and goes into the kernel just after good-bye. Later executions emerge from the kernel just before hello.

There are two small pieces of code to be added to get to first: step 7 & 8 of before jumps into first; an `swi` in first, plus steps 1 & 2 of after jump back to the kernel. I am very risk averse and want to do them one at a time. I notice that I can put

```
b    first
```

inside kerxit to get to the beginning of first, which, combined adding `swi` to `first`, should leave me with a functioning program.

When I have this working I then I put in the other part, and check that the first entry to first initializes and that later ones start at hello.

C.5. Marching

At this point there is nothing very unusual left to do for assignment 1.

- Arguments and return values need to be put in the right places.

* Steps 7 & 8 of before and steps 1 & 2 of after are omitted.

† Every parent knows that from one year until four years children run, after which they start to walk.

- If you are choosing the kernel function using the argument allowed by `swi` you can find it at `[lr, #-4]`.
- You need to put `swi` inside wrapper (also called stub) functions with the approved names.
- You need to implement the priority queues and the scheduling algorithms. The usual way of implementing queues in an OS like this one is to put pointers into the `TD`, like so.

```
typedef struct tskdes {
     //other junk
     tskdes *pqnxt;
     tskdes *pqprv;
     //other junk
} TD;
```

  Then if you make the queue circular the kernel needs only an array of pointers to `TD`s, one for each priority. Alternatively, you could make the list at each priority singly-linked, in which case the kernels needs an array or pairs of `TD`s, pointing to the head and tail of each queue.

D. Hacks

There are a few problems I 'solved' where there ought to be a better solution.

Function pointers to kernel entry and first. I had to put in relocation constants for function by hand. Possibly this is just a matter of finding the right switch on the compiler or linker. One thing a little more elegant would be to improve the linker script so that it always puts `main` first, the grab the pc of the first instruction executed and use it for a program-introduced relocation value.

Labels in C code. It ought to be possible to get the compiler/assembler/linker to recognize them as global symbols, which would make it possible to fill in the `swi` jump table without the dreadful `kerxit`/`kerent` hack.

Mode barriers. Different stack and link pointers for different modes is very nice, but when you have a stack or link pointer in one mode that you want to access in another mode, you need to put it somewhere temporarily when you are changing mode. The `ip` is handy, but interacts badly with most debugging techniques. There must be better tricks.

The frame pointer. Memory is much more easily accessible when you have a frame pointer. The frame pointer is constant across modes, but changes when a stack is popped into registers. The most elegant way to get things across mode barriers is to use memory, which is least hacky when using a frame pointer. You could have the compiler allocate some memory on the stack of kerxit accessible by a frame pointer.

Think ahead. The context switch solution you get here will probably use `r0-r3` as scratch registers here and there. This is fine for `swi`, but won't work for hardware interrupts: you have to save all the registers, `r0-pc`, of the interrupted task. Consider how it might be done. A design here that doesn't require reworking will make things easier later. Remember that

the kernel does not need to save the scratch registers because you control exactly what happens after it is restored.