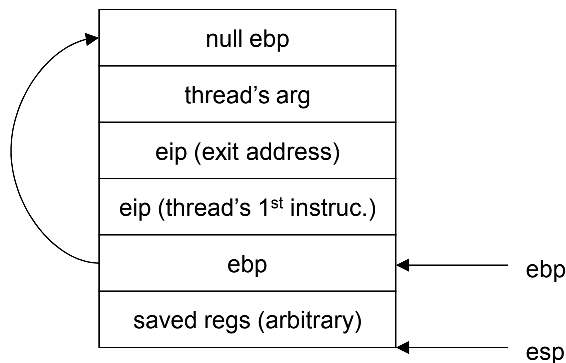
CS167 Homework Assignment 1 Solutions

Fall 2009

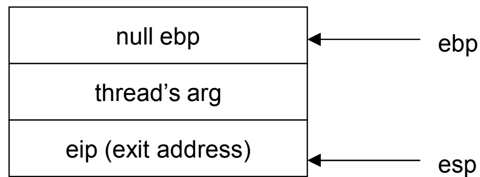
1. *The text, on page 3-10, describes how to switch from one thread to another. The implicit assumption is that the thread being switched to in the* switch *routine has sometime earlier yielded the processor by calling* switch *itself. Suppose, however, that this thread is newly created and is being run for the first time. Thus when its creator calls* switch *to enter its context, it should start execution as if its first routine had just been called. Show what the initial contents of its stack should be to make this happen. Assume an x86-like assembler language, as used in Chapter 3 of the text.*

We must set up the thread's stack so that when a return instruction is executed in its context, the thread's instruction pointer is set to its first instruction and the now-current stack frame contains the argument to the thread. It's also important (though not necessary for full credit on this problem) that if the thread returns from its first procedure, it returns to code that calls pthread_exit (or the equivalent).

So, the stack should be set up as follows:



When a return is executed in the context of this stack, the result is the following.



The thread is now executing its first instruction, which will push the frame pointer on the stack, save the current registers (whose contents aren't important), and allocate space on the stack for the local variables of its first procedure. If this procedure returns, the instruction pointer will be set to exit address, a location that contains a call to pthread_exit (what's not shown is setting up the stack to pass an appropriate argument to pthread_exit).

2. *Recursion, in the context of programming languages, refers to a function that calls itself. For example, the following is a simple example (assume that it's called only with appropriate argument values):*

```
int factorial(int n) {
   if (n == 1)
      return n;
   else
      return n*factorial(n-1);
}
```

Tail recursion is a restriction of recursion in which the result of a recursive call is simply returned — nothing else may be done with the result. For example, here's a tail-recursive version of the factorial function:

```
int factorial(int n) {
   return f2(n, 1);
}

int f2(int a1, int a2) {
   if (a1 == 1)
      return a2;
   else
      return f2(a1-1, a1*a2);
```

```
}
```

a. *Why is tail recursion a useful concept? (Hint: consider memory use.)*

With tail recursion, one can write recursive functions that use a minimum of stack space.

b. *Explain how tail recursion might be implemented so as to obtain the advantages mentioned in part a. (You don't need to supply the assembler code.)*

Each tail call reuses the current stack frame rather than pushing a new one on the stack. Thus placing a call simply involves restoring the stack and registers to the state they were in when the current subroutine was entered (on the x86 this means that the local variables are popped off the stack, saved registers are restored, and the saved ebp is popped from the stack and into the ebp register).

3. *Slides VI-6 and VI-7 (from pages 3-23 and 3-24 of the text) show the assembler code of main.s and subr.s. Describe what changes are made to the corresponding machine code (in main.o and subr.o) when these routines are processed by ld, forming prog. Be specific. I.e., don't say "the address of X goes in location Y," but say, for example, "the value 11346 goes in the four-byte field starting at offset 21 in xyz.o." Note that for the x86 instructions used in this example, an address used in an instruction appears in the four-byte field starting with the second byte of the instruction.*

As indicated in slide VI-8 (text page 3-25), the four bytes at offset 7 within main.o are updated to contain the address of aX. According to slide VI-12 (page 3-27), this value is 16384. The four bytes at offset 20 within main.o are updated to contain the PC-relative address of subr. The actual address of subr is 4132. To obtain the PC-relative address, we must figure out what the PC (or instruction pointer) would be when the instruction using this address is executed. From slide VI-6, the PC would refer to offset 24 within main. According to slide VI-12, main is at location 4096. Thus the PC's value would be 4120 and the PC-relative address of subr would be 4132−4120 = 12.

Similarly for subr.o: the four bytes at offset 7 are updated to contain the address of printfargs, which is 16388. The four bytes at offset 12 are updated to contain the PC-relative address of printf. The PC's value would be 4148. Since printf is at 4156, the PC-relative address is 8.

4. *Many C and C++ compilers allow programmers to declare thread-specific data (Section 2.2.4 of the text) as follows.*

```
__thread int X=6;
```

This declares X to be a thread-specific integer, initialized to 6. Somehow we must arrange so that each thread has its own private copy of it, initialized to 6. Doing so requires the cooperation of the compiler, the linker, and the threads library. Describe what must be done by all for this to work. Note that thread-specific data (TSD) items must be either global or static local variables. It makes no sense for them to be non-static local variables. You may assume that TSD items may be initialized only to values known at compile time (in particular, they may not be initialized to the results of function calls). Assume that only static linking is done — do not discuss the further work that must be done to handle dynamic linking. Note that a program might consist of a number of separately compiled modules.

A possible approach is for each thread to have a *TSD table* containing all its TSD items. Code to construct this table would be produced by the linker using information provided by the compiler. This table-construction code would be called each time a new thread is created.

To do this, the compiler would produce, for each separately compiled module, an entry in the object code's header containing the names of externally visible TSD items (i.e., those that are

non-static global variables), the total amount of storage required for all TSD items defined in the module, and a function to be called that, given the address of the storage allocated for the TSD items, initializes that storage. The linker would then, when linking all modules together, determine the total length of all the TSD items and create a function that allocates storage for a thread's TSD table and calls each of the per-module initialization functions to initialize this table. The address of this function would be stored in a global variables, such as *__TSD_init*. The *startup* routine, which initializes the first thread (before calling *main*), and *pthread_create* would each call *TSD_init*.

The final concern is how TSD is referenced when the program is running. The compiler might, for each separately compiled module, set up a table known as the *TSD offset table.* It is similar to the *global offset table* (slide VI-31, text page 3-29), but, unlike the global offset table which contains actual addresses, this table would contain the offsets of the TSD items within the TSD tables. It would be filled in by the linker once it has determined the layouts of the TSD tables. A thread would then access a TSD item by getting its offset from TSD offset table and then using that offset to access the item within the thread's TSD table.