SOLUTION NOTES

Compiler Construction 2001 Paper 6 Question 6 (AM)

This is the file as submitted to the Examiners in January 2001.

Bookwork. Here is what was lectured:

1 Object Modules and Linkers

We have shown how to generate assembly-style code for a typical programming language using relatively simple techniques. What we have still omitted is how this code might be got into a state suitable for execution. Usually a compiler (or an assembler, which after all is only the word used to describe the direct translation of each assembler instruction into machine code) takes a source language and produces an *object file* or *object module* (`.o` on Unix and `.OBJ` on MS-DOS). These object files are linked (together with other object files from program libraries) to produce an *executable file* (`.EXE` on MS-DOS) which can then be loaded directly into memory for execution. Here we sketch briefly how this process works.

Consider the C source file:

```
int m = 37;
extern int h(void);
int f(int x) { return x+1; }
int g(int x) { return x+m+h(); }
```

Such a file will produce a *code segment* (often called a *text segment* on Unix) here containing code for the functions `f` and `g` and a *data segment* containing static data (here `m` only).

The data segment will contain 4 bytes probably `[0x25 00 00 00]`.

The code for `f` will be fairly straightforward containing a few bytes containing bit-patterns for the instruction to add one to the argument (maybe passed in a register like `%eax)`) and return the value as result (maybe also passed in `%eax`). The code for `g` is more problematic. Firstly it invokes the procedure `h()` whose final location in memory is not known to `g` so how can we compile the call? The answer is that we compile a 'branch subroutine' instruction with a dummy 32-bit address as its target; we also output a *relocation entry* in a *relocation table* noting that before the module can be executed, it must be linked with another module which gives a definition to `h()`.

Of course this means that the compilation of `f()` (and `g()`) cannot simply output the code corresponding to `f`; it must also register that `f` has been defined by placing an entry to the effect that `f` was defined at (say) offset 0 in the code segment for this module.

Header information; positions and sizes of sections
`.text` segment (code segment): binary data
`.data` segment: binary data
`.rela.text` code segment relocation table: list of (offset,symbol) pairs showing which offset within `.text` is to be relocated by which symbol (described as an offset in `.symtab`)
`.rela.data` data segment relocation table: list of (offset,symbol) pairs showing which offset within `.data` is to be relocated by which symbol (described as an offset in `.symtab`)
`.symtab` symbol table: List of external symbols used by the module: each is listed together with attribute 1. undef: externally defined; 2. defined in code segment (with offset of definition); 3. defined in data segment (with offset of definition). Symbol names are given as offsets within `.strtab` to keep table entries of the same size.
`.strtab` string table: the string form of all external names used in the module

Figure 1: Summary of ELF

It turns out that even though the reference to `m` within `g()` is defined locally we will still need the linker to assist by filling in its final location. Hence a relocation entry will be made for the 'add `m`' instruction within `g()` like that for 'call `h`' but for 'offset 0 of the current data segment' instead of 'undefined symbol `h`'.

A typical format of an object module is shown in Figure 1 for the format ELF often used on Linux (we only summarise the essential features of ELF).

2 The linker

Having got a sensible object module format as above, the job of the linker is relatively straight-forward. All code segments from all input modules are concatenated as are all data segments. These form the code and data segments of the executable file.

Now the relocation entries for the input files are scanned and any symbols required, but not yet defined, are searched for in (the symbol tables of) the library modules. (If they still cannot be found an error is reported and linking fails.) Object files for such modules are concatenated as above and the process repeated until all unresolved names have been found a definition.

Now we have simply to update all the dummy locations inserted in the code and data segments to reflect their position of their definitions in the concatenated code or data segment. This is achieved by scanning all the relocation entries and using their definitions of 'offset-within-segment' together with the (now know) absolute positioning of the segment in the resultant image to replace the dummy value references with the address specified by the relocation entry.

(On some systems exact locations for code and data are selected now by simply concatenating code and data, possibly aligning to page boundaries to fit in with virtual memory; we want code to be read-only but data can be read-write.)

The result is a file which can be immediately executed by *program fetch*; this is the process by which the code and data segments are read into virtual memory at their predetermined locations and branching to the *entry point* which will also have been marked in the executable module.

3 Dynamic linking

Consider a situation in which a user has many small programs (maybe 50k bytes each in terms of object files) each of which uses a graphics library which is several megabytes big. The classical idea of linking (*static linking*) presented above would lead to each executable file being megabytes big too. In the end the user's disc space would fill up essentially because multiple copies of library code rather than because of his/her programs. Another disadvantage of static linking is the following. Suppose a bug is found in a graphics library. Fixing it in the library (`.OBJ`) file will only fix it in my program when I re-link it, so the bug will linger in the system in all programs which have not been re-linked—possibly for years.

An alternative to static linking is *dynamic linking*. We create a library which defines *stub* procedures for every name in the full library. The procedures have forms like the following for (say) `sin()`:

```
static double (*realsin)(double) = 0;  /* pointer to fn */
double sin(double x)
{   if (realsin == 0)
    {   FILE *f = fopen("SIN.DLL");    /* find object file */
        int n = readword(f);           /* size of code to load */
        char *p = malloc(n);           /* get new program space */
        fread(p, n, 1, f);             /* read code */
        realsin = (double (*)(double))p;  /* remember code address */
    }
    return (*realsin)(x);
}
```

Essentially, the first time the `sin` stub is called, it allocates space and loads the current

version of the object file (`SIN.DLL` here) into memory. The loaded code is then called. Subsequent calls essentially are only delayed by two or three instructions.

In this scheme we need to distinguish the stub file (`SIN.OBJ`) which is small and statically linked to the user's code and the dynamically loaded file (`SIN.DLL`) which is loaded in and referenced at run-time. (Some systems try to hide these issues by using different parts of the same file or generating stubs automatically, but it is important to understand the principle that (a) the linker does some work resolving external symbols and (b) the actual code for the library is loaded (or possibly shared with another application on a sensible virtual memory system!) at run-time.)

Dynamic libraries have extension `.DLL` (dynamic link library) on Microsoft Windows and `.so` (shared object file) on Linux. Note that they should incorporate a version number so that an out-of-date DLL file cannot be picked up accidentally by a program which relies on the features of a later version.

The principal disadvantage of dynamic libraries is the management problem of ensuring that a program has accept to acceptable versions of all DLL's which it uses. It is sadly not rare to try to run a Windows `.EXE` file only to be told that given DLL's are missing or out-of-date because the distributor forgot to provide them or assumed that you kept your system up to date by loading newer versions of DLL's from web sites! Probably static linking is more reliable for executables which you wish still to work in 10 years' time.