Application Optimization on Linux
Tutorial Material

Ulrich Drepper*
Cygnus Solutions
Sunnyvale, CA

December 14th, 1999

*drepper@cygnus.com

Contents

1 What is this about?

Most programmers think that the programs they write are fairly well written and perform as good as possible. In most cases this is not correct. Many people *think* they know how to do it right but they most probably miss something. Writing optimized programs is a learning process. One learns about a new technique every time one looks sharply at its own code and thinks about the interactions with the underlying libraries or the processor.

In this tutorial manual we will discuss several optimization techniques which the author learned in the past years of programming. It is certainly not a complete list nor is it a structured approach. It is merely a list of concrete examples and techniques. All examples are given in C but most of the techniques can be applied to C++ as well. Using templates as allowed in standard C++ is not at all covered.

Based on a concrete program (which was written explicitly to show bad code) the different sections of the manual point out problems in the example code and propose solutions for better code. This approach hopefully allows spotting optimization opportunities in own programs more easily.

This paper is *not* about optimizations performed in the optimizer of the compiler. This is a completely different field and a lot of literature exists describing the possibilities. What we will describe are changes to the source code and ways to find out where they are needed. The remainder of the paper will cover the following aspects.

- Using optimizations performed at compile-time
- Helping the compiler to generate better code
- Knowing the libraries and understand the function implementation
- Writing better code in the first place
- Profiling

These will not necessarily be introduced in this order and each each of the optimizations not necessarily in one place. Instead an approach one would take in practice is taken. The steps a programmer has to make are described in detail.

To read and understand the following text it is necessary that the reader knows how the preprocessor works, has advanced knowledge about the C programming language itself, and preferably knows the functions of the C language library. The text also describes a few machine specific optimizations but it is OK to ignore these points if one is not familiar with the described architecture.

1.1 Limitations

Programmers hopefully follow the advice of Donald Knuth and first finish writing their applications before starting to optimize them. A good programmer can combine this to some extend but in the end there will always be the possibility for some optimization.

But one should not expect wonders. Some algorithms have a complexity which require an non-linear increase of runtime with increasing size of the input (remember your studies in Complexity Theory). This problem cannot be overcome other than by replacing the algorithm with a more efficient one. If there is a function in the system library to perform the task chances are good that this implementation uses the most efficient algorithms and one can safe a lot of work buy using them. We will *not* discuss finding more efficient algorithms here but we will point out a few system library functions which perform good jobs.

Another part of "hardcore-optimization" which we will not discuss is optimization of memory layout. Programs very every part of a percent in performance increase counts are optimized to have code and data optimally layed out in the system memory. This is, e.g., often the case for simulation programs where it is crucial to place the variables containing matrices not in the wrong places and where most of the time is spend in a few small inner loops which better should not aligned correctly in memory. Tools to handle this are slowly appearing for Linux but they are not yet available (at least not freely).

1.2 The Example Program

To show the different optimization this tutorial is structured around an example program. All the examples are taken from this program which immediately allows to check the effect of the changes. The code of the program is printed in appendix **??**. The program is an application of a simple source code browser. Using a web browser it is possible to look at files and follow cross-references for objects defined in other files. To do this an implementation of the HTTP is implemented as well as functionality to read large numbers of files, extract information, and rewrite them. In short: the work is similar to what all these "Internet applications" have to do and therefore is hopefully a good choice.

It is not absolutely necessary to understand the program completely. Especially the network handling code can be safely ignored. The fundamental operations are implemented in the kernel and so do not leave much room for optimizations. One should only understand the concept of a system call and the associated costs which directly lead to the conclusion: the fewer system calls, the better.

The example program is for convenience reasons contained in one single file of name `main.c`. Therefore generating an executable can happen using the command

```
gcc -D_GNU_SOURCE main.c
```

which produces an executable named `a.out`. In the following text we will several times see compiler invocations like this and it necessary to remember that the file name of the program is `main.c`.

2 A First Look

In the worst case scenario the task is to optimize a program which one has not written (or written only in parts). In this case one first has to make oneself familiar with the program. In large programs one it is not even necessary to not having written all the code since the interaction between the different pieces of the program often introduce unexpected costs.

Therefore it is necessary to have the ability to get first a rough understanding of the performance of the program. If possible the program should be investigated when performing tasks seperately. This is possible with the example program which has two clearly distinguishable phases: the startup phase where the data is prepared and the actual phase where the program is running as a server and is serving data.

Sometimes it is necessary to help the system tools to perform the analysis. The example program is running by default as a daemon in the background. It is detached from the terminal. To help debugging and also profiling it is useful to prevent the program from running in the background. To do this ones often finds code like the following in programs:

```
/* If we are not debugging, daemonize now.  */
if (! do_debug)
  daemon (1, 0);
```

The function `daemon` comes from the system library and is responsible for "daemonizing" the application (i.e., detaching the program from the terminal and running it in the background). This function should only be called if one is not debugging (or profiling) which is here controlled by the variable `do_debug`.

The program was also prepared with an option `-s` for the profiling. This option causes the program to stop after the setup is finished. This allows, as requested above, to examine the startup phase and the server phase of the program independently.

2.1 The Steps to Take

Now is the question what possibilities to learn about the program exists. First of course is the possibility to run then program and see how it performs. If the program already runs good enough (on the kind of machine it is intended to run) then it might be better to spend time on something more useful. In the case of this example program this is not the case, it runs really slow once it has to handle many and also large files.

Once one found out that the program does not run as expected there are different possibilities to find out where the time spend (this is what is normally referred to as "profiling").

- Stepping through the execution of a program. This can happen by adding instructions to the program code which announce that the program reached a certain point or by running the program inside a debugger which lets the user step through the program.

  Needless to say that this is not a very efficient way, especially for large programs.

- Use the profiling capabilities of the dynamic linker. The dynamic linker implementation in glibc 2.1 and later has the ability to track calls to functions in a shared object. This can be used to get some information of how a program uses a library (which functions, how long). It also helps to analyze the implementation of the library itself.

  In out case here it is mainly interesting in the first case. We can very easily see which functions are used since this profiling techniques does not require recompiling the application. It can be, as it should normally be, linked dynamically to the library in question. In the following we are interested in the use of the C library since this is the only one we are linking to.

- Using full-blown profiling. This is implemented using the help of the compiler. It inserts into the generated code at the begin of every function a call to a function named `_mcount`. The program must also be linked with a special function (done if `-pg` is passed to the linker) which enables code which starts collecting the profiling information when the program is started and writes out the results when it is finished.

  To get very detailed information about all functions involved one should also link against special versions of the libraries involved. These have normally names ending in `_p`. Beside requiring these special libraries being available it also means that these libraries must be linked in statically. The profiling mechanisms as currently implemented are not working for functions in shared objects.

- Using Coverage analysis it is possible to determine which parts of the program are used and how often. It is enabled by compiling the program using the options `-fprofile-arcs -ftest-coverage -a` and by adding `-a` when linking the program. While unused code has direct impact on the runtime it might have secondary effects if the unused code is worsening the memory and cache behavior.

  Coverage analysis also helps to determine how conditional jumps (on assembler-level) behave. One can find out often which branch is taken. With some knowledge about the architecture it is possible to optimize the execution flow so that the most often taken path is executed the fastest.

The information gathered by the code coverage code is also written out at program exit time. Running the program more than once will add more information to the file. The information can also be used as additional input to the program analyzing the profiling data.

- Specialized analysis of, e.g., the memory allocation behavior. Memory allocation is a really important part of each non-trivial program and it often provides large potentials to improve the program. The `memprof` program can help here.

We will not get into the first listed possibility but will now show how to perform the other three methods and explain how to interpret the output.

2.2 Using `sprof`

`sprof` is the tool which is used to interpret the output data of the profiling activities of the dynamic linker. Before it can do anything the program must of course run. Since the dynamic loader is used we must make sure the program is using it. This is quite easy: avoid using the `-static-` option while linking the application.

```
gcc -D_GNU_SOURCE main.c
```

After doing this we can execute the program but we have to add something to signal the dynamic loader that we want to profile. Otherwise the program would behave like normal. (Did I mention we assume a Bourne shell?)

```
LD_PROFILE=libc.so.6 ./a.out -s /usr/src/libc
```

The `LD_PROFILE` environment variable signals the dynamic loader that the user is interested in profiling the calls to the library `libc.so.6` (which is the standard C library on Linux systems; except on Alphas where it would be `libc.so.6.1`). The output is written to a standard directory (normally `/var/tmp`). The `sprof` program which is used the examine the profiling data will automatically find the data file. To see the profiling data call

```
sprof libc.so.6 /var/tmp/libc.so.6.profile
```

The output is a familiar sight for those used to `gprof` (see section 2.3). It consists of two parts:

- Flat profiling data. Here each of the functions of the shared object which actually was used is listed with the number of calls of this function, the time spend executing it, an incremental sum, and the average time spend it it. The beginning of the file for our example program looks like this:

```
Flat profile:
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 99.36     39.03    39.03309569425     0.13           strcmp
  0.15     39.09     0.06    39797     1.51           __strtoul_internal
  0.13     39.14     0.05        0     0.00           getdelim
  0.10     39.18     0.04    49701     0.80           malloc
  0.08     39.21     0.03    84494     0.36           index
  0.08     39.24     0.03      340    88.24           read
  0.05     39.26     0.02    49598     0.40           getline
  0.03     39.27     0.01    76238     0.13           strlen
  0.03     39.28     0.01    39797     0.25           strtoul
  0.00     39.28     0.00    94297     0.00           flockfile
  0.00     39.28     0.00    94297     0.00           funlockfile
  0.00     39.28     0.00    44698     0.00           feof
  0.00     39.28     0.00    27300     0.00           strcpy
```

- Representation in a "graph". It is graph, not graphical! Also the graph-aspects of the output are limited. The output simply shows for each used function the places from where the were called and functions used in it, together with the proportion contributed to the overall runtime. The beginning of this part looks like this:

```
index % time    self  children    called     name

                0.00    0.00       0/0           <UNKNOWN>
[0]      0.0    0.00    0.00       0         _dl_sysdep_start@@GLIBC_2.0 [0]
                0.00    0.00       1/2           __deregister_frame_info [2076]
                0.00    0.00       1/1           exit [188]
                0.00    0.00       1/1           fclose@@GLIBC_2.1 [357]
                0.00    0.00309569425/309569425   strcmp [601]
                0.00    0.00   39797/39797       strtoul [235]
                0.00    0.00   84494/84494       index [600]
                0.00    0.00   49598/49598       getline [348]
                0.00    0.00   44698/44698       feof [394]
                0.00    0.00       1/1           fopen@@GLIBC_2.1 [362]
                0.00    0.00       2/4           cfree [541]
                0.00    0.00       1/1           _xstat [1029]
                0.00    0.00       1/1           __errno_location [13]
                0.00    0.00       4/4           strcat [599]
                0.00    0.00   27300/27300       strcpy [603]
                0.00    0.00   49698/49701       malloc [539]
                0.00    0.00   76238/76238       strlen [610]
                0.00    0.00       1/1           argp_parse [1499]
                0.00    0.00       1/1           mcheck [580]
                0.00    0.00       1/2           __register_frame_info [2072]
                0.00    0.00       1/1           __libc_start_main [7]
```

This is the output for only one function, the one with index zero (see first column). The function name in the `name` column is the one which is shifted four positions to the left, relative to all the other names. The numbers in brackets behind the names are the indices of the functions called in this function or the function, from where this function is called. In this case the function is called only from one place, `UNKNOWN`. This indicated that the dynamic loader was not able to determine the function from where it was called.

The `called` column consists of two values: the left number specifies the number of calls which fall into this context. The right number specifies the total number of calls. E.g., the function `malloc` is called 49701 times in total, of which 49698 calls happen from the current function.

The name of this function (`_dl_sysdep_start@@GLIBC_2.0` is itself a bit strange. The numbers in this profiling data indicate that these are actually the numbers of the program itself. And this is in fact true. The problem is that the dynamic loader has not enough information to determine this. Not a real problems ones one is used to this.

We will discuss the data found by running `sprof` later, once we discussed the other profiling methods as well.

2.3 Using `gprof`

Profiling using `gprof` is very old and widely available. Very early BSD systems supported it. Unlike the use of `sprof` this profiling method can provide information about the whole program. The drawback is that all function about which information is wanted must be linked in statically. This not only increases the binary size and increases the link time, it will also not exactly reflect the profile of the program running in its final shape (i.e., dynamically linked). But this is important only for the finest tuning methods (like code and data object layout) and can be ignored here.

To prepare a program for profiling using `gprof` all files one is interested in must be compiled using the `-pg` option. This instructs the compiler to insert a call to a function named `_mcount` at the beginning of every function. This function gets passed information about the current function and the place of the call to this function. With this information it is later possible to construct a call graph where we can determine which function was called from which other function and how often.

Once all object files are translated we have two possibilities to link the application. If we are not interested in profiling the C library functions as well the parameter `-pg` must be passed here as well. If we want to profile the C library functions as well (to learn very detailed where the time is spend) the option `-profile` must be used. The later is useful to look inside the functions of the program we want to profile. If it is not given the functions in the own program which only call C library functions are black boxes and one will not learn much about their execution.

Libraries other than the C library have no such easy handling. They cannot be selected with a single linker option. Instead one has to give the name of the profiling variants of these functions explicitly. The names of these libraries normally end with `_p.a`. So the profiling version of the math library is named `libm_p.a` and instead of `-lm` one has to give `-lm_p` at the linker command line.
The binary build this way then can be executed normally:

```
./a.out
```

The program executes normally and when it is done the current directory contains one new file with the name `gmon.out`. This file contains the profiling data which got written out just before the program terminated. To analyze the data the `gprof` program has to be used. It has, like `sprof`, two basic modes to display the information. Data is emitted in both formats if no parameter is given. If `--flat` is given only the flat profiling data is displayed, with `--graph` the graph form is used. The output looks the same as in the example of the last section so we save repeating it here.

`gprof` has one more output mode but we will delay discussing this in to the next section. Beside this there are several options to modify the output and before honestly starting to use this program one should spend a bit time learning about them.

2.4 Coverage analysis

Mostly needed in testing and quality engineering coverage analysis also helps optimizing an application. During coverage analysis information not only about the functions calls is recorded but also about the execution paths inside the functions. The basis for this analysis are basic blocks. Basic blocks are parts of the generated code which contain no jumps and no jump targets. I.e., wherever the linear execution path can be changed (or changed to) a basic block starts or ends.

If now the program is run (if necessary multiple times) with all possible input data sets the program can record which basic block is executed how often and which exit from each basic block is taken (if there is more than one). The number of execution can also be zero in which case one has discovered dead code.
To get this data the program has to be compiled specially again.

```
gcc -fprofile-arcs -ftest-coverage -D_GNU_SOURCE main.c
```

Once the program run finished it is possible to analyze the data using the `gcov` program. Once start `gcov` with the name of a source code file and gets as the result a listing of this file annotated with information about the execution.

```
$ gcov -l -b main.c
```

```
 25.36% of 485 source lines executed in file main.c
 23.83% of 386 branches executed in file main.c
 18.39% of 386 branches taken at least once in file main.c
 26.43% of 333 calls executed in file main.c
Creating main.c.gcov.
```

As the last line said, the resulting information is stored in the file `main.c.gcov`. Beside the source code of the file each line for which actually code was generated, there is at the beginning of a line a number specifying how often a each line got executed. For lines in the same basic block the number is obviously the same. Lines which were not executed at all are marked with `######`. In addition the `-b` parameter causes `gcov` to emit information about the behavior of the program at branches. An excerpt of the output is:

```
                /* Search for a symbol.  */
                static inline struct known_symbol *
                symbol_search (const char *symbol)
                {
       39880      struct known_symbol *runp = known_symbols;

       39880      while (runp != NULL && strcmp (symbol, runp->symbol) != 0)
branch 0 taken = 0%
call 1 returns = 100%
branch 2 taken = 3%
branch 3 taken = 100%
branch 4 taken = 100%
   309529365        runp = runp->next;
branch 0 taken = 3%

       39880      return runp;
branch 0 taken = 100%
      ######    }
call 0 returns = 0%
```

This information is a bit cryptic and one has to learn to interpret it (and it helps to know a bit about code generation). Even with this is is not always possible to exactly figure out everything. Looking at the actually generated code is sometimes necessary. The first two lines of the function are executed for all invocations of the function. So we know that the function was called 39880 times (we could have learned this also from the `gprof` profiling). But then it starts getting exciting. The `while` instruction at the begin of a loop is the start of a new basic block and because of the non-trivial nature of the `while`-expression there are five branches in total. Using the information above it is not really easy to figure out what these branches are. Writing down the branching factor is not always helpful. Since not three possible jumps can be taken 100% of the time it should be obvious that the numbers are not adding up. Instead the output above must be interpreted as "if branch 0 is not taken, call 1 is made 100% of the time" and "branch 2 is not taken, branch 3 is taken 100% of the time". But even with this clarified the numbers look strange. With a short look at the algorithms one sees the `branch 0 taken = 0%` means that there is never a member in the list `known_symbols` when the function is entered. This cannot be true. To see actual numbers one has to add the parameter `-c` to the `gcov` command line above. The so generated file contains the following section:

```
                /* Search for a symbol.  */
                static inline struct known_symbol *
                symbol_search (const char *symbol)
                {
       39880      struct known_symbol *runp = known_symbols;

       39880      while (runp != NULL && strcmp (symbol, runp->symbol) != 0)
branch 0 taken = 22445
call 1 returns = 309546800
branch 2 taken = 309529365
branch 3 taken = 17435
branch 4 taken = 39880
   309529365        runp = runp->next;
branch 0 taken = 309529365

       39880      return runp;
branch 0 taken = 39880
      ######    }
call 0 returns = 0
```

Now we see that the 0% of the first output does not actually means zero branches. It simply means a number $< 1\%$. With this additional information it should be possible to "decrypt" to output. We should do some number magic to get a clearer image. We know that the loop has to be left exactly 39880 times (we know this since this is the number of invocations and by the fact that the program terminated successfully). This number does not appear directly in any of the counts but $22445 + 17435 = 39880$. I.e., branch 0 and branch 3 are the jumps out of the loop. From this we can learn that branch 0 is taken if `runp != NULL` is untrue and branch 3 is taken if `strcmp (symbol, runp->symbol) != 0` is untrue. Call 1 is obviously the call to the functions `strcmp` and since this function always returns we know now that the function was called here 309546800 times.

From this we can learn that branch 2 taken if strcmp determines the strings are not equal since $309529365 + 17435 = 309546800$. Now the only question remaining is what is branch 4. It is executed exactly once for each call of the function. Here it is necessary to remember that we haven't enabled optimization during the compilation and so gcc generates very stupid code. Especially it generates code that has exactly one end node for each basic block (which makes optimization easier). Therefore we can safely assume that branch 4 is the branch to the instruction following the loop.

This is already very exciting but it is possible to do more with the output of the coverage analysis. One thing which happens at the end of the program runs is that a file named bb.out is created. It contains information about each basic block in the program and it is possible to convert this file into a format which is then useful for the gprof program. Unfortunately the script to do this is normally not installed. It is part of the gprof sources in the GNU binutils package and is named bbconv.pl. To make this information really useful one has to enable profiling and code coverage analysis at the same time. So we compile the program again

```
gcc -c -g -fprofile-arcs -ftest-coverage -a -p -D_GNU_SOURCE main.c
gcc -profile main.o
```

Note the extra parameter -a in the linker invocation. This one causes the program to write out the data in one single file bb.out. After the so generated program ran the information in the bb.out file must now be (for now) converted

```
bbconv.pl < bb.out > bb.mon
```

and the resulting file can be processed together with the profiling information if gprof is called with the --annotated-source and --list parameters. The output for the function we looked at in the last two examples looks like this:

```
                /* Search for a symbol.  */
                static inline struct known_symbol *
                symbol_search (const char *symbol)
 39797,39797 -> {
                  struct known_symbol *runp = known_symbols;

309567357,22398,  while (runp != NULL && strcmp (symbol, runp->symbol) != 0)
   309567357 ->     runp = runp->next;

       39797 ->   return runp;
       79594 -> }
```

This output does not include any new information. The numbers are the same one can get from running gcov but it is more compact. For each line it is specified how often each of the basic blocks belonging to this line is executed. It can happen that more than one basic block belongs to one line and in this case the numbers are all listed, separated by commas.

2.5 Memory Profiling

Our test program uses extensively the functions to dynamically allocate memory. Therefore it is worthwhile introducing a tool to investigate the use of this function here. glibc 2.2 will come with such a tool which is named `memprof`. It's use is very easy and does not require and special preparation of the target program.

The functionality is contained in a shared object which gets preloaded to the program. Therefore it only works with programs which are dynamically linked with at least one object. The preloaded object requires a few parameters which are communicated through environment variables.

```
MEMPROF_PROG_NAME=a.out MEMPROF_OUTPUT=out LD_PRELOAD=memprof.so \
./a.out -d /usr/src/libc
```

The `MEMPROF_PROG_NAME` environment variable specifies the name of the program we want to profile. `MEMPROF_OUTPUT` specifies the name of the file where the profiling data is written into. `LD_PRELOAD` is an environment variable the dynamic loader recognize and which makes it load the named object before any of the dependencies of the application itself. It is therefore possible to intercept calls to library functions.

When the program terminates the module emits the first information. It consists of an overview of the total memory usage and a histogram to of the sizes of the various requests (in color!). The following is how this looks for a run of the example program.

```
Memory usage summary: heap total: 985282983, heap peak: 865008, stack peak: 33876
         total calls   total memory   failed calls
 malloc|      51235         787962             0
realloc|      27687      984486733             0   (in place: 27680, dec: 0)
 calloc|          2           8288             0
   free|        242          89705
Histogram for block sizes:
    0-15          18045  22% ===========================
   16-31          31921  40% =================================================
   32-47           1207   1% =
   48-63             79  <1%
   64-79             17  <1%
...
65440-65455           4  <1%
65456-65471           3  <1%
65472-65487           2  <1%
65488-65503           5  <1%
65504-65519           1  <1%
65520-65535          16  <1%
   large           2714   3% ====
```

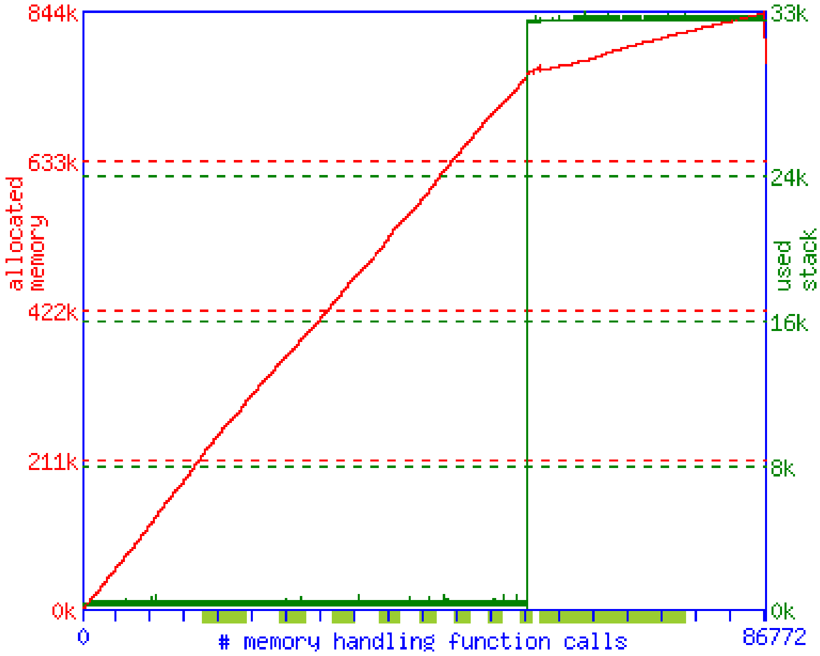

Figure 1: Memory use linear in number of function calls

To avoid wasting too much paper a lot of lines of the histogram are removed. The beginning of the contains basic information. The total size of allocations is 985282983 bytes (940 MB). But the peak memory use was only about 844 kB. This means that often memory got freed before new memory was allocated or that `realloc` was used extensively. The maximum stack use was about 33 kB. This information is sometimes interesting as, especially on IA32 machines, the stack size is several limited.

The next part provides numbers split up for each of the allocation functions. Here we see that we have a lot of `malloc` calls but the total amount of allocated memory is only 769 MB, which is an average of 15 bytes. The situation is different `realloc`. They contribute the major part of the allocated memory with roughly half the number of calls. Interesting in this line is that except for 7 calls all `realloc` calls returned memory in the same place (i.e., they expanded the current block). Also noteworthy is that no `realloc` calls reduces the allocated size. One thing one can derive from this is that the program uses `realloc` inefficiently since extending the block in place is only possible if the extensions are quite small.

The third part of the output is the histogram. The granularity of each is 16 bytes. From the numbers we saw in the second part we can guess that the allocations of sizes 0 to 47 almost exclusively were done using `malloc` since these are only 51173 calls while we computed an average of 15 bytes for

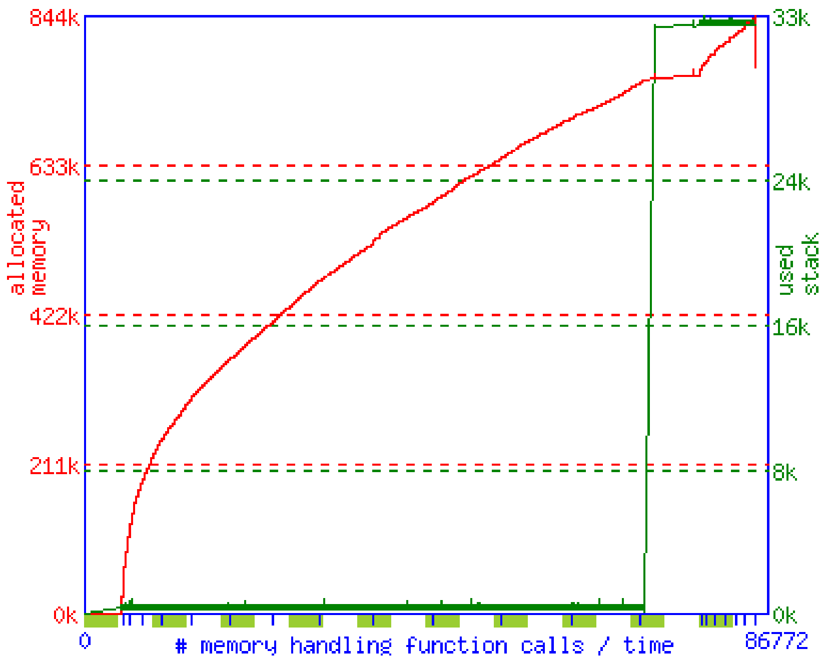

Figure 2: Memory use linear in time

each of the 51235 `malloc` calls. This leaves all the allocation requests with sizes from 64 bytes to `realloc`. Now one can really be sure that something strange happens in the `realloc` calls.

But this is not all the information we get from the memory profiling code. We still have the output file which was named `out` in the examples above. It contains a recording of all the memory allocation and deallocation operations the program performed. To analyze them a program named `memprofstat` must be used. It converts the data into graphics. There are two fundamental modes: the data can be displayed linear with the number of calls or it can be displayed linear with the ongoing time. Both views are useful.

Figure 1 shows the memory allocation linear with the number of function calls. We see a very slow and linear increase over time. The stack usage is minimal. Then suddenly the behavior changes. With knowledge of the program one can guess that this is the time the programs starts acting as a web server. The stack use increases dramatically but stays within a perfectly normal range. The heap usage increases even slower from this point on, but it does increase.

Looking at the same time at the numbers we discussed before we can guess that the allocations in the startup phase (i.e., before the stack usage went up) are all `malloc` calls and therefore the very slow increase afterwards is due to `realloc` calls. This perfectly well matches with what we have seen, namely, that we have a lot of `realloc` calls which all add only a bit and so despite many calls we only get a slow increase in used memory.

The second graphic in figure 2 shows the memory use linear over time. Again we can easily the two phases of the program. In the first phase we have the increase of the memory use up to about 740kB. But the line is not linear here. Instead the line the inverse of a parable. I.e., the time between the `malloc` calls increases quadratically which must be due to the algorithm used in this part of the program. We also see here that the startup phase takes quite long, relative to the rest of the program (where one directory and one large file were downloaded).

By now it should be clear why memory profiling a useful. Every non-trivial program uses dynamically allocated memory and this often in inner loops. Often this then allows the user to make educated guesses about the program itself. Optimizing the memory handling is most of the time an essential part of the overall optimization.

3 Starting to Work

The remainder of this consists of hints on how to proceed with optimizing the application. It is not necessary to do everything or in the order given. But the comments are partly assume that prior optimizations are already performed. It is assumed that the previous section about the tools is read and the reader can apply those tools now and interpret the output.

For each of the steps the reader is led to a solution in smaller steps. To get the most out of the tutorial please try to answer one question before going to the next since the answer is in many cases implied in the introduction to the next question.

Step 1: Algorithms used during startup

One of the things we already learned about the by looking at the output of `memprof` that some of the algorithms used in the startup functions have a $O(n^2)$ asymptotic behavior. The author of the program has already added an option to stop the program once the startup is finished, just to enable profiling.

$\Longrightarrow$ *Determine using the most adequate tool which functions in the program (not the libraries) are used at program start time.*

Once the functions are determine it is necessary to find out where the most time is spend. Many functions are unimportant when it comes to runtime since they do not contain any loops.

$\Longrightarrow$ *Which of the functions found in the last step contain (large) loops?*

$\Longrightarrow$ *What are the characteristics of the implementation of the algorithm in the inner most function?*

The algorithm used explains the quadratic behavior we previously observed. What is a better one? One possible answer is trees. The C library contains an implementation of a tree data structure.

⟹ *Replace the implementation of the single-linked list with a tree based data structure.*

Step 2: Memory Allocation

New measurements of the startup code performance should already show great improvements. But the `memprof` results showed another characteristic of the code: there are a lot of calls to `malloc`.

⟹ *Try finding a possibility to reduce the number* `malloc` *calls. We still want to allocate an individual memory block for each data set.*

Worth to remember: Whenever a data set includes one or more strings and those copies of the strings are not used anywhere else it is worthwhile allocating the memory required for the strings together with the `struct`. Basically it would look like this:

```
struct the_object
{
  ...
  const char str[0];
};

struct the_object *
allocate (const char *s, size_t len, <more parameters if needed>)
{
  struct the_object *newp;
  newp = (struct the_object *) malloc (sizeof (*sym) + len + 1);
  if (newp != NULL)
    {
      ...
      memcpy (newp->str, s, len + 1);
    }
  return newp;
}
```

Declaring the element `str` of the structure as it is done in the code above only works with `gcc`. With other compilers one has to make the string at least of size 1. If possible the `gcc` syntax should be used whenever `gcc` is used since it is a sign to the compiler which might add checks for boundary violations otherwise. The zero-sized strings *always* must be the last element in the structure.

When more than one string has to be stored one should allocate pointers for all but one of the strings. The memory allocated should be large enough for the structure and all the strings. Then one copies all the strings one after the other to the extra memory in the memory block and assigns pointers to the beginnings of the strings in this memory block to the character pointer elements in the structure. It should be obvious how to implement this, right?

Step 3: Copying Strings

When programming C handling strings is one of the things which has to be done for almost every program since the language does not provide a basic datatype for strings. The standard C library provides some functions to deal with strings but

- the standard functions are everything but ideal
- there are many better, but non-standard functions

⟹ *Rewrite the function* `gen_etags` *to handle strings better. Hint: take a look at the functions* `stpcpy` *and* `mempcpy`*, declared in* `<string.h>`*.*

When dealing with strings one almost always knows the size of the strings when copying. This is simply necessary to prevent buffer overruns which are the main reason for security leaks in programs. Many (most?) people simply ignore this extra information and explicitly or implicitly compute them again.

⟹ *Hopefully you rewrote* `gen_etags` *in a way where the length information is preserved and reused. If not, rewrite it again.*

To get a bit of inspiration this is how a function to concatenate tow strings in a newly allocated block of memory should look like:

```
char *
concat2 (const char *s1, const char *s2)
{
  size_t s1len = strlen (s1);
  size_t s2len = strlen (s2) + 1;
  char *buf = (char *) malloc (s1len + s1len);

  if (buf != NULL)
    (void) mempcpy (mempcpy (buf, s1, s1len), s2, s2len);

  return buf;
}
```

Step 4: Temporary Memory

Now that we have reworked the function `gen_etags` lets take another look. The body of the function consists mainly of code to construct a command line and then call `system`. The string is discarded afterwards. Now one can ask: why is the string allocated permanently in the first place? To this question there is no good answer.

⟹ *Rewrite* `gen_etags` *once more to allocate the memory for the command line* not *on the heap. Hint: read about* `alloca` *in the C library manual.*

Step 5: Function Scoping

A very minor point compared to the others and completely irrelevant in this context. But it is important when writing shared object code.

⟹ *List reasons why all functions if possible should be declared* `static`.

⟹ *Change all functions which possibly can to be defined in the local scope only.*

Step 6: Similar Problems

Now we are starting to look at the main part of the program where the web service is implemented. The functions `handle_request`, `get_request`, `title`, and especially `add_string` are still using `strcat`. `strcat` must die!!!

⟹ *Try replacing all calls of* `strcat`*. If necessary introduced extra variables to keep size information.*

Step 7: Understanding Short-Cut Evaluation

The expressions in `if` and `while` statements are something special. Normally the semicolon and braces are the only possibilities to create sequence points. But `&&` and `||` in conditional expressions are also sequence points.

Even more: the compiler is required to abort evaluating the expressions if

- the left side of a `&&` expression evaluated to zero. In this case the result is zero.
- the left side of a `||` expression evaluated to a nonzero value. In this case the result is one.

These rules should always be kept in mind since they allow to write much more efficient (and often shorter) programs then it would be without them. But it is also important to remember that they are always evaluated from left to right.

⟹ *With this knowledge in mind, rewrite* `print_file` *to be more efficient.*

Step 8: Minimize Number of System Calls

Basic functions like `read` or `write` are on Unix systems normally implemented as system calls. I.e., the functions are not really implemented in the system libraries (mainly the C library) but instead in the kernel. The C library might contain a function of this name but is nothing but a wrapper around the code which actually performs the system call.

Since the kernel of a Unix system is normally running on a different protection level than the user-level programs there are significant costs associated with the entry in the kernel to execute the system call. This is often two orders of magnitude slower than a normal system call. Optimizing here means avoiding system calls when necessary, especially when they occur in a loop.

⟹ *Can the functions* `start_reply` *and* `title` *be rewritten to use only one system call?*

Yes. For `title` the result would look like this:

```
void
title (int fd, const char *str, size_t len)
{
  struct iovec iov[3] =
  {
    { STR_AND_LEN ("<TITLE>") },
    { (void *) str, len },
    { STR_AND_LEN ("</TITLE>\n") }
  };
  writev (fd, iov, 3);
}
```

4 For High-Flyers

The example program tried to teach many of the aspects of optimization not everything could covered since otherwise the example problem would have been either too clumsy or too large. In this section we explain a few more optimization techniques which certainly can come handy in time.

4.1 `__builtin_constant_p`

Though the name `__builtin_constant_p` looks very much like the names of the intrinsic functions `gcc` knows about it is instead an operator similar to `sizeof`. Since it follows the good old LISP tradition to use the ending `_p`, one can see from the name that it is a predicate. It takes a single parameter and the return value is nonzero if the parameter value is constant at runtime.

This is a very useful thing to have. Many of the optimizations in the remainder of this text as well as many of the optimization implemented in the GNU C library headers depend on this feature. The `__builtin_constant_p` operator allows to avoid code bloat in situations where a macro definition only leads to favorable code if, due to compile-time computations, value propagation, and dead code elimination, the code size is reduced drastically. The following real-world example hopefully shows this.

```
#define strdup(s) \
  (__extension__                                          \
    (__builtin_constant_p (s) && __string2_1bptr_p (s)    \
     ? (((__const char *) (s))[0] == '\0'                 \
        ? (char *) calloc (1, 1)                          \
        : ({ size_t __len = strlen (s) + 1;               \
             char *__ret = (char *) malloc (__len);       \
             if (__ret != NULL)                           \
               __ret = (char *) memcpy (__ret, s, __len); \
             __ret; }))                                   \
     : strdup (s)))
```

The use of `__builtin_constant_p` prevents the use of the whole bunch of lines of this macro if the parameter is not constant. For the sake of it, one should once look at the code to see how much code would have to be generated if one would always use the replacement. We simply could not write such a macro without `__builtin_constant_p` guarding the expansion (the `__string2_1bptr_p` is not interesting here; interested parties should look at the `<bits/string2.h>` header of a `glibc` 2.1 installation).

For a constant parameter `s` (which must be a string constant) most of the expression can be computed at runtime. Since the compiler can see the first character of the string it knows whether the `calloc` call has to be made or whether the statement expression has to be executed. In the statement expression the result of the `strlen` call can be determined at compile-time.

One can see that in either case the piece of code which remains is very small. Due to the optimizations performed by the compiler many of the operations of the code do not have be executed at runtime. In the above case, for a non-empty string, one would in the end have two function calls and an `if` expression but it would not be necessary to compute the string length which might be a big advantage.

4.2 Type-generic Macros

When writing a macro to help speeding up certain operations it is sometimes the case that one wants the same functionality for different type. For simple operations this is easy, as one can simply let the compiler figure out how to use the arguments. It gets complicated as soon as one has to define variables inside the macro and if different functions depending on the used type have to be used.

Assume we want to write code which implements $ac + bc$ and we want to have it available for all types a, b, and c might have. The compiler can help figure out by itself which is the type of the result. Since the result of operations on two numbers is of the larger type of the two operands and because the C compiler is performing this job whenever an expression is constructed if subexpressions, numbers, or constants of different types, we can simply use the type of `a + b + c` as the type of the end result. Using a feature in `gcc` it is possible to define such a variable:

```
#define tgscale(result, a, b, c) \
  do {                                                  \
    __extension__ __typeof__ ((a) + (b) + (c)) c__ = (c);  \
    *(result) = (a) * c__ + (b) * c__;                  \
  } while (0)
```

This might be a bit confusing at first. But `__typeof__`(o) defines a type and it is the same type o has. Therefore the changed line as before defines a variable `c__` and assigns a value to it. What we gained through this change is that we now can call `tgscale` with arguments of type `int`, `long long int` and even `double` and we always get `__c` to be defined the best way with respect to the result.

There is another situation where `__typeof__` helps writing code even if not many variables are used and the common type has to be deduced. The ISO C9x standard introduces a new header `<tgmath.h>` allowing the user to write code using the mathematical functions without taking care of the different variants for the different types. E.g., one can call `sin` and depending on the type of the parameter the correct function from the six possibilities is picked. A naïve approach to this problem would be:

```
#define sin(Val) \
```

```
(sizeof (__real__ (Val)) > sizeof (double)      \
 ? (sizeof (__real__ (Val)) == sizeof (Val)     \
    ? sinl (Val) : csinl (Val))                 \
 : (sizeof (__real__ (Val)) == sizeof (double)  \
    ? (sizeof (__real__ (Val)) == sizeof (Val)  \
       ? sin (Val) : csin (Val))                \
    : (sizeof (__real__ (Val)) == sizeof (Val)  \
       ? sinf (Val) : csinf (Val))))
```

But this is not correct. Try to find out yourself before reading the footnote and looking at the correct code.[1] To correct the problem we have to introduce a variable and this is where `__typeof__` comes into play again.

```
#define sin(Val) \
  (__extension__                                          \
   ({ __typeof__ (Val) __tgmres;                          \
      if (sizeof (__real__ (Val)) > sizeof (double))      \
        {                                                 \
          if (sizeof (__real__ (Val)) == sizeof (Val))    \
            __tgmres = sinl (Val);                        \
          else                                            \
            __tgmres = csinl (Val);                       \
        }                                                 \
      else if (sizeof (__real__ (Val)) == sizeof (double)) \
        {                                                 \
          if (sizeof (__real__ (Val)) == sizeof (Val))    \
            __tgmres = sin (Val);                         \
          else                                            \
            __tgmres = csin (Val);                        \
        }                                                 \
      else                                                \
        {                                                 \
          if (sizeof (__real__ (Val)) == sizeof (Val))    \
            __tgmres = sinf (Val);                        \
          else                                            \
            __tgmres = csinf (Val);                       \
        }                                                 \
      __tgmres; }))
```

This example summarizes almost everything we discussed so far. If you would have been able to write this code yourself you have learned the lessons. Because the code now is not a single expression the compiler does not find the most general type. In fact, the assignment to the variable `__tgmres` might force a conversion to a narrower type. But this never happens for any

[1]The above is a single expression and it must have exactly one statically determined type. This type must be the most general one to be able to represent values of all the other types without loss. Therefore the returned value is always of type `complex long double` which certainly is not what we want.

reached code: only one if the six assignments is really executed and here the assignment does not loose any precision. All the other cases are dead code and will be eliminated by the optimizer.

4.3 Functions of No Return

Every large project contains at least one function which is used for fatal errors and which gracefully terminates the application. Such a function is often not treated in an optimal way since the compiler does not know that the function does not return. Take the following example function and its use in some code.

```
void fatal (...) __attribute__ ((__noreturn__));

void
fatal (...)
{
  ... /* Print error message. */ ...
  exit (1);
}

{
  ... /* read d */ ..
  if (d == 0)
    fatal (...);
  else
    a = b / d;
  ... /* and so on */ ...
}
```

The function `fatal` is guaranteed to never return since the function `exit` has the same guarantees. Therefore we annotated the prototype of the function with `__attribute__((__noreturn__))`. This `gcc` extension lets the author specify that: the function will never return.

Without this assurance the compiler would have to translate the `if` clause in the example to something which corresponds the following pseudo-code:

1 Compare `d` with zero
2 If not zero jump to 5
3 Call `fatal`
4 Jump to 6
5 Compute `b / d` and assign it to `a`
6 ...

This is not far from the optimum but there is unneeded code. The line 4 is never executed since the call to `fatal` does not return. If the compiler knows about this it will avoid this line and effectively transforms the source code to this:

```
{
  ... /* read d */ ..
  if (d == 0)
    fatal (...);
  a = b / d;
  ... /* and so on */ ...
}
```

Please note that the `else` is gone. This transformation would have been invalid without the knowledge about the behavior of `fatal`. Even if this is no big improvement and does not happen all that frequently, one should always think about marking a function this way. The compiler will emit warnings about unreachable code if one forgets about the behavior of the not-returning function and adds some extra code after the function call.

4.4 Constant Value Functions

Some functions only depend on the parameters passed into it and have no side effects (i.e., they don't change any global or static variable, produce no output, and expect no input). This property of *pure functions* is for the compiler not visible from the prototype and so it always has to assume the worst, namely, that the function has side effects which is something preventing certain optimizations.

As an example take the `htons` function which either returns its argument (if the machine is big-endian) or swaps the byte order (if it is a little-endian machine). There are no side effects and only the parameter is used to determine the result. `htons` clearly is a pure function.
The following code demonstrates the less than optimal case:

```
{
  short int serv = ... /* Somehow find this out */ ...;

  while (1)
    {
      struct sockaddr_in s_in;

      memset (&s_in, '\0', sizeof s_in);
      s_in.sin_port = htons (serv);

      ... /* lots of code where serv is not used */ ...
    }
}
```

This might be the outer loop of a network application which opens a socket connection to various hosts one after the other. The port to which the host connects always is the same since the variable `serv` does not change. Since we said that `htons` is a pure function this means that in every iteration of the loop the result of call to htons is the same. What we would like to see is an automatic transformation of the code to something like this:

```
{
  short int serv = ... /* Somehow find this out */ ...;
  serv = htons (serv); /* Once and for all compute the port */

  while (1)
    {
      struct sockaddr_in s_in;

      memset (&s_in, '\0', sizeof s_in);
      s_in.sin_port = serv;

      ... /* lots of code where serv is not used */ ...
    }
}
```

It is possible to achieve this by marking the htons function appropriately. gcc allows to give a function the attribute __const__ which tells the compiler that the function is pure. I.e., if we would have added

```
extern uint16_t htons (uint16_t __x) __attribute__ ((__const__));
```

before the code of the initial example we would have given the compiler the opportunity to generate the code we want. Marking pure functions using __const__ can mean quite an improvement and will in no case lead to worse code. Therefore one should always think about this optimization.

4.5 Different Calling Conventions

Each platform has specific calling conventions so for programs/libraries written by different people with possibly different compilers to work together. These calling conventions were defined when the platform was young and maybe even when the architecture/processor behaved differently.

Anyhow, there might be platforms and situations where one wants to use a different calling convention that the compiler supports because it is more efficient. This is not possible in general but nobody can forbid using this internally in a project. If only functions which are never called outside the project are defined with a different calling convention there is no problem with this.

Especially on the Intel IA32 platform, there are a few calling conventions supported by the compiler which are different from the standard Unix IA32 calling conventions and which can occasionally speed up the program significantly. Other platforms might allow similar changes. The GNU C compiler manual explains the details. For this section we will restrict the descriptions to the IA32 platform.

Changes to the calling conventions can be made in two ways: generally change the conventions by a command line option or individually change it using a function attribute. We will discuss only the latter since using a command line option is unsafe because exported functions might be effected or the command line option might be missing in another compiler run. Function attributes are always preferable.

4.5.1 `__stdcall__`

The first attribute changes the way the memory used to pass parameters is freed. Parameters are normally passed on the stack and at some point the stack pointer has to be adjusted to take this into account. The standard calling conventions on IA32 Unix is to let the caller correct the stack. This allows delaying the stack correction so that the effect of more than one function call can be corrected at once. On the other hand, if the `__stdcall__` attribute is defined for a function this signals that the function itself corrects the stack. This is not a bad idea on IA32 platforms since the architecture has a single instruction that allows returning from the function call and correcting the stack in one instruction. The effect can be seen in the following example.

```
int
__attribute__ ((__stdcall__))
add (int a, int b)
{
  return a + b;
}

int
foo (int a)
{
  return add (a, 42);
}

int
bar (void)
{
  return foo (100);
}
```

The translated the assembler output looks like this:

```
 8                     add:
 9 0000 8B442408              movl    8(%esp), %eax
10 0004 03442404              addl    4(%esp), %eax
11 0008 C20800                ret     $8
...
```

```
17                    foo:
18 0010 6A2A                   pushl   $42
19 0012 FF742408               pushl   8(%esp)
20 0016 E8E5FFFF               call    add
20      FF
21 001b C3                     ret
...
27                    bar:
28 0020 6A64                   pushl   $100
29 0022 E8E9FFFF               call    foo
29      FF
30 0027 83C404                 addl    $4, %esp
31 002a C3                     ret
```

Only the important lines are shown in the listing above. What has to be recognized is that the function `foo` does not have to do anything after the call of function `add`. The `ret` instruction in line 11 takes care of the memory necessary for the two parameters passed on the stack. The situation in function `bar` is different. Since `foo` is not marked with the `__stdcall__` attribute, it does not free the memory and this has to be done in the caller. Therefore we see a stack pointer manipulation in line 30.

From this short example it seems that using `__stdcall__` only has advantages. Even the code generated is smaller (the size of the `ret` instruction increased by two bytes, but the `addl` instruction has three bytes). But this is not so. Since the compiler is clever and corrects the stack pointer for several function calls that are done in a row only once the gain in code size is not that big anymore. In addition, future changes in the compiler will make handling the parameter allocation much faster and using `__stdcall__` will become counter-productive.[2] So, use `__stdcall__` with care.

4.5.2 `__regparm__`

A more interesting function attribute is `__regparm__`. It is only available for the IA32 platform since most other platforms do not have a standard calling convention which would make this necessary.

Using the `__regparm__` attribute, specify how many integer and pointer parameters (up to three) are passed in registers instead of on the stack. This can make a significant difference especially if the work performed in the called function is not much and the parameters have to be used immediately. The following not very realistic example code shows this dramatically.

```
int
__attribute__ ((__regparm__ (3)))
```

[2] One method already used on some platforms is to allocate the stack space needed for all functions calls once at function entry time. This is not possible with the `__stdcall__` calling convention.

```
add (int a, int b)
{
  return a + b;
}
```

Compiling this with optimizations leads to the following assembler output (only the code-producing lines are shown).

```
 8                    add:
 9 0000 01D0                  addl    %edx, %eax
10 0002 C3                    ret
```

This is the optimal code which could be generated for the functions. Using this way to pass parameters almost always has advantages. In the worst case the called function will store the variables itself somewhere on the stack if it needs the registers for some other computations. But in many cases, the values can be used directly from the registers.

4.6 Sibling Calls

One optimization performed automatically by the compiler for many platforms is sibling call optimization. If a function call is the last thing happening in a function, the code that usually gets generated looks like this:

	This is inside function F1
n	call function *F2*
$n+1$	execute code of *F2*
$n+2$	get return address from call in *F1*
$n+3$	jump back into function *F1*
$n+4$	optionally adjust stack pointer from call to *F2*
$n+5$	get return address from call to *F1*
$n+6$	jump back to caller of *F1*

This is not surprising but often not optimal. If no work has to be done in step $n+4$ or if this work can be moved before the call to *F2* the return from *F2* stops in *F1* only to jump again. It would be much better if the return from *F2* would immediately end up at the caller of *F1*.

To do this the subroutine call performed in *F1* must be changed into a jump which does not store a new return address. This way the return address normally used in *F1* would be used in *F2* and the result would be exactly as we want.

The compiler already performs this optimization occasionally and in the future will do it more often. It will recognize all situations in which the intermediate return step can be avoided. This is not possible on all machines at all times but will be quite frequent. The consequence for the programmer is that s/he should try to arrange the code so that function calls happen as the last thing in a function.

4.7 Using goto

Despite what Dijkstra says[3], using the `goto` command sometimes has advantages. One should, though, not use it without knowing the effect. Using `goto`s definitely disturbs the compiler to some extend since the various flow analysis mechanisms don't work as well in the presence of `goto`s.

Before using `goto`, one should look at the generated assembler code. It will also be necessary to understand the branch prediction mechanism of the processor. Equipped with this knowledge one can insert in strategic places `goto`s where normally the translation of a loop or conditional statement would lead to different code. `goto`s also might enable writing loops differently. With `goto` it is possible to leave a loop at an arbitrary point.

It is not possible to give general advice when to use `goto`. But if one decides to use it one should make sure to take measurements of the performance both with and without it. Normally the compilers do a good job and adding `goto`s might even hurt.

A The Source Code

```
/* Demonstration program for optimization tutorial.
   Copyright (C) 1999 Free Software Foundation, Inc.
   Written by Ulrich Drepper <drepper@cygnus.com>, November 1999.

   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., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.  */

#include <argp.h>
#include <ctype.h>
#include <dirent.h>
#include <errno.h>
#include <error.h>
```

[3] You know his famous article "`goto` *considered harmful*"?

```
#include <ftw.h>
#include <mcheck.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/param.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/uio.h>


/* Define the images we need.  */
#include "blank.h"
#include "folder.h"
#include "text.h"


/* Name of the tags file.  */
#define TAGS "TAGS"
/* Port to be used for incoming connections.  */
#define PORT 8080


/* Helper macro for lazy coders.  */
#define STR_AND_LEN(Str) Str, sizeof (Str) - 1



/* Exit value when the requested amount of memory is not available.
   The caller may set it to some other value.  */
int xmalloc_exit_failure = EXIT_FAILURE;

static void *
fixup_null_alloc (n)
     size_t n;
{
  void *p;

  p = 0;
  if (n == 0)
    p = malloc ((size_t) 1);
```

```
  if (p == 0)
    error (xmalloc_exit_failure, 0, "memory exhausted");
  return p;
}

/* Allocate N bytes of memory dynamically, with error checking.  */

void *
xmalloc (n)
     size_t n;
{
  void *p;

  p = malloc (n);
  if (p == 0)
    p = fixup_null_alloc (n);
  return p;
}

/* Allocate memory for N elements of S bytes, with error checking.  */

void *
xcalloc (n, s)
     size_t n, s;
{
  void *p;

  p = calloc (n, s);
  if (p == 0)
    p = fixup_null_alloc (n);
  return p;
}

/* Change the size of an allocated block of memory P to N bytes,
   with error checking.
   If P is NULL, run xmalloc.  */

void *
xrealloc (p, n)
     void *p;
     size_t n;
{
  if (p == 0)
    return xmalloc (n);
```

```
  p = realloc (p, n);
  if (p == 0)
    p = fixup_null_alloc (n);
  return p;
}



char *
xstrdup (string)
     char *string;
{
  return strcpy (xmalloc (strlen (string) + 1), string);
}


/* Data structure to store symbol information.  */
struct known_symbol
{
  const char *symbol;
  unsigned long int line;
  const char *file;
  struct known_symbol *next;
};

/* List of the known symbols.  */
struct known_symbol *known_symbols;

/* Search for a symbol.  */
static inline struct known_symbol *
symbol_search (const char *symbol)
{
  struct known_symbol *runp = known_symbols;

  while (runp != NULL && strcmp (symbol, runp->symbol) != 0)
    runp = runp->next;

  return runp;
}

/* Helper inline functions to handle symbol list.  */
static inline void
symbol_add (const char *symbol, unsigned long int line, const char *file)
{
```

```
  struct known_symbol *sym = symbol_search (symbol);

  if (sym == NULL)
    {
      sym = (struct known_symbol *) xmalloc (sizeof (*sym));
      sym->symbol = xstrdup (symbol);
      sym->line = line;
      sym->file = file;
      sym->next = known_symbols;

      known_symbols = sym;
    }
  else
    {
      /* A symbol of this name already exists.  We prefer definitions in
 .c files over those in .h files.  So test for this.  */
      if (strlen (file) > 2 && strcmp (file + strlen (file) - 2, ".c") == 0
  && (strlen (sym->file) <= 2
      || strcmp (sym->file + strlen (sym->file) - 2, ".c") != 0))
{
  /* Replace the symbol.  */
  sym->file = file;
  sym->line = line;
}
    }
}


/* Helper function to construct long strings.  */
static void
add_string (char **bufp, size_t *actp, size_t *lenp, const char *s, ...)
{
  va_list va;

  va_start (va, s);

  while (s != NULL)
    {
      if (*actp + strlen (s) >= *lenp)
{
  /* Resize the buffer.  */
#ifdef GOOD
  *lenp = 2 * MAX (*lenp, strlen (s));
#else
```

```
  *lenp = *actp + strlen (s) + 1;
#endif
  *bufp = (char *) xrealloc (*bufp, *lenp);
  (*bufp)[*actp] = '\0';
}

      strcat (*bufp, s);
      *actp += strlen (s);

      s = va_arg (va, const char *);
    }

  va_end (va);
}

static void
add_char (char **bufp, size_t *actp, size_t *lenp, char c)
{
  if (*actp + 1 >= *lenp)
    {
      /* Resize the buffer.  */
#ifdef GOOD
      *lenp = 2 * MAX (*lenp, 100);
#else
      *lenp += 1;
#endif
      *bufp = (char *) xrealloc (*bufp, *lenp);
    }

  (*bufp)[(*actp)++] = c;
  (*bufp)[*actp] = '\0';
}


/* Concatenate three strings.  */
static inline char *
concat3 (const char *s1, const char *s2, const char *s3)
{
  char *res = (char *) xmalloc (strlen (s1) + strlen (s2) + strlen (s3) + 1);
  strcpy (res, s1);
  strcat (res, s2);
  strcat (res, s3);
  return res;
}
```

```
/* HTTP protocol definitions.  */
#define MSG(nr) \
  msg[nr / 100 - 1][nr % 100].str, msg[nr / 100 - 1][nr % 100].len
struct standard_msg
{
  const char *str;
  size_t len;
  const char *body;
} msg[5][18] =
{
#define ENTRY(Nr, Str) \
    [Nr % 100] = { #Nr " " STR_AND_LEN (Str) + 4 }
  [0] = {
    ENTRY (100, "Continue"),
    ENTRY (101, "Switching Protocols")
  },
  [1] = {
    ENTRY (200, "OK"),
    ENTRY (201, "Created"),
    ENTRY (202, "Accepted"),
    ENTRY (203, "Non-Authoritative Information"),
    ENTRY (204, "No Content"),
    ENTRY (205, "Reset Content"),
    ENTRY (206, "Partial Content")
  },
  [2] = {
    ENTRY (300, "Multiple Choices"),
    ENTRY (301, "Moved Permanently"),
    ENTRY (302, "Found"),
    ENTRY (303, "See Other"),
    ENTRY (304, "Not Modified"),
    ENTRY (305, "Use Proxy"),
    ENTRY (307, "Temporary Redirect")
  },
  [3] = {
    ENTRY (400, "Bad Request"),
    ENTRY (401, "Unauthorized"),
    ENTRY (402, "Payment Required"),
    ENTRY (403, "Forbidden"),
    ENTRY (404, "Not Found"),
    ENTRY (405, "Method Not Allowed"),
    ENTRY (406, "Not Acceptable"),
```

```
    ENTRY (407, "Proxy Authentication Required"),
    ENTRY (408, "Request Timeout"),
    ENTRY (409, "Conflict"),
    ENTRY (410, "Gone"),
    ENTRY (411, "Length Required"),
    ENTRY (412, "Precondition Failed"),
    ENTRY (413, "Request Entity Too Large"),
    ENTRY (414, "Request-URI Too Long"),
    ENTRY (415, "Unsupported Media Type"),
    ENTRY (416, "Requested Range Not Satisfiable"),
    ENTRY (417, "Expectation Failed"),
  },
  [4] = {
    ENTRY (500, "Internal Server Error"),
    ENTRY (501, "Not Implemented"),
    ENTRY (502, "Bad Gateway"),
    ENTRY (503, "Service Unavailable"),
    ENTRY (504, "Gateway Timeout"),
    ENTRY (505, "HTTP Version Not Supported")
  }
};



/* Definitions of arguments for argp functions.  */
static const struct argp_option options[] =
{
  { NULL, 0, NULL, 0, "Startup Options:" },
  { "debug", 'd', NULL, 0, "Don't run in the background" },
  { "iterative", 'i', NULL, 0, "Don't handle multiple requests in parallel" },
  { "recreate-etags", 'e', NULL, 0, "Recreate " TAGS " file" },
  { "startup-only", 's', NULL, 0, "Run only startup code" },
  { NULL, 0, NULL, 0, NULL }
};


/* Short description of program.  */
static char doc[] = "Source Code Display Server.";

/* Strings for arguments in help texts.  */
static char args_doc[] = "ROOT";

/* Prototype for option handler.  */
extern error_t parse_opt (int key, char *arg, struct argp_state *state);
```

```
/* Data structure to communicate with argp functions.  */
static struct argp argp =
{
  options, parse_opt, args_doc, doc
};


/* If nonzero run in debug mode.  */
int do_debug = 0;

/* If nonzero process one requests at a time.  */
int do_iterative = 0;

/* If nonzero recreate etags file.  */
int do_etags = 0;

/* If nonzero terminate after startup code ran (for profiling).  */
int do_startup_only;

/* Device and node number of the root directory.  */
dev_t root_dev;
ino_t root_ino;

/* Prototypes for local functions.  */
extern void create_etags (const char *root, int recreate);
extern void read_etags (const char *root);
extern int create_socket (int port);
extern void server_loop (int sock);
extern void handle_request (int sock);
extern void get_request (int fd, char *url, char *buf);
extern void print_file (int fd, const char *fname);
extern void start_reply (int fd, int nr);
extern void title (int fd, const char *str, size_t len);
extern void start_body (int fd);
void finish_reply (int fd);


int
main (int argc, char *argv[])
{
  int remaining;
  const char *root;
  int sock;
```

```
  /* Enable memory allocation debugging.  */
  mcheck (0);

  /* Parse and process arguments.  */
  argp_parse (&argp, argc, argv, 0, &remaining, NULL);

  /* Get name of root directory.  */
  if (argc - remaining != 1 || argv[remaining][0] == '\0')
    argp_help (&argp, stdout, ARGP_HELP_SEE | ARGP_HELP_EXIT_ERR,
	       program_invocation_short_name);

  root = argv[remaining];

  /* Make sure the etags file is available.  */
  create_etags (root, do_etags);

  /* Read the TAGS file.  */
  read_etags (root);

  if (do_startup_only)
    exit (0);

  /* Go to the root directory.  */
  if (chdir (root) != 0)
    error (EXIT_FAILURE, errno, "cannot change to directory `%s'", root);

  /* If we are not debugging, daemonize now.  */
  if (! do_debug)
    daemon (1, 0);

  /* Create socket.  */
  sock = create_socket (PORT);

  /* The serverloop.  */
  server_loop (sock);

  /* When we return prepare to terminate.  */
  shutdown (sock, 2);
  close (sock);

  return 0;
}
```

```
/* Handle program arguments.  */
error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
  switch (key)
    {
    case 'd':
      do_debug = 1;
      break;
    case 'i':
      do_iterative = 1;
      break;
    case 'e':
      do_etags = 1;
      break;
    case 's':
      do_startup_only = 1;
      break;
    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}


int
gen_etags (const char *fname, const struct stat *st, int flag,
   struct FTW *info)
{
  int result = 0;

  if (S_ISDIR (st->st_mode)
      && strcmp (strrchr (fname, '/') ?: fname, "/CVS") != 0
      && strcmp (strrchr (fname, '/') ?: fname, "/RCS") != 0
      && strcmp (strrchr (fname, '/') ?: fname, "/SCCS") != 0)
    {
      /* Construct the command.  */
      char *cmd = (char *) xmalloc (strlen ("etags -a -o ")
    + (info->level == 0
       ? strlen (basename (fname)) + 1
       : (info->level - 1) * 3)
    + strlen (TAGS)
    + strlen (" ")
```

```
    + strlen (basename (fname))
    + strlen ("/*.[ch] 2>/dev/null") + 1);
      int i;

      strcpy (cmd, "etags -a -o ");
      if (info->level == 0)
{
  strcat (cmd, basename (fname));
  strcat (cmd, "/");
}
      else
for (i = 1; i < info->level; ++i)
  strcat (cmd, "../");
      strcat (cmd, TAGS);
      strcat (cmd, " ");
      strcat (cmd, basename (fname));
      strcat (cmd, "/*.[ch] 2>/dev/null");

      /* Now execute the command.  */
      result = system (cmd);

      /* Free the resources.  */
      free (cmd);
    }

  return result;
}


/* Make sure the etags file is available and recreate it if wanted.  */
void
create_etags (const char *root, int recreate)
{
  struct stat st;
  char *fname;

  fname = (char *) xmalloc (strlen (root)
    + (root[strlen (root) - 1] == '/' ? 0 : 1)
    + strlen (TAGS) + 1);

  /* Construct the file name.  */
  strcpy (fname, root);
  if (root[strlen (root) - 1] != '/')
    strcat (fname, "/");
```

```
  strcat (fname, TAGS);

  /* Test whether the file is available.  */
  errno = 0;
  if (stat (fname, &st) < 0 && errno == ENOENT)
    recreate = 1;
#if GOOD
  else if (errno == 0 && recreate)
    /* If the file exists and we have to recreate it, remove the old file.  */
    unlink (fname);
#endif

  /* Remember device and inode number.  */
  root_dev = st.st_dev;
  root_ino = st.st_ino;

  /* Now really do the work.  */
  if (recreate)
    {
#ifndef GOOD
      /* Remove the old file if it exists.  */
      if (access (fname, F_OK) == 0)
unlink (fname);
#endif

      /* XXX For now only use nftw.  Make worse when time permits.  */
      if (nftw (root, gen_etags, 10, FTW_CHDIR) < 0)
error (EXIT_FAILURE, errno, "cannot create " TAGS " file");
    }

  /* Free the resources.  */
  free (fname);
}


/* Read the tags file and create appropriate data structure to search
   in it.  */
void
read_etags (const char *root)
{
  char *fname;
  FILE *fp;
  char *buf = NULL;
  size_t len = 0;
```

```
  char *filename;

  fname = (char *) xmalloc (strlen (root)
    + (root[strlen (root) - 1] == '/' ? 0 : 1)
    + strlen (TAGS) + 1);

  /* Construct the file name.  */
  strcpy (fname, root);
  if (root[strlen (root) - 1] != '/')
    strcat (fname, "/");
  strcat (fname, TAGS);

  /* Open the file.  */
  fp = fopen (fname, "r");
  if (fp == NULL)
    error (EXIT_FAILURE, errno, "cannot open " TAGS " file");

  while (!feof (fp))
    {
      char *del;
      char *one;
      char *name;
      unsigned long int line;

      if (getline (&buf, &len, fp) < 0)
break;

      if (buf[0] == '\14')
{
  /* This means a new file is about to start.  Free the old
             name.  */
  char *endp;

  if (getline (&buf, &len, fp) < 0)
    break;

  endp = strchr (buf, ',');
  if (endp != NULL)
    {
      *endp = '\0';
      filename = xstrdup (buf);
    }

  continue;
```

```
}

      /* DEL is used as a marker to start either the name of the line
         number.  */
      del = strchr (buf, '\177');
      if (del == NULL)
continue;

      /* If there is now a \1 character the name is next.  Otherwise
         it is preceding the \177 character.  */
      one = strchr (del, '\1');
      if (one == NULL)
{
  /* The name is before the DEL character.  Search backwards.  */
  name = one = del;
  while (name > buf && ! isalnum (name[-1]) && name[-1] != '_')
    --name;

  /* This is the end of the name.  */
  *name = '\0';
  while (name > buf && (isalnum (name[-1]) || name[-1] == '_'))
    --name;

  if (*name == '\0')
    /* No name found.  */
    continue;
}
      else
{
  /* The name is between the DEL and the \1 character.  */
  name = one;
  while (name > buf && ! isalnum (name[-1]) && name[-1] != '_')
    --name;

  /* This is the end of the name.  */
  *name = '\0';

  name = del + 1;
  while (*name != '\0' && ! isalnum (*name) && *name != '_')
    ++name;
}

      line = strtoul (one + 1, NULL, 10);
```

```
      /* Add the symbol to the list.  */
      symbol_add (name, line, filename);
    }

  /* Close the file.  */
  fclose (fp);

  /* Free the resources.  */
  free (buf);
}


int
create_socket (int port)
{
  struct sockaddr_in sin;
  int sock;

  /* Prepare the socket description.  */
  memset (&sin, '\0', sizeof (sin));
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = htonl (INADDR_ANY);
  sin.sin_port = htons (port);

  /* Create the socket.  */
  sock = socket (PF_INET, SOCK_STREAM, 0);

  /* Bind the socket to the port.  */
  if (bind (sock, &sin, sizeof (sin)) < 0)
    error (EXIT_FAILURE, errno, "cannot bind socket");

  /* Start listening.  */
  listen (sock, SOMAXCONN);

  return sock;
}


void
server_loop (int sock)
{
  while (1)
    {
      struct sockaddr_in cli_addr;
```

```
      socklen_t cli_len;
      int cli_socket;

      /* Get the new client.  */
      cli_len = sizeof (struct sockaddr_in);
      cli_socket = accept (sock, &cli_addr, &cli_len);
      if (cli_socket < 0)
        continue;

      /* We are handling the connection synchronously.  */
      handle_request (cli_socket);

      /* Close client connection.  */
      close (cli_socket);
    }
}


void
handle_request (int sock)
{
  char buf[8192];
  ssize_t n;
  char *begin;
  char *end;

  n = read (sock, buf, sizeof buf);
  if (n <= 0)
    return;

  /* Decompose the command.  */
  for (begin = buf; isspace (*begin); ++begin)
    ;

  end = memchr (begin, '\r', n);
  if (end == NULL)
    /* Incomplete request.  */
    return;

  /* At the end of the buffer there should be the protocol
     specification.  */
  if (end - 8 <= begin || memcmp (end - 8, "HTTP/", 5) != 0 || end[-2] != '.')
    /* Invalid request.  */
    return;
```

```
  if (end[-3] != '1' || (end[-1] != '0' && end[-1] != '1'))
    {
      /* This version of the protocol is not supported.  */
      start_reply (sock, 505);

      title (sock, MSG (505));

      start_body (sock);

      write (sock, STR_AND_LEN ("<H1>Unsupported protocol</H1>"));

      finish_reply (sock);

      return;
    }

  /* Terminate the URI.  */
  end[-9] = '\0';

  if (strncmp (begin, "GET", 3) == 0)
    get_request (sock, begin + 4, end + 2);
  else
    {
      /* This command isn't implemented.  */
      char repl[8192];

      start_reply (sock, 400);
      title (sock, MSG (400));
      start_body (sock);

      strcpy (repl, "<H1>Not yet implemented</H1>");
      strcat (repl, "<P>Request was:\n");
      strcat (repl, buf);
      strcat (repl, "\n</P>\n");
      write (sock, repl, strlen (repl));

      finish_reply (sock);
    }
}


void
get_request (int fd, char *url, char *buf)
```

```
{
  struct stat st;

  if (strcmp (url, "") == 0 || strcmp (url, "/") == 0
      || strcmp (url, "index.html") == 0)
    url = "/.";

  /* Discard the leading slash.  All names are relative to the current
     directory.  */
  ++url;

  /* Special requests (images etc) here.  */
  if (strcmp (url, "quit") == 0)
    exit (0);
  else if (strcmp (url, "blank.gif") == 0)
    {
      dprintf (fd, "\
HTTP/1.1 200 OK\r\n\
Accept-Ranges: bytes\r\n\
Content-Length: %d\r\n\
Connection: close\r\n\
Content-Type: image/gif\r\n\r\n", sizeof (blank));
      write (fd, blank, sizeof (blank));
      return;
    }
  if (strcmp (url, "folder.gif") == 0)
    {
      dprintf (fd, "\
HTTP/1.1 200 OK\r\n\
Accept-Ranges: bytes\r\n\
Content-Length: %d\r\n\
Connection: close\r\n\
Content-Type: image/gif\r\n\r\n", sizeof (folder));
      write (fd, folder, sizeof (folder));
      return;
    }
  if (strcmp (url, "text.gif") == 0)
    {
      dprintf (fd, "\
HTTP/1.1 200 OK\r\n\
Accept-Ranges: bytes\r\n\
Content-Length: %d\r\n\
Connection: close\r\n\
Content-Type: image/gif\r\n\r\n", sizeof (text));
```

```
    write (fd, text, sizeof (text));
    return;
  }

/* Check whether the named object is a directory.  */
if (stat (url, &st) < 0)
  {
    /* If we reach this point the file was not found.  */
    char buf[8192];

    start_reply (fd, 404);
    title (fd, MSG (404));
    start_body (fd);

    strcpy (buf, "<H1>Not Found</H1>\nThe requested URL ");
    strcat (buf, url);
    strcat (buf, " was not found on this server.<P>\n");

    write (fd, buf, strlen (buf));

    finish_reply (fd);

    return;
  }

if (S_ISDIR (st.st_mode))
  {
    /* It's a directory.  Print a list of all the directory entries.  */
    DIR *dir = opendir (url);
    struct dirent *de;
    char *buf = NULL;
    size_t buflen = 0;
    size_t bufact = 0;
    char *cp;

    start_reply (fd, 200);

    cp = xmalloc (strlen ("Content of ") + strlen (url) + 1);
    strcpy (cp, "Content of ");
    strcat (cp, url);
    title (fd, cp, strlen (cp));
    free (cp);

    start_body (fd);
```

```
      add_string (&buf, &bufact, &buflen, "\
<H1>Content of ", url, "</H1>\n\
<PRE><IMG SRC=\"/blank.gif\" ALT=\"     \"> Name\n<HR>\n", NULL);

      while ((de = readdir (dir)) != NULL)
{
  int type;
#ifdef GOOD
  if (de->d_type != DT_UNKNOWN)
    type = DTTOIF (de->d_type);
  else
#endif
    {
      struct stat st;
      int res;

      cp = concat3 (url, "/", de->d_name);
      res = stat (cp, &st);
      free (cp);
      if (res < 0)
continue;

      type = st.st_mode;
    }

  if (S_ISDIR (type))
    {
      if ((strcmp (de->d_name, "..") == 0 && strcmp (url, ".") == 0)
  || strcmp (de->d_name, ".") == 0)
continue;

      add_string (&buf, &bufact, &buflen, "\
<IMG SRC=\"/folder.gif\" ALT=\"[DIR]\"> <A HREF=\"/", url, "/",
  de->d_name, "\">", de->d_name, "</A>\n", NULL);
    }
  else
    {
      /* If it is no .c or .h file we ignore it.  */
      if (strlen (de->d_name) <= 2
  || (strcmp (de->d_name + strlen (de->d_name) - 2, ".c")
      && strcmp (de->d_name + strlen (de->d_name) - 2, ".h")))
continue;
```

```
      add_string (&buf, &bufact, &buflen, "\
<IMG SRC=\"/text.gif\" ALT=\"[TXT]\"> <A HREF=\"/", url, "/",
  de->d_name, "\">", de->d_name, "</A>\n", NULL);
    }
}
      add_string (&buf, &bufact, &buflen, "</PRE>", NULL);

      write (fd, buf, bufact);
      free (buf);

      finish_reply (fd);
    }
  else if (S_ISREG (st.st_mode))
    {
      /* Normal file.  We handle that.  */
      print_file (fd, url);
    }
  else
    {
      /* Ick, what's that?  */
      char repl[8192];

      start_reply (fd, 405);
      title (fd, MSG (405));
      start_body (fd);

      strcpy (repl, "<H1>Cannot handle this file type</H1>\n<P>");
      if (S_ISCHR (st.st_mode))
strcat (repl, "Character device \"");
      else if (S_ISBLK (st.st_mode))
strcat (repl, "Block device \"");
      else if (S_ISFIFO (st.st_mode))
strcat (repl, "FIFO \"");
      else if (S_ISLNK (st.st_mode))
strcat (repl, "Symbolic link \"");
      else
strcat (repl, "File with unknown type \"");
      strcat (repl, url);
      strcat (repl, "\" cannot be handled.\n</P>\n");
      write (fd, repl, strlen (repl));

      finish_reply (fd);
    }
}
```

```
void
print_file (int fd, const char *fname)
{
  char *buf = NULL;
  size_t bufact = 0;
  size_t bufmax = 0;
  FILE *fp;

  start_reply (fd, 200);
  title (fd, fname, strlen (fname));
  start_body (fd);

  add_string (&buf, &bufact, &bufmax, "<PRE>", NULL);

  fp = fopen (fname, "r");
  if (fp != NULL)
    {
      int linenr = 0;
      char *line = NULL;
      size_t linelen = 0;
      enum
      {
normal,
comment,
preprocessor
      } state = normal;

      while (!feof (fp))
{
  ssize_t n = getline (&line, &linelen, fp);
  char linenrstr[18];
  char *cp;

  if (n < 0)
    break;

  if (n == 0)
    continue;

  sprintf (linenrstr, "<SUB>%4d</SUB>  ", ++linenr);
  add_string (&buf, &bufact, &bufmax, linenrstr, NULL);
```

```
  cp = line;

  while (*cp == ' ' || *cp == '\t')
    {
      add_char (&buf, &bufact, &bufmax, *cp);
      ++cp;
    }

  if (state == preprocessor || *cp == '#')
    {
      int was_hash = 0;

      if (state != preprocessor && *cp == '#')
{
  add_string (&buf, &bufact, &bufmax,
      "<FONT color=\"#ff6347\">#</FONT>", NULL);
  ++cp;
  while (*cp == ' ' || *cp == '\t')
    add_char (&buf, &bufact, &bufmax, *cp++);
  was_hash = 1;
}

      do
{
  char *startp = cp;

  while (islower (*cp))
    ++cp;

  if (startp < cp
      && ((strncmp (startp, "defined", cp - startp) == 0
   && cp - startp == 7)
  || (was_hash
      && ((strncmp (startp, "if", cp - startp) == 0 && cp - startp == 2)
  || (strncmp (startp, "define", cp - startp) == 0 && cp - startp == 6)
  || (strncmp (startp, "ifdef", cp - startp) == 0 && cp - startp == 5)
  || (strncmp (startp, "ifndef", cp - startp) == 0 && cp - startp == 6)
  || (strncmp (startp, "else", cp - startp) == 0 && cp - startp == 4)
  || (strncmp (startp, "elif", cp - startp) == 0 && cp - startp == 4)
  || (strncmp (startp, "endif", cp - startp) == 0 && cp - startp == 5)
  || (strncmp (startp, "undef", cp - startp) == 0 && cp - startp == 5)
  || (strncmp (startp, "pragma", cp - startp) == 0 && cp - startp == 6)
  || (strncmp (startp, "include", cp - startp) == 0 && cp - startp == 7)))))
    {
```

```
      add_string (&buf, &bufact, &bufmax,
  "<FONT color=\"#ff6347\">", NULL);
      while (startp < cp)
add_char (&buf, &bufact, &bufmax, *startp++);
      add_string (&buf, &bufact, &bufmax, "</FONT>", NULL);
    }
  else if (cp > startp)
    while (startp < cp)
      add_char (&buf, &bufact, &bufmax, *startp++);
  else if (*cp == '<')
    {
      add_string (&buf, &bufact, &bufmax, "&lt;", NULL);
      ++cp;
    }
  else if (*cp == '>')
    {
      add_string (&buf, &bufact, &bufmax, "&gt;", NULL);
      ++cp;
    }
  else
    add_char (&buf, &bufact, &bufmax, *cp++);

  was_hash = 0;
}
      while (*cp != '\n');

      state = cp[-1] == '\\' ? preprocessor : normal;
    }
  else if (state != normal)
    {
      add_string (&buf, &bufact, &bufmax, "<FONT color=\"#",
  state == comment
  ? "2e8b57" : "000000",
  "\">", NULL);
    }

  while (*cp != '\n')
    {
      switch (state)
{
case comment:
  if (cp[0] == '*' && cp[1] == '/')
    {
      add_string (&buf, &bufact, &bufmax, "*/</FONT>", NULL);
```

```
      cp += 2;
      state = normal;
      continue;
    }
  break;

default:
  if (cp[0] == '/' && cp[1] == '*')
    {
      add_string (&buf, &bufact, &bufmax,
  "<FONT color=\"#2e8b57\">", NULL);
      state = comment;
    }

  if (isalpha (*cp) || *cp == '_')
    {
      /* Find end of identifier.  */
      char *startp = cp;

      while (isalnum (*cp) || *cp == '_')
++cp;

      if (cp > startp
  && ((strncmp ("break", startp, cp - startp) == 0
       && cp - startp == 5)
      || (strncmp ("case", startp, cp - startp) == 0
  && cp - startp == 4)
      || (strncmp ("char", startp, cp - startp) == 0
  && cp - startp == 4)
      || (strncmp ("const", startp, cp - startp) == 0
  && cp - startp == 5)
      || (strncmp ("continue", startp, cp - startp) == 0
  && cp - startp == 8)
      || (strncmp ("default", startp, cp - startp) == 0
  && cp - startp == 7)
      || (strncmp ("do", startp, cp - startp) == 0
  && cp - startp == 2)
      || (strncmp ("double", startp, cp - startp) == 0
  && cp - startp == 6)
      || (strncmp ("extern", startp, cp - startp) == 0
  && cp - startp == 6)
      || (strncmp ("enum", startp, cp - startp) == 0
  && cp - startp == 4)
      || (strncmp ("float", startp, cp - startp) == 0
```

```
  && cp - startp == 5)
      || (strncmp ("if", startp, cp - startp) == 0
  && cp - startp == 2)
      || (strncmp ("else", startp, cp - startp) == 0
  && cp - startp == 4)
      || (strncmp ("goto", startp, cp - startp) == 0
  && cp - startp == 4)
      || (strncmp ("int", startp, cp - startp) == 0
  && cp - startp == 3)
      || (strncmp ("long", startp, cp - startp) == 0
  && cp - startp == 4)
      || (strncmp ("return", startp, cp - startp) == 0
  && cp - startp == 6)
      || (strncmp ("register", startp, cp - startp) == 0
  && cp - startp == 8)
      || (strncmp ("short", startp, cp - startp) == 0
  && cp - startp == 5)
      || (strncmp ("static", startp, cp - startp) == 0
  && cp - startp == 6)
      || (strncmp ("struct", startp, cp - startp) == 0
  && cp - startp == 6)
      || (strncmp ("switch", startp, cp - startp) == 0
  && cp - startp == 6)
      || (strncmp ("typedef", startp, cp - startp) == 0
  && cp - startp == 7)
      || (strncmp ("union", startp, cp - startp) == 0
  && cp - startp == 5)
      || (strncmp ("unsigned", startp, cp - startp) == 0
  && cp - startp == 8)
      || (strncmp ("void", startp, cp - startp) == 0
  && cp - startp == 4)
      || (strncmp ("while", startp, cp - startp) == 0
  && cp - startp == 5)))
{
  add_string (&buf, &bufact, &bufmax, "<b>", NULL);
  while (startp < cp)
    add_char (&buf, &bufact, &bufmax, *startp++);
  add_string (&buf, &bufact, &bufmax, "</b>", NULL);

  continue;
}
      else if (cp > startp)
{
  /* We have to see whether this is an identifier
```

```
     defined somewhere else.  */
  char *id = (char *) xmalloc (cp - startp + 1);
  struct known_symbol *sym;

  strncpy (id, startp, cp - startp + 1);
  id[cp - startp] = '\0';

  sym = symbol_search (id);
  if (sym != NULL && strcmp (fname, sym->file) != 0)
    {
      add_string (&buf, &bufact, &bufmax,
  "<A href=\"/", sym->file, "\">",
  id, "</A>", NULL);
    }
  else
    add_string (&buf, &bufact, &bufmax, id, NULL);
  continue;
}
    }

  break;
}

      if (*cp == '<')
add_string (&buf, &bufact, &bufmax, "&lt;", NULL);
      else if (*cp == '>')
add_string (&buf, &bufact, &bufmax, "&gt;", NULL);
      else
add_char (&buf, &bufact, &bufmax, *cp);

      ++cp;
    }

  if (state != normal)
    add_string (&buf, &bufact, &bufmax, "</FONT>", NULL);

  add_char (&buf, &bufact, &bufmax, '\n');
}

      fclose (fp);
    }

  add_string (&buf, &bufact, &bufmax, "</PRE>", NULL);
```

```
  write (fd, buf, strlen (buf));
  free (buf);

  finish_reply (fd);
}


void
start_reply (int fd, int nr)
{
  /* The usual SGML header for the HTML DTD must come first.  */
  write (fd, STR_AND_LEN ("HTTP/1.1 "));
  write (fd, MSG (nr));
  write (fd, STR_AND_LEN ("\n\
Connection: close\n\
Content-Type: text/html\n\n\
<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 TRANSITIONAL//EN\">\n\
<HTML><HEAD>\n\
<META http-equiv=\"Content-Type\" content=\"text/html; charset=iso-8859-1\">\n"));
}


void
title (int fd, const char *str, size_t len)
{
  char *buf = (char *) xmalloc (strlen ("<TITLE>")
+ len
+ strlen ("</TITLE>") + 1);

  strcpy (buf, "<TITLE>");
  strcat (buf, str);
  strcat (buf, "</TITLE>");

  write (fd, buf, strlen (buf));

  free (buf);
}


void
start_body (int fd)
{
  write (fd, STR_AND_LEN ("</HEAD><BODY>\n"));
}
```

```
void
finish_reply (int fd)
{
  write (fd, STR_AND_LEN ("</BODY></HTML>\n"));
}
```