Linux technical study
Minimum C program

by Eric Nicolas, enicolas@dvdfr.com
22 Jan 2005

Environement used:

Linux 2.6 (should work on any Linux 2.x)
Gcc 3.3 (any other version should do)
Intel x86 processor (required)

The other day I was wondering: how does a program start? (I mean from the kernel being given the instruction to start a process to the main() C entry point) How does a C program interacts with the kernel? This small technical study brings some answers to those two questions by showing a minimum C program. This self-contained work will not rely on the C library or any other object file or library in the system. We will have, for once, the full source code of our program!

When Linux starts a process it loads its source code into memory and jumps to the `_start` entry point. The stack is then filled with usual startup information: the command line (int argc, char **argv) and the environement variables (int envc, char **envp), layed out as follow:

```
int   argc
char* argv[0]
char* argv[1]
...
char* argv[argc - 1]
null pointer
char* envp[0]
...
char* envp[envc - 1]
null pointer
```

So our first task is to implement a main entry point which will transform this stack structure into the more usual C convention (int argc, char **argv, char **envp). Then we let the user-code runs, and in the end we must exit the process. For now let's assume this C function can be used:

```
void __exit(int retcode);
```

The adaptation code from the kernel environement stack layout to the C standard main is simple:

__main.c

```
void __main(int argc, char *stack)
{
  char **argv = &stack;
  char **envp = argv + argc + 1;

  int retcode = main(argc, argv, envp);

  __exit(retcode);
}
```

However, we cannot let the kernel directly jump to this C function because the compiler will assume this function has been 'called' with the return address first on the stack whereas the kernel would 'jump' right to it without pushing this additional address. So we need a tiny assembly boot code which will properly call into this C function.

__boot.s

```
.text
  .global _start

_start:
  call __main
  ret
```

Now is time to implement the first system call: the `__exit` function prototyped above. All system calls are entered via the 0x80 interruption. Parameters and system call number are provided via registers:

```
syscall id  EAX
1st param   EBX
2nd param   ECX
3rd param   EDX
```

As the C convention pushes the arguments onto the stack, our `__exit` implementation will simply obtain the return code value from the stack and set the appropriate registers before the syscall.

__exit.s

```
.text
.global __exit

__exit:
  movl  4(%esp), %ebx    # "exit" code
  movl  $1,      %eax    # exit() is syscall #1
  int   $0x80
  ret
```

In order to allow a program not totally dull, we also need two small C library functions:

```
void write(int fileno, const void *buffer, unsigned int length);
int strlen(const char *string);
```

The write function is available as a system call, so a tiny assembly implementation just translates C calling convention to the syscall registry parameters.

write.s

```
.text
.global write

write:
  movl  (12)(%esp), %edx     # length
  movl  (8 )(%esp), %ecx     # buffer
  movl  (4 )(%esp), %ebx     # fileno
  movl  $4,         %eax     # write() is syscall #4
  int   $0x80
  ret
```

`strlen` can be implemented very simply in C. That will do for this study (even if this can be a lot faster when implemented in assembly language).

strlen.c

```
int strlen(const char *string)
{
  int length = 0;
  while(*string) { string++; length++; }
  return length;
}
```

And finally we have everything required to write a small test program. It displays all the command line arguments and all the environment variables to the standard output file. It also returns the number of arguments on the command line.

test.c

```
void println(char *string)
{
  write(1, string, strlen(string));
  write(1, "\n", 1);
}

int main(int argc, char **argv, char **envp)
{
  int i;

  println("-- Command line --");
  for(i = 0; i < argc; ++i) println(argv[i]);

  println("-- Environment --");
  while(*envp)
  {
    println(*envp);
    envp++;
  }

  return argc;
}
```

It's now time to compile everything. Remember that we won't use any other source than those described here. So we first compile the assembly parts:

```
as __boot.s -o __boot.o
as __exit.s -o __exit.o
as write.s -o write.o
```

Then the C parts using `gcc`:

```
gcc -g -c __main.c -o __main.o
gcc -g -c strlen.c -o strlen.o
gcc -g -c test.c -o test.o
```

Finally we link everything together without using any other library, via `ld`:

```
ld test.o strlen.o write.o __boot.o __main.o __exit.o -o test
strip test
```

We managed to create a working program without using any libraries or pre-existing object files. The produced binary executable is only 948 bytes on my Linux box.