Creating Main and Linking C Sources

In normal C programming practice, the function main() is your normal program entry point. In order to try to keep your normal programming practices and familiarize yourself with kernel development, this tutorial will keep the main() function the entry point for your C code. As you remember in the previous section of this tutorial, we tried to keep minimal assembler code. In later sections, we will have to go back into 'start.asm' in order to add Interrupt Service Routines to call C functions.

In this section of the tutorial, we will attempt to create a 'main.c' as well as a header file to include some common function prototypes: 'system.h'. 'main.c' will also contain the function main() which will serve as your C entry point. As a rule in kernel development, we should not normally return from main(). Many Operating Systems get main to initialize the kernel and subsystems, load the shell application, and then finally main() will sit in an idle loop. The idle loop is used in a multitasking system when there are no other tasks that need to be run. Here is an example 'main.c' with the basic main, as well as the function bodies for functions that we will need in the next part of the tutorial.

```
#include < system.h >

/* You will need to code these up yourself!  */
unsigned char *memcpy(unsigned char *dest, const unsigned char *src, int count)
{
    /* Add code here to copy 'count' bytes of data from 'src' to
    *  'dest', finally return 'dest' */
}

unsigned char *memset(unsigned char *dest, unsigned char val, int count)
{
    /* Add code here to set 'count' bytes in 'dest' to 'val'.
    *  Again, return 'dest' */
}

unsigned short *memsetw(unsigned short *dest, unsigned short val, int count)
{
    /* Same as above, but this time, we're working with a 16-bit
    *  'val' and dest pointer. Your code can be an exact copy of
    *  the above, provided that your local variables if any, are
    *  unsigned short */
}

int strlen(const char *str)
{
    /* This loops through character array 'str', returning how
    *  many characters it needs to check before it finds a 0.
    *  In simple words, it returns the length in bytes of a string */
}

/* We will use this later on for reading from the I/O ports to get data
*  from devices such as the keyboard. We are using what is called
*  'inline assembly' in these routines to actually do the work */
unsigned char inportb (unsigned short _port)
{
    unsigned char rv;
    __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
    return rv;
}

/* We will use this to write to I/O ports to send bytes to devices. This
*  will be used in the next tutorial for changing the textmode cursor
*  position. Again, we use some inline assembly for the stuff that simply
*  cannot be done in C */
```

```
void outportb (unsigned short _port, unsigned char _data)
{
    __asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
}

/* This is a very simple main() function. All it does is sit in an
*  infinite loop. This will be like our 'idle' loop */
void main()
{
    /* You would add commands after here */

    /* ...and leave this loop in. There is an endless loop in
    *  'start.asm' also, if you accidentally delete this next line */
    for (;;);
}
```

'main.c': Our kernel's small, yet important beginnings

Before compiling this, we need to add 2 lines into 'start.asm'. We need to let NASM know that main() is in an 'external' file and we need to call main() from the assembly file, also. Open 'start.asm', and look for the line that says 'stublet:'. Immediately after that line, add the lines:

```
    extern _main
    call _main
```

Now wait just a minute. Why are there leading underscores for '_main', when in C, we declared it as 'main'? The compiler gcc will put an underscore in front of all of the function and variable names when it compiles. Therefore, to reference a function or variable from our assembly code, we must add an underscore to the function name if the function is in a C source file!.

This is actually good enough to compile 'as is', however we are still missing our 'system.h'. Simply create a blank text file named 'system.h'. Add all the function prototypes for memcpy, memset, memsetw, strlen, inportb, and outportb to this file. It is wise to use macros to prevent an include file, or 'header' file from declaring things more than once using some nice #ifndef, #define, and #endif tricks. We will include this file in each C source file in this tutorial. This will define each function that you can use in your kernel. Feel free to expand upon this library with anything you think you will need. Observe:

```
#ifndef __SYSTEM_H
#define __SYSTEM_H

/* MAIN.C */
extern unsigned char *memcpy(unsigned char *dest, const unsigned char *src, int count);
extern unsigned char *memset(unsigned char *dest, unsigned char val, int count);
extern unsigned short *memsetw(unsigned short *dest, unsigned short val, int count);
extern int strlen(const char *str);
extern unsigned char inportb (unsigned short _port);
extern void outportb (unsigned short _port, unsigned char _data);

#endif
```

Our global include file: 'system.h'

Next, we need to find out how to compile this. Open your 'build.bat' from the previous section in this tutorial, and add the following line to compile your 'main.c'. Please note that this assumes that 'system.h' is in an 'include' directory in your kernel sources directory. This command executes the compiler 'gcc'. Among the various arguments passed in, there is '-

Wall' which gives you warnings about your code. '-nostdinc' along with '-fno-builtin' means that we aren't using standard C library functions. '-I./include' tells the compiler that our headers are in the 'include' directory inside the current. '-c' tells gcc to compile only: No linking yet! Remembering from the previous section in this tutorial, '-o main.o' is the output file that the compiler is to make, with the last argument, 'main.c'. In short, compile 'main.c' into 'main.o' with options best for kernels.

TIP Right click the batch file and select 'edit' to edit it!

```
gcc -Wall -O -fstrength-reduce -fomit-frame-pointer -finline-functions -nostdinc -fno-builtin -I./include -c -o main.o main.c
```

Add this line to 'build.bat'

Don't forget to follow the instructions we left in 'build.bat'! You need to add 'main.o' to the list of object files that need to be linked to create your kernel! Finally, if you are stuck creating our accessory functions like memcpy, a solution 'main.c' is shown here.