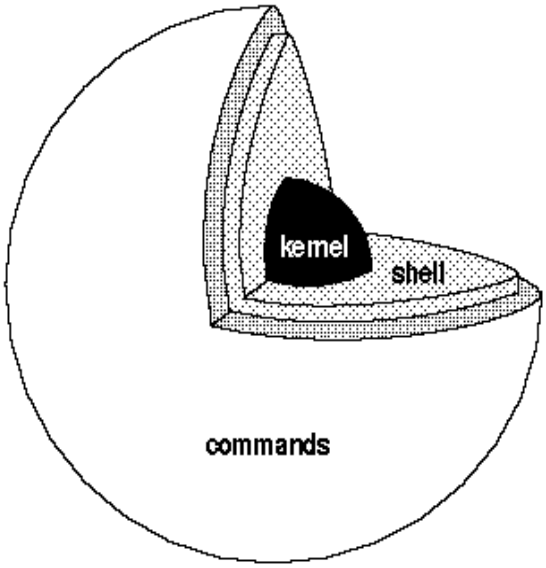
Beginning OS Development

First consider that there are several types of kernels you could develop:

A Monolithic kernel – The core os functions are integrated & it can load additional modules at runtime.

A Microkernel – Only minimal os functionality is in the kermel (mem/process management) other things run in userspace.

A Hybrid kernel – Essentially “microkernel with extentions”, similar to a monolithic kernel except that it does not load modules on its own.



Ofcourse there are more, but the good news is you don't have to decide what type you are developing yet, you just want something that boots.

What you need:
-Knowledge of C & Basic ASM
-NASM Assembler & GCC
-A Text Editor
-Quemu for Testing (Unless you like using floppies)
-Grub Files
(Only the files **stage1** and **stage2** from the grub project website**)**
-LD Linker

First the Kernel Entry:

This is called by the bootloader, it intializes basic settings and is the entry into your kernel.
It is almost always written in assembly since some things just can't be done in C.

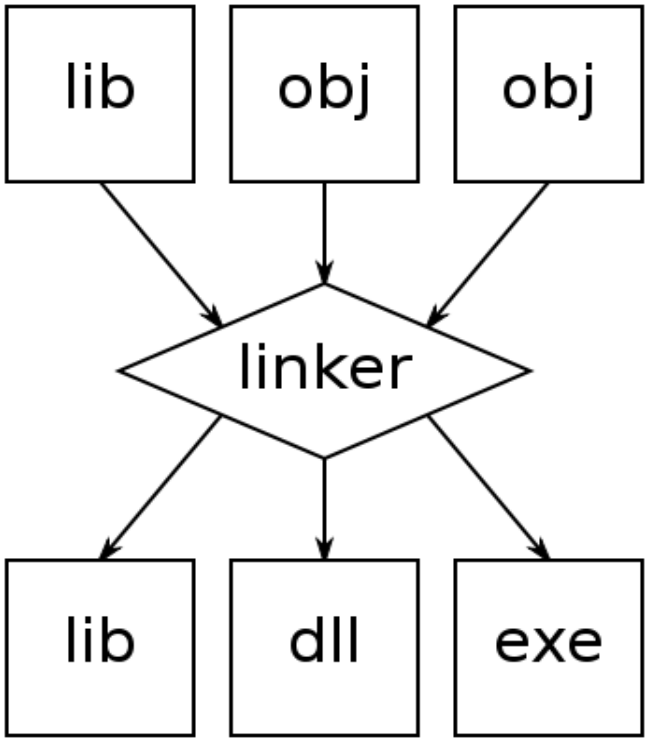
```
[BITS 32]
global start
start:
	mov esp, _sys_stack;
	jmp stublet

ALIGN 4
mboot:
   MULTIBOOT_PAGE_ALIGN    equ 1<<0
   MULTIBOOT_MEMORY_INFO        equ 1<<1
   MULTIBOOT_AOUT_KLUDGE        equ 1<<16
   MULTIBOOT_HEADER_MAGIC       equ 0x1BADB002
   MULTIBOOT_HEADER_FLAGS       equ MULTIBOOT_PAGE_ALIGN |
MULTIBOOT_MEMORY_INFO | MULTIBOOT_AOUT_KLUDGE
   MULTIBOOT_CHECKSUM     equ -(MULTIBOOT_HEADER_MAGIC +
MULTIBOOT_HEADER_FLAGS)
   EXTERN code, bss, end

   ; GRUB Multiboot header (Boot Sig)
   dd MULTIBOOT_HEADER_MAGIC
   dd MULTIBOOT_HEADER_FLAGS
   dd MULTIBOOT_CHECKSUM

   ; For linker script
   dd mboot
   dd code
   dd bss
   dd end
   dd start

stublet:
	extern main <--- Refers to our main C file which we will code later
	call main <--- You don't need these two lines (yet)
	jmp $

SECTION .bss
	resb 8192
_sys_stack:
```

Loads up a new 8 Kylobyte Stack and Jumps into an infinite loop. Has a unique signature recognized by grub (that is the bootloader you are going to be using).

Next The Linker:

The Linker is the application that takes the compiled C and ASM out files and combines them into one binary file that will be your kernel. It is typically ELF format.

```
OUTPUT_FORMAT(elf32-i386)  <--- The type of image you want to produce
ENTRY(start)
phys = 0x00100000;  <--- Pointer to 1MB where we want to load and run our binary
SECTIONS
{
	.text phys : AT(phys) {
	code = .;
	*(.text)
	*(.rodata)
	. = ALIGN(4096); <--- Ensures that each section starts at a sepperate page in memory
	}
	.data : AT(phys + (data - code)) {
	data = .;
	*(.data)
	. = ALIGN(4096);
	}
	.bss : AT(phys + (bss - code)) {
	bss = .;
	*(.bss)
	. = ALIGN(4096);
	}
	end = .;
}
```



At this point you can build your 'basic' system, it wont do anything yet ...but atleast it will compile.

```
nasm -f aout -o start.o start.asm

ld -T link.ld -o kernel.bin start.o
```

You may want to create a makefile to streamline this process as it gets more complicated later.
Next your Main file:

This is your C entry point, you will have all further code branching out of this. Since this is your new operating system you will have to write your own standard libraries.

General rule: If you #include it and you haven't coded it, you are doing it wrong!

This main never returns, instead it will end up in a infinite loop.

```
void main()
{
	for(;;);
}
```

The first library you want to start working on is your system.h
-A Basic system.h should have following, having these functions will make your life easier.

```
#ifndef __SYSTEM_H
#define __SYSTEM_H

/* These could be put in your “main” file */

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);

extern void cls();
extern void putch(unsigned char c);
extern void puts(unsigned char *str);
```

----Now you can #include system.h and get started on coding it!----

Update main:

```
#include system.h
/* Functions for the system go here */
void main()
{

	for(;;);
}
```

To compile:

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

Notice how things are omited during the compile: “-fomit-frame-pointer, -nosttdinc & fno-builtin”

Creating the boot floppy:

To run this basic system you need to make your floppy image. Your linker should spit out a binary for you called kernel.bin.

```
cat  stage1 stage2 pad kernel.bin > floppy.img //On *Nix systems
copy /b stage1+stage2+pad+kernel.bin   floppy.img //Win32
```

To run with QEMU:

```
Qemu -fda floppy.img
```

Whats Next?

-Writing Drivers

-Basic Graphics

-Memory Management

-Filesystem