# Butterfly OS

## tool/mkiso

### Makefile

|  |
| --- |
| BUILD = build  all:  mkdir -p $(BUILD)  gcc -m32 mkiso.c -o $(BUILD)/mkiso  #$(BUILD)/mkiso $(BUILD)/iso\_header.bin $(BUILD)/iso\_bootable.bin  clean:  rm -rvf $(BUILD) |

### mkiso.c

|  |  |
| --- | --- |
| #ifndef \_MKISO\_C\_  #define \_MKISO\_C\_  #include <stdio.h>  #include <stdlib.h>  typedef unsigned char u8;  typedef unsigned short u16;  typedef unsigned int u32;  typedef unsigned long long u64;  typedef struct iso\_struct  {  u8 boot\_record\_indicator;  char iso\_9660\_identifier[5];  u8 version\_of\_this\_descriptor;  char boot\_system\_identifier[23];  u8 pad0;  u8 pad1;  u8 pad2[32];  u8 pad3[7];  u8 first\_sector0;  u8 first\_sector1;  u8 first\_sector2;  u8 first\_sector3;  u8 pad4[1973];  } iso\_struct;  typedef struct iso\_boot\_catalog  {  u8 boot\_indicator;  u8 platform\_id;  u8 reserved0;  u8 reserved1;  u8 pad0[26];  u8 end\_55;  u8 end\_AA;  u8 bootable;  u8 boot\_media\_type;  //if 0 use 0x07c0  u8 load\_segment\_for\_image0;  u8 load\_segment\_for\_image1;  u8 system\_type;  u8 pad1;  u8 sector\_count0;  u8 sector\_count1;  u32 start\_address\_of\_the\_virtual\_disk;  u8 pad2[20];  u8 pad3[1984];  } iso\_boot\_catalog;  typedef iso\_struct iso;  typedef iso\_boot\_catalog iso\_boot;  void str\_copy(char \*to, char \*from, int len)  {  int i=0;  for (; i<len; ++i) {  to[i] = from[i];  }  }  int main(int argc, char \*argv[])  {  iso iso\_m;  iso\_m.boot\_record\_indicator = 0;  str\_copy(iso\_m.iso\_9660\_identifier, "CD001", 5);  iso\_m.version\_of\_this\_descriptor = 1; | str\_copy(iso\_m.boot\_system\_identifier, "EL TORITO SPECIFICATION", 23);  iso\_m.pad0 = 0;  iso\_m.pad1 = 0;  int i=0;  for (i=0; i<32; ++i) {  iso\_m.pad2[i] = 0;  }  for (i=0; i<7; ++i) {  iso\_m.pad3[i] = 0;  }  iso\_m.first\_sector0 = 0x13;  iso\_m.first\_sector1 = 0x0;  iso\_m.first\_sector2 = 0x0;  iso\_m.first\_sector3 = 0x0;    for (i=0; i<1973; ++i) {  iso\_m.pad4[i] = 0;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  iso\_boot iso\_b;  iso\_b.boot\_indicator = 1;  iso\_b.platform\_id = 0;  iso\_b.reserved0 = 0;  iso\_b.reserved1 = 0;  i=0;  for (i=0; i<26; ++i) {  iso\_b.pad0[i] = 0;  }  iso\_b.pad0[24] = 0xAA;  iso\_b.pad0[25] = 0x55;  iso\_b.end\_55 = 0x55;  iso\_b.end\_AA = 0xAA;  iso\_b.bootable = 0x88;  iso\_b.boot\_media\_type = 0;  iso\_b.load\_segment\_for\_image0 = 0x00;  iso\_b.load\_segment\_for\_image1 = 0x00;  iso\_b.system\_type = 0;  iso\_b.pad1 = 0;  iso\_b.sector\_count0 = 0x04;  iso\_b.sector\_count1 = 0x02;  iso\_b.start\_address\_of\_the\_virtual\_disk = 0x00000125;  for (i=0; i<20; ++i) {  iso\_b.pad2[i] = 0;  }  for (i=0; i<1984; ++i) {  iso\_b.pad3[i] = 0;  }  FILE \*p = fopen(argv[1], "wb");  fwrite((char \*)&iso\_m, sizeof(char), sizeof(iso\_m), p);  fclose(p);  FILE \*p2 = fopen(argv[2], "wb");  fwrite((char \*)&iso\_b, sizeof(char), sizeof(iso\_b), p2);  fclose(p2);  return 0;  }  #endif //\_MKISO\_C\_ |

## lidqos

### boot.S

|  |  |
| --- | --- |
| #ifndef \_BOOT\_S\_  #define \_BOOT\_S\_  //16位操作数和16位寻址模式  .code16  //开始过程  .global \_start  //数据段  .section .data  //代码段  .section .text  //开始  \_start:  //将es设为0xb800  movw $0xb800, %ax  movw %ax, %es  //将ds设为0x07c0  movw $0x07c0, %ax  movw %ax, %ds  //将si指向Hello world!字符串首地址  movw $\_str, %si  //清空di  xorw %di, %di  \_copy: | //判断字符串结束符\0  cmp $0, %ds:(%si)  je \_loop  //拷贝字符串到al  movb %ds:(%si), %al  movb $0x07, %ah  movw %ax, %es:(%di)  //si加1  addw $1, %si  //di加2  addw $2, %di  jmp \_copy  \_loop:  jmp \_loop  \_str:  .string "Hello world!\0"  //占位，从此行开始到0x1fe止匀为0x90也就是nop  .org 0x1fe, 0x90  //在0x1ff终止符0xaa55  .word 0xaa55  //一共0x200个字节  #endif |

### Makefile

|  |
| --- |
| #工程名称  PROJECT = lidqos  BUILD\_PATH = build  MKDIR = mkdir  BOOT = boot  KERNEL = kernel  MKISO = mkiso  PSHELL = pshell  TOOL = ../tool  FS = fs  PSHELL = shell  ECC = ecc  #命令  #编译c程序  CC = gcc  #连接object文件到elf可执行文件  LD = ld  #将elf可执行文件转为纯机器码的bin文件  OC = objcopy  #将elf可执行文件反汇编到S文件  OD = objdump  #制作iso光盘文件  DD = dd  #gcc参数，编译32位代码  #-mpreferred-stack-boundary=2  GC\_PARAMS = -fno-builtin -Iinclude -c -m32 -std=c99  GC\_B\_PARAMS = $(GC\_PARAMS)  GC\_K\_PARAMS = $(GC\_PARAMS)  GC\_S\_PARAMS = $(GC\_PARAMS)  #ld连接参数，  LD\_BOOT\_PARAMS = -m elf\_i386 -Ttext 0x0 -Tdata 0x1000  #ld连接内核参数  LD\_KERNEL\_PARAMS = -m elf\_i386 -Ttext 0x2000 -Tdata 0x0  #ld连接shell参数  LD\_PSHELL\_PARAMS = -m elf\_i386 -Ttext 0x0  #转bin文件参数  OC\_PARAMS = -O binary  #转16位bin程序参数  OD\_PARAMS\_16 = -S -D -m i8086  #转16位at&t汇编  OD\_PARAMS\_16\_ATT = $(OD\_PARAMS\_16) -M att  #转16位intel汇编  OD\_PARAMS\_16\_INTEL = $(OD\_PARAMS\_16) -M intel  #转32位bin程序参数  OD\_PARAMS\_32 = -S -D -m i386  #转32位at&t汇编  OD\_PARAMS\_32\_ATT = $(OD\_PARAMS\_32) -M att  #转32位intel汇编  OD\_PARAMS\_32\_INTEL = $(OD\_PARAMS\_32) -M intel  #dd参数，光盘以0x800(2048)为一个扇区  DD\_PARAMS = bs=2048  #iso\_header.bin所在扇区0x11  DD\_PARAMS\_HEADER = 17  #iso\_bootable.bin所在扇区0x13  DD\_PARAMS\_BOOTABLE = 19  #boot.bin所在扇区0x125  DD\_PARAMS\_BOOT = 293  #kernel.bin所在扇区0x129  DD\_PARAMS\_KERNEL = 297  #文件系统起始扇区号0x400  DD\_PARAMS\_FILESYS = 1024  #结尾内容所在扇区0x800  DD\_PARAMS\_END = 4096  ##################################################################################################################  # 创建文件夹 光盘iso文件 启动程序 系统内核 shell程序  all: $(MKDIR) $(MKISO) $(BOOT) $(KERNEL) $(PSHELL)  ##################################################################################################################  #创建文件夹  $(MKDIR):  mkdir -p $(BUILD\_PATH)/$(MKISO) \  $(BUILD\_PATH)/$(BOOT) \  $(BUILD\_PATH)/$(KERNEL) \  $(BUILD\_PATH)/$(PSHELL)  ##################################################################################################################  #光盘iso文件  $(MKISO): $(MKISO)\_mk $(MKISO)\_S  $(DD) $(DD\_PARAMS) seek=$(DD\_PARAMS\_HEADER) count=1 conv=notrunc if=$(BUILD\_PATH)/$(MKISO)/iso\_header.bin of=$(BUILD\_PATH)/$(MKISO)/$(PROJECT).iso  $(DD) $(DD\_PARAMS) seek=$(DD\_PARAMS\_BOOTABLE) count=1 conv=notrunc if=$(BUILD\_PATH)/$(MKISO)/iso\_bootable.bin of=$(BUILD\_PATH)/$(MKISO)/$(PROJECT).iso  $(DD) $(DD\_PARAMS) seek=$(DD\_PARAMS\_END) count=1 conv=notrunc if=/dev/zero of=$(BUILD\_PATH)/$(MKISO)/$(PROJECT).iso  #运行mkiso生成iso头内容  $(MKISO)\_mk: $(MKISO)\_run  $(BUILD\_PATH)/$(MKISO)/$(MKISO) $(BUILD\_PATH)/$(MKISO)/iso\_header.bin $(BUILD\_PATH)/$(MKISO)/iso\_bootable.bin  #编译mkiso程序反汇编文件  $(MKISO)\_S: $(MKISO)\_run  $(OD) $(OD\_PARAMS\_32\_ATT) $(BUILD\_PATH)/$(MKISO)/$(MKISO) > $(BUILD\_PATH)/$(MKISO)/$(MKISO)\_att.S  $(OD) $(OD\_PARAMS\_32\_INTEL) $(BUILD\_PATH)/$(MKISO)/$(MKISO) > $(BUILD\_PATH)/$(MKISO)/$(MKISO)\_intel.S  #编译mkiso程序  $(MKISO)\_run:  $(CC) -m32 $(TOOL)/$(MKISO)/$(MKISO).c -o $(BUILD\_PATH)/$(MKISO)/$(MKISO)  ##################################################################################################################  #启动程序  $(BOOT): $(BOOT)\_bin $(BOOT)\_S  $(DD) $(DD\_PARAMS) seek=$(DD\_PARAMS\_BOOT) count=4 conv=notrunc if=$(BUILD\_PATH)/$(BOOT)/$(BOOT).bin of=$(BUILD\_PATH)/$(MKISO)/$(PROJECT).iso  #启动程序的二进制文件  $(BOOT)\_bin: $(BOOT)\_elfs  $(OC) $(OC\_PARAMS) $(BUILD\_PATH)/$(BOOT)/$(BOOT) $(BUILD\_PATH)/$(BOOT)/$(BOOT).bin  #启动程序反汇编文件  $(BOOT)\_S: $(BOOT)\_elfs  $(OD) $(OD\_PARAMS\_16\_ATT) $(BUILD\_PATH)/$(BOOT)/$(BOOT) > $(BUILD\_PATH)/$(BOOT)/$(BOOT)\_att.S  $(OD) $(OD\_PARAMS\_16\_INTEL) $(BUILD\_PATH)/$(BOOT)/$(BOOT) > $(BUILD\_PATH)/$(BOOT)/$(BOOT)\_intel.S  #启动程序的elf可执行文件  $(BOOT)\_elfs: $(BOOT)\_objs  $(LD) $(LD\_BOOT\_PARAMS) -o $(BUILD\_PATH)/$(BOOT)/$(BOOT) -e \_start $(BUILD\_PATH)/$(BOOT)/$(BOOT).o \  $(BUILD\_PATH)/$(BOOT)/main.o  #启动程序的obj文件  $(BOOT)\_objs:  $(CC) $(GC\_B\_PARAMS) $(BOOT)/$(BOOT).S -o $(BUILD\_PATH)/$(BOOT)/$(BOOT).o  $(CC) $(GC\_B\_PARAMS) $(BOOT)/main.c -o $(BUILD\_PATH)/$(BOOT)/main.o    ##################################################################################################################  #系统内核  $(KERNEL): $(KERNEL)\_bin $(KERNEL)\_S  $(DD) $(DD\_PARAMS) seek=$(DD\_PARAMS\_KERNEL) count=256 conv=notrunc if=$(BUILD\_PATH)/$(KERNEL)/$(KERNEL).bin of=$(BUILD\_PATH)/$(MKISO)/$(PROJECT).iso  #内核程序的二进制文件  $(KERNEL)\_bin: $(KERNEL)\_elfs  $(OC) $(OC\_PARAMS) $(BUILD\_PATH)/$(KERNEL)/$(KERNEL) $(BUILD\_PATH)/$(KERNEL)/$(KERNEL).bin  #内核程序反汇编文件  $(KERNEL)\_S: $(KERNEL)\_elfs  $(OD) $(OD\_PARAMS\_32\_ATT) $(BUILD\_PATH)/$(KERNEL)/$(KERNEL) > $(BUILD\_PATH)/$(KERNEL)/$(KERNEL)\_att.S  $(OD) $(OD\_PARAMS\_32\_INTEL) $(BUILD\_PATH)/$(KERNEL)/$(KERNEL) > $(BUILD\_PATH)/$(KERNEL)/$(KERNEL)\_intel.S  #内核程序的elf可执行文件  $(KERNEL)\_elfs: $(KERNEL)\_objs  $(LD) $(LD\_KERNEL\_PARAMS) -o $(BUILD\_PATH)/$(KERNEL)/$(KERNEL) -e start\_kernel $(BUILD\_PATH)/$(KERNEL)/$(KERNEL).o \  $(BUILD\_PATH)/$(KERNEL)/printf.o  #内核程序的obj文件  $(KERNEL)\_objs:  $(CC) $(GC\_K\_PARAMS) $(KERNEL)/$(KERNEL).c -o $(BUILD\_PATH)/$(KERNEL)/$(KERNEL).o  $(CC) $(GC\_K\_PARAMS) printf/printf.c -o $(BUILD\_PATH)/$(KERNEL)/printf.o    ##################################################################################################################  #外壳程序  $(PSHELL): $(PSHELL)\_elfs $(PSHELL)\_S  #外壳程序反汇编文件  $(PSHELL)\_S: $(PSHELL)\_elfs    #外壳程序的elf可执行文件  $(PSHELL)\_elfs: $(PSHELL)\_ecc  #外壳程序的ecc可重定向文件  $(PSHELL)\_ecc: $(PSHELL)\_objs    #外壳程序的obj文件  $(PSHELL)\_objs:  clean:  rm -rvf $(BUILD\_PATH)  mkdir $(BUILD\_PATH) |

### lidqos/boot

#### boot.S

|  |  |
| --- | --- |
| #ifndef \_BOOT\_S  #define \_BOOT\_S  //16位操作数和16寻址模式  .code16  #include <boot/boot.h>  //开始过程  .global \_start, \_to\_the\_protect\_mode  //数据段  .section .data  //代码段  .section .text  //开始  \_start:  //设定段寄存器为0x7c00  movw %cs, %ax  movw %ax, %ds  movw %ax, %es  movw %ax, %ss  //清空%sp  xorw %sp, %sp  //跳转到0x7C00运程程序  ljmp $\_SEG\_BOOT, $\_setup  \_setup:  //将0x7c00处的boot程序copy到\_SEG\_MAIN处  calll \_copy\_boot    //跳转到0x90000处来执行程序  ljmp $\_SEG\_MAIN, $\_boot  //0x90000  \_boot:  //设置data段ds、es和ss:sp  movw $\_SEG\_MAIN, %ax  movw %ax, %ds  movw %ax, %es  movw %ax, %fs  movw %ax, %gs  movw %ax, %ss  xorw %sp, %sp  //测试，如果显示字符a说明程序执行正常  //calll \_display\_ASCII  //调用\_\_SEG\_MAIN处的main()  calll main  \_boot\_end: nop  //将0x7c00处的boot程序copy到0x90000处  \_copy\_boot:  //保存现场  pushw %ax  pushw %bx  pushw %cx  pushw %dx  pushw %es  pushw %ds  pushw %si  pushw %di  //将es和di设置为0x90000  xorw %ax, %ax  movw $\_SEG\_MAIN, %ax  movw %ax, %es  xorw %ax, %ax  movw %ax, %di  //将ds和si设置为0x7c00  xorw %ax, %ax  movw $\_SEG\_BOOT, %ax  movw %ax, %ds  xorw %ax, %ax  movw %ax, %si  //将cx设置成启动程序大小  movw $\_SEG\_BOOT\_SIZE, %cx  cld  //循环拷贝启动程序到0x90000  rep movsb %ds:(%si), %es:(%di)  //恢复现场  popw %di  popw %si  popw %ds  popw %es  popw %dx  popw %cx  popw %bx  popw %ax  retl  \_copy\_boot\_end:nop  \_display\_ascii:  //保存现场  pushw %ax | pushw %bx  pushw %cx  pushw %dx  pushw %es  pushw %ds  pushw %si  pushw %di  movw $0xb800, %ax  movw %ax, %es  xorw %ax, %ax  movw %ax, %di  movw $0x0761, %ax  movw %ax, %es:(%di)  //恢复现场  popw %di  popw %si  popw %ds  popw %es  popw %dx  popw %cx  popw %bx  popw %ax  retl  \_display\_ascii\_end:nop  //跳转到保护模式  \_to\_the\_protect\_mode:  //保存现场  pushl %ebp  movl %esp, %ebp  //载入gdt全局描述符  lgdt gdtp  //打开保护模式， 将cr0的0位置成1  movl %cr0, %eax  orl $0x1, %eax  movl %eax, %cr0  //将0x9c00处的kernel程序copy到0x0处  movw $\_GDT\_IND\_KERNEL\_DATA, %ax  movw %ax, %ds  movw %ax, %es    movl $\_SEG\_KERNEL\_PH, %esi  movl $0x0, %edi    //将%ecx寄存器置成内核程序大小  movl $\_KERNEL\_SIZE, %ecx  \_copy\_kernel:  movw %ds:(%esi), %ax  movw %ax, %es:(%edi)  //每次加2  add $0x2, %esi  add $0x2, %edi  sub $0x2, %ecx  cmp $0x0, %ecx  jne \_copy\_kernel  jmp \_copy\_kernel\_end  \_copy\_kernel\_end: nop  //处理所有寄存器，为跳转到保护模式做准备  movw $\_GDT\_IND\_KERNEL\_DATA, %ax  movl $\_SEG\_KERNEL\_DATA\_OFFSET, %ebx  //设定全局选择子  movw %ax, %ds  movw %ax, %es  movw %ax, %fs  movw %ax, %gs  movw %ax, %ss    //设定相对内存地址  xorl %eax, %eax  movl %eax, %edi  movl %eax, %esi    //设定堆栈地址  movl %ebx, %esp  movl %ebx, %ebp    //跳转到(0x000000000)处，不再返回这里  //内核实际地址在\_SEG\_KERNEL\_OFFSET  \_ljmp:  .byte 0xea  \_ljmp\_offset:  .word \_SEG\_KERNEL\_OFFSET  \_ljmp\_section:  .word \_GDT\_IND\_KERNEL    //恢复现场  popl %ebp  retl  \_to\_the\_protect\_mode\_end: nop  //占位，从此行开始到0x01fe都为0x90也就是nop  .org 0x1fe, 0x90  //在0x1ff终止符0xaa55  .word 0xaa55  //一共0x200个字节  #endif //\_BOOT\_S |

#### main.c

|  |  |
| --- | --- |
| #include <boot/code16.h>  #include <boot/main.h>  //全局描述符表个数  #define GDT\_MAX\_SIZE (3)  //全局描述符表  s\_gdt gdts[GDT\_MAX\_SIZE];  //全局描述符  s\_gdtp gdtp;  int main(int argc, char \*\*args)  {  //关中断  cli();  //打开A20, 启用CPU的32根内存寻址线，可进行4GB内存寻址  enable\_a20();  //设置全局描述符  set\_gdt();  //跳转到保护模式，不再返回，直接启动内核程序  to\_protect\_mode();  return 0;  }  /\*  \*addr\_to\_gdt:将物理地址转为gdt描述地址，并存放到gdt全局描述符当中  \* - u32 addr : 32位物理地址  \* - s\_gdt \*gdt :gdt全局描述符  \* - u8 cs\_ds :0为代码段，1为数据段  \* return void  \*/  void addr\_to\_gdt(u32 addr, s\_gdt \*gdt, u8 cs\_ds)  {  //最大尺寸低16位  gdt->limit = 0xffff;  //基地址低16位  gdt->baseaddr = addr & 0xffff;  //基地址中8位  gdt->baseaddr2 = (addr >> 16) & 0xff;  //如果是代码段  if (cs\_ds == 0)  {  //代码段描述  gdt->p\_dpl\_type\_a = 0x9a;  }  //如果是数据段  else  {  //数据段描述  gdt->p\_dpl\_type\_a = 0x92;  }  //相关标识和最大尺寸高位  gdt->uxdg\_limit2 = 0xcf;  //基地址高8位  gdt->baseaddr3 = (addr >> 24) & 0xff; | }  /\*  \* set\_gdt: 设置全局gdt描述符， 其中包括默认地址，内核地址，显存地址等  \* return void  \*/  void set\_gdt()  {  //默认地址  u32 addr = 0x0;  //默认空描述符0x0  gdts[0].gdt = 0x0;  gdts[0].gdt2 = 0x0;  //设置kernel的全局描述符0x8  addr = 0x0;  addr\_to\_gdt(addr, &gdts[1], 0);  //设置kernel data的全局描述符0x10  addr = 0x0;  addr\_to\_gdt(addr, &gdts[2], 1);  //设置gdt描述符  //gdt总数-1  gdtp.gdt\_lenth = sizeof(s\_gdt) \* GDT\_MAX\_SIZE - 1;  //gdt全局描述符的高16位地址  gdtp.gdt\_addr2 = ds() \* 0x10 >> 16;  //gdt全局描述符地址低16位  gdtp.gdt\_addr = (u32)gdts;  }  /\*  \* enable\_a20:打开A20  \* return : void  \*/  void enable\_a20()  {  u8 port\_a;  //从0x92端口读入数据  port\_a = inb\_p(0x92);  //打开A20  port\_a |= 0x02;  //不重置电脑  port\_a &= ~0x01;  //向0x92输出设定后的值  outb\_p(port\_a, 0x92);  }  /\*  \* to\_protect\_mode : 跳转到保护模式，不再返回，直接启动内核程序  \* return : void  \*/  void to\_protect\_mode()  {  //跳转到保护模式，不再返回，直接启动内核程序  \_to\_the\_protect\_mode();  } |

### lidqos/include/boot

#### boot.h

|  |  |
| --- | --- |
| ifndef \_BOOT\_H\_  #define \_BOOT\_H\_  //定义启动段地址为0x7c00  #define \_SEG\_BOOT 0x7c0  //定义启动程序大小0x800\*0x4=0x2000  #define \_SEG\_BOOT\_SIZE 0x2000  //定义内核所在位置0x9c00 (\_SEG\_BOOT+\_SEG\_BOOT\_SIZE)  #define \_SEG\_KERNEL 0x9c0  #define \_SEG\_KERNEL\_PH 0x9c00  //定义启动main函数段地址为0x90000 | #define \_SEG\_MAIN 0x9000  //定义内核程序 .Ttext的offset  #define \_SEG\_KERNEL\_OFFSET 0x2000  #define \_SEG\_KERNEL\_DATA\_OFFSET 0x400000  //内核大小256个扇区\*2048  #define \_KERNEL\_SIZE 0x80000  //定义内核地址为0的全局描述符的选择子  #define \_GDT\_IND\_KERNEL 0x8  #define \_GDT\_IND\_KERNEL\_DATA 0x10  #endif //\_BOOT\_H |

#### code16.h

|  |
| --- |
| ##ifndef \_CODE16GCC\_H  #define \_CODE16GCC\_H  //gcc的16位程序声明  \_\_asm\_\_(".code16gcc");  #endif //\_CODE16GCC\_H |

#### main.h

|  |  |
| --- | --- |
| #ifndef \_main\_h  #define \_main\_h  #include <kernel/typedef.h>  #include <kernel/io.h>  int main(int argc, char \*\*args);  void addr\_to\_gdt(u32 addr, s\_gdt \*gdt, u8 cs\_ds);  /\*  \* set\_gdt : 设置全局gdt描述符， 其中包括默认地址，内核地址，显存地址  \* return : void  \*/  void set\_gdt();  /\* | \* enable\_a20 : 打开A20，启用CPU的32根内存寻址， 可进行4GB内存寻址  \* return : void  \*/  void enable\_a20();  /\*  \* to\_protect\_mode : 跳转到保护模式，不再返回，直接启动内核程序  \* return : void  \*/  void to\_protect\_mode();  /\*  \* to\_protect\_mode : 跳转到保护模式，由boot.s中定义  \* return : void  \*/  extern void \_to\_the\_protect\_mode();  #endif //\_main\_h |

### lidqos/include/kernel

#### io.h

|  |  |
| --- | --- |
| /\*  \* 端口输入输出头文件  \*/  #ifndef \_IO\_H  #define \_IO\_H  #include <kernel/typedef.h>  /\*  \* 关中断  \*/  #define cli() \  ({\_\_asm\_\_ volatile("cli");})  /\*  \* 打中断  \*/  #define sti() \  ({\_\_asm\_\_ volatile("sti");})  /\*  \* 载入gdt  \*/  #define load\_gdt(gdtp) \  ({ \  \_\_asm\_\_ volatile("lgdt %0" :: "m"(gdtp)); \ | })  /\*  \* 向端口写一个字节  \*/  static inline void outb\_p(u8 val, u16 port)  {  \_\_asm\_\_ volatile("outb %0, %1" :: "a" (val), "dN" (port));  }  /\*  \* 从端口读入一个字节  \*/  static inline u8 inb\_p(u16 port)  {  u8 val;  \_\_asm\_\_ volatile("inb %%dx, %%al" : "=a"(val) : "dx"(port));  }  static inline u16 ds()  {  u16 ds;  \_\_asm\_\_ volatile("movw %%ds, %0" : "=a" (ds) : );  return ds;  }  #endif //\_IO\_H |

#### kernel.h

|  |
| --- |
| #ifndef \_INCLUDE\_KERNEL\_KERNEL\_H  #define \_INCLUDE\_KERNEL\_KERNEL\_H  int start\_kernel(int argc, char \*\*args);  #endif //\_INCLUDE\_KERNEL\_KERNEL\_H |

#### printf.h

|  |  |
| --- | --- |
| #ifndef \_PRINTF\_H\_  #define \_PRINTF\_H\_  #include <kernel/typedef.h>  #include <kernel/io.h>  void set\_cursor(u16 x, u16 y);  u16 get\_cursor(); | void putascii(u16 x, u16 y, char ch);  void putchar(char ch);  int printf(char \*fmt, ...);  #endif //\_PRINTF\_H\_ |

#### typedef.h

|  |  |
| --- | --- |
| #ifndef \_TYPEDEF\_H  #define \_TYPEDEF\_H  typedef unsigned char u8;  typedef unsigned short u16;  typedef unsigned int u32;  typedef signed int s32;  typedef unsigned long long u64;  #ifndef NULL  #define NULL ((void\*)(0))  #endif //NULL  //GDT全局描述符表  typedef struct gdt\_s  {  union  {  struct  {  u64 lgdt;  };  struct  { | u32 gdt, gdt2;  };  struct  {  u16 limit;  u16 baseaddr;  u8 baseaddr2;  u8 p\_dpl\_type\_a;  u8 uxdg\_limit2;  u8 baseaddr3;  };  };  } s\_gdt;  //GDT全局描述符  typedef struct gdt\_ptr  {  u16 gdt\_lenth;  u16 gdt\_addr;  u16 gdt\_addr2;  } s\_gdtp;  #endif |

### lidqos/kernel

#### kernel.c

|  |  |
| --- | --- |
| #include <kernel/kernel.h>  #include <kernel/printf.h>  //全局字符串指针变量  char \*str = "Hello World!";  //内核启动程序入口  int start\_kernel(int argc, char \*\*args)  {  //显存地址  char \*p = (char \*)0xb8000;  //显示str的内容到显示器上 | for (int i = 0; str[i] != '\0'; i++)  {  putchar(str[i]);  }  //永无休止的循环  for (;;)  {  }  return 0;  } |

### lidqos/printf

#### printf.c

|  |  |
| --- | --- |
| #include <kernel/printf.h>  /\*  \* 设置光标位置  \* u16 x: 光标的横坐标  \* u16 y: 光标的纵坐标  \* \*/  void set\_cursor(u16 x, u16 y)  {  //计算光标的线性位置  u16 cursor\_pos = y \* 80 + x;  //告诉地址寄存器要接下来要使用14号寄存器  outb\_p(14, 0x3d4);  //向光标位置高位寄存器写入值  outb\_p((cursor\_pos >> 8) & 0xff, 0x03d5);  //告诉地址寄存器要接下来使用15号寄存器  outb\_p(15, 0x3d4);  //向光标位置高位寄存器写入值  outb\_p(cursor\_pos & 0xff, 0x03d5);  }  /\*  \* 取得光标位置  \* return: 光标的线性位置  \* \*/  u16 get\_cursor()  {  //告诉地址寄存器要接下来要使用14号寄存器  outb\_p(14, 0x03d4);  //从光标位置高位寄存器读取值  u8 cursor\_pos\_h = inb\_p(0x03d5);  //告诉地址寄存器要接下来使用15号寄存器  outb\_p(15, 0x03d4);  //从光标位置高位寄存器读取值  u8 cursor\_pos\_l = inb\_p(0x3d5);  //返回光标位置  return (cursor\_pos\_h << 8) | cursor\_pos\_l;  }  /\*  \* 根据一个字符的ascii显示到指定位置  \* u16 x:横坐标  \* u16 y:纵坐标  \* char ch: 要显示的字符  \* \*/  void putascii(u16 x, u16 y, char ch)  {  //定义显存的位置  char \*video\_addr = (char \*)0xb8000;  //写入显存 | u32 where = (y \* 80 + x) \* 2;  //显示字符的实际物理地址  u8 \*p = (u8 \*)(video\_addr)+where;  //字符的ascii码  \*p = ch;  \*(p + 1) = 0x07;  }  /\*  \* 显示一个字符到当前光标位置  \* char ch:要显示的字符  \* \*/  void putchar(char ch)  {  //取得当前光标的线性位置  u16 cursor\_pos = get\_cursor();  //计算横纵坐标  u16 x = cursor\_pos % 80;  u16 y = cursor\_pos / 80;  //如果是换行符 \n  if (ch == 0xa)  {  //换行  x = 0;  y++;  set\_cursor(x, y);  }  else if (ch == 0x9) //如果是制表符 \t  {  ch = 0x20;  //显示8个空格  for (int i = 0; i < 8; i++)  {  putascii(x, y, ch);  x++;  set\_cursor(x, y);  }  }  else  {  putascii(x, y, ch);  x++;  set\_cursor(x, y);  }  }  int printf(char \*fmt, ...)  {  return 0;  } |