Lab1 feedback

- Lab1 feedback
- Hints for implementing locking

- Lab1 feedback
  - 必答题

Submission

- much better than Lab0

- however
  - no project in stage 3
    - 121220046, 121220133
  - non-standard character
    - 121220151, 121220158, 121220319, 121242031, 121250198
  - 20% penalty

Shell commands

1. **shell命令** 完成Lab1的内容之后，你整个工程中的.c，.h和.S文件总共有多少行代码？你是使用什么命令得到这个结果的？和Lab1的框架代码相比，你在Lab1中编写了多少行代码？(Hint：使用git checkout可以回到"过去") 你可以把这条命令写入Makefile中，随着实验进度的推进，你可以很方便地统计工程的代码行数，例如敲入make count就会自动运行统计代码行数的命令。再来个难一点的，除去空行之外，你整个工程中的.c，.h和.S文件总共有多少行代码？

```
find . -name '*.[chS]' | xargs wc
```

```
find . -name '*.[chS]' | xargs grep -v '^$' | wc
```

Compiling & linking

2. **编译与链接** 你应该在框架代码中看到include/x86/io.h中看到一些由static inline开头定义的函数. 分别尝试去掉static, inline或者去掉两者, 然后进行编译, 你会看到发生错误. 请解释为什么会发生这些错误? 你有办法证明你的想法吗?

- without "static"
  - multiple definition
- without "inline"
  - unused-function
- without "static" and "inline"
  - multiple definition

Why?

- What does "static" mean?

```
static int x;
static void fun() {
    static int y;
    y ++;
}
```

- Can you use "x" in another source file?
- What is "static" from the view of machine?

static & symbol table

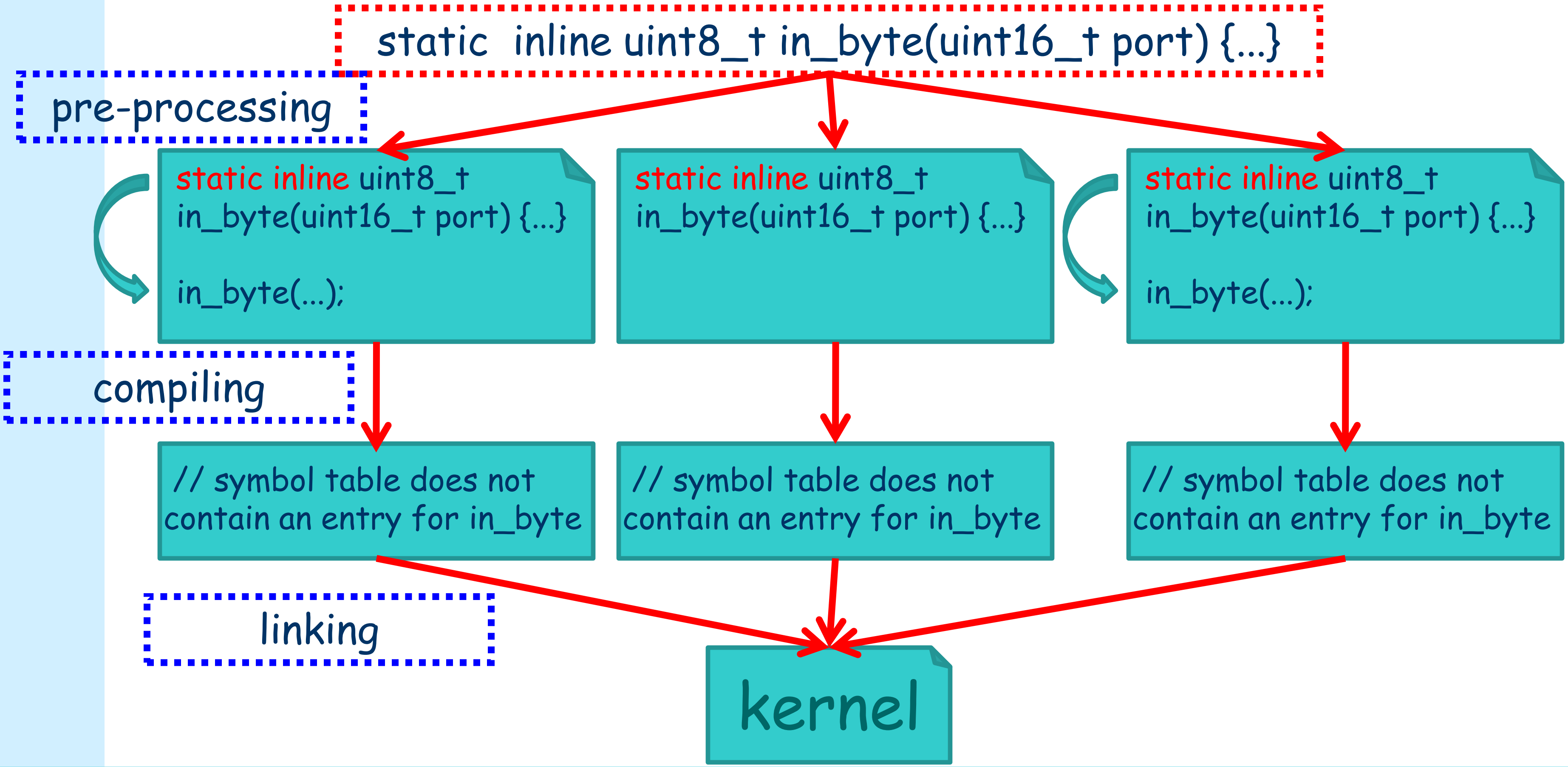
```
// main.c
static void fun() {
    static int yyyyy;
    yyyyy ++;
}

// fun.c
void fun() {
}
```

```
35: 080483f0     0 FUNC    LOCAL  DEFAULT   14 frame_dummy
36: 080495e4     0 OBJECT  LOCAL  DEFAULT   19 __frame_dummy_init_array_
37: 00000000     0 FILE    LOCAL  DEFAULT  ABS main.c
38: 0804841c    26 FUNC    LOCAL  DEFAULT   14 fun
39: 08049708     4 OBJECT  LOCAL  DEFAULT   26 yyyyy.1816
40: 00000000     0 FILE    LOCAL  DEFAULT  ABS fun.c
41: 00000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
42: 080485e0     0 OBJECT  LOCAL  DEFAULT   18 __FRAME_END__
43: 080495ec     0 OBJECT  LOCAL  DEFAULT   21 __JCR_END__
44: 080495e8     0 NOTYPE  LOCAL  DEFAULT   19 __init_array_end
45: 080495f0     0 OBJECT  LOCAL  DEFAULT   22 _DYNAMIC
46: 080495e4     0 NOTYPE  LOCAL  DEFAULT   19 __init_array_start
47: 080496e4     0 OBJECT  LOCAL  DEFAULT   24 _GLOBAL_OFFSET_TABLE_
48: 08048480     5 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
49: 080484ea     0 FUNC    GLOBAL HIDDEN    14 __i686.get_pc_thunk.bx
50: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
51: 080496fc     0 NOTYPE  WEAK   DEFAULT   25 data_start
52: 00000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.0
53: 08049704     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
54: 080484f0     0 FUNC    GLOBAL DEFAULT   15 _fini
55: 0804846c    13 FUNC    GLOBAL DEFAULT   14 fun
56: 080496fc     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
57: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
58: 08049700     0 OBJECT  GLOBAL HIDDEN    25 __dso_handle
59: 0804850c     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
```

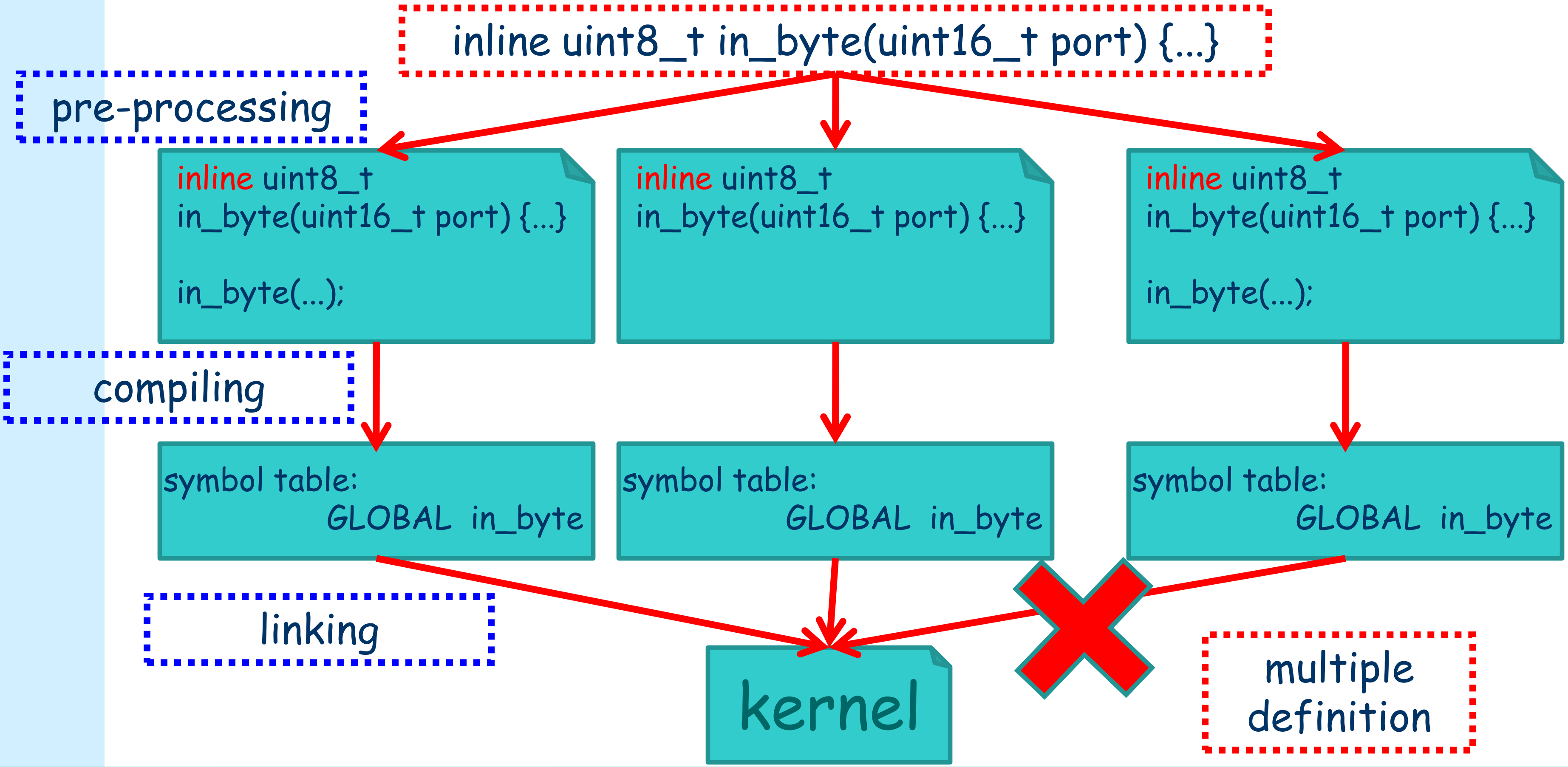
inline

- Unroll the code of a function at calling points.
- After inlining, the body of the function will not exist.
- exceptions:
  - recursive function
  - global function
  - function pointer

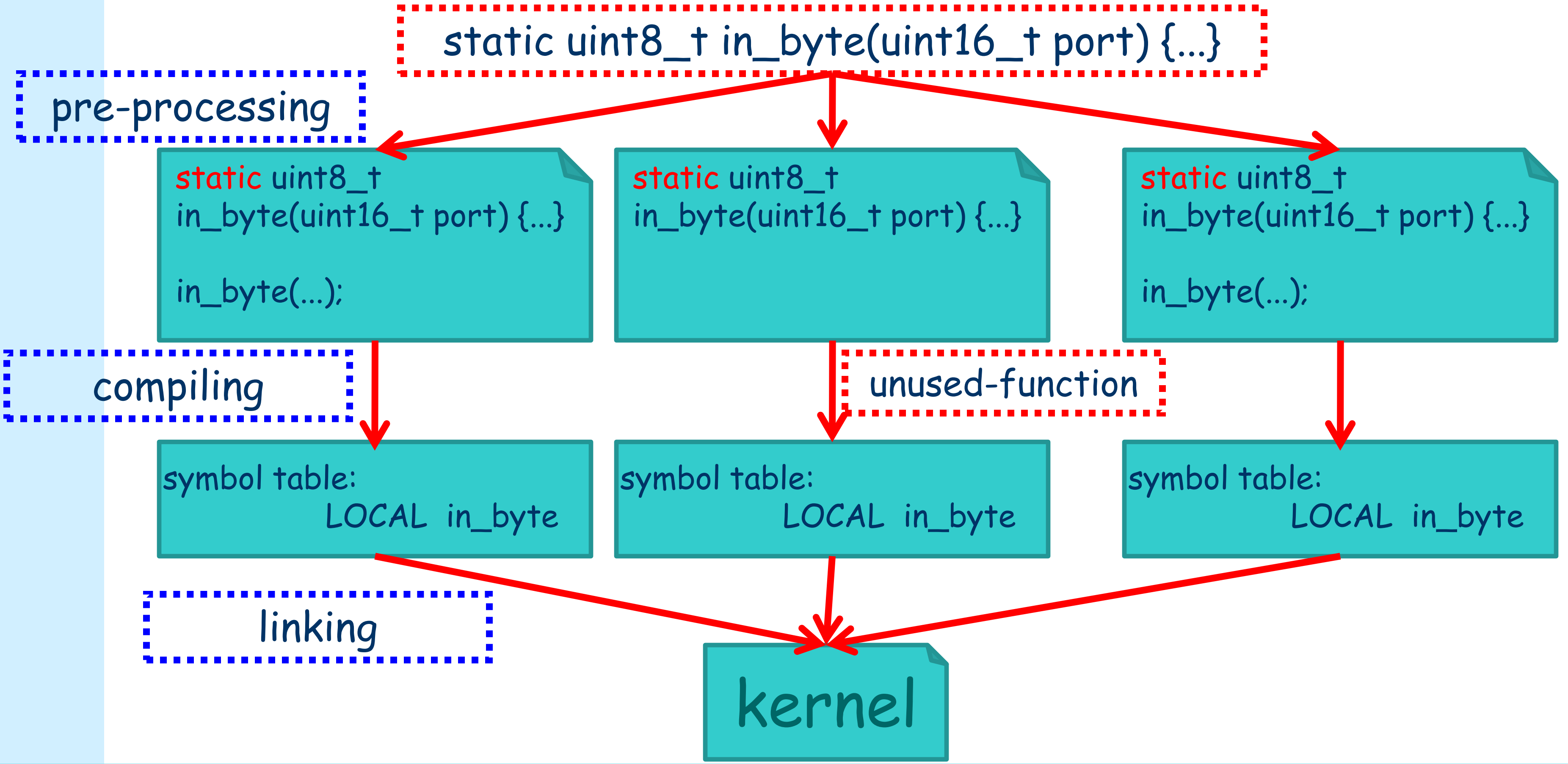
With static & inline



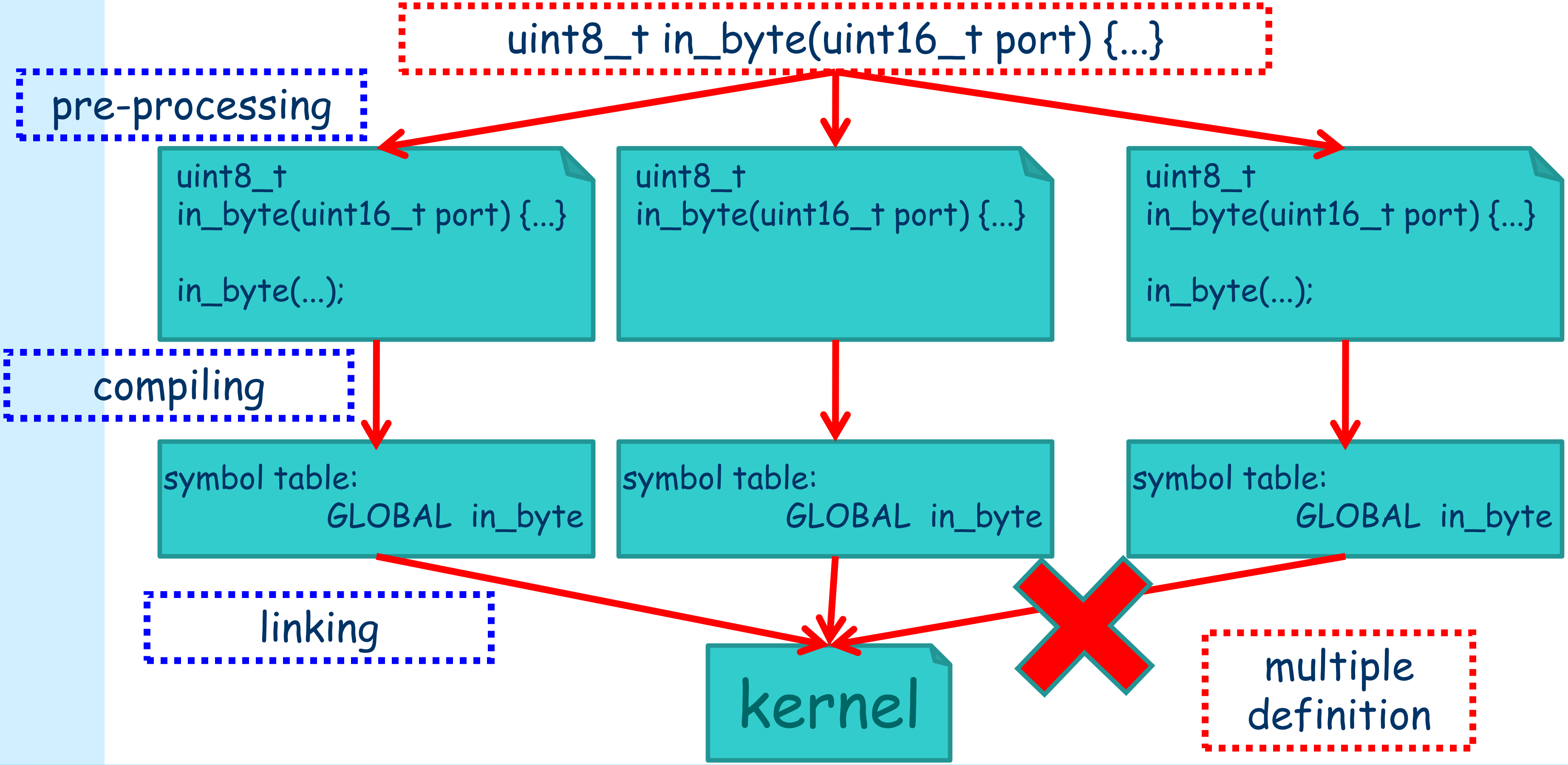
Without static



Without inline



Without static & inline



Unused-function in gcc

- man gcc

```
-Wunused-function
    Warn whenever a static function is declared but not defined or a non-inline static
    function is unused.  This warning is enabled by -Wall.
```

- gcc has many options
  - only search for those you are insterested in

Compiling & linking

3. **编译与链接** 在include/common.h中添加一行

```
volatile static int dummy;
```

然后编译. 请问编译结果含有多少个dummy变量的实体? 你是如何得到这个结果的? 为什么会产生这样的结果?(Hint: 使用readelf命令. 回答完本题后可以删除添加的代码.)

- see symbol table

```
readelf -s kernel | grep -c dummy
```

Why?

- “static” makes it local for each instance of “dummy”.
  - like the previous question
- “volatile” here is to prevent gcc for optimizing out “dummy”s.
- What about
  - volatile static int dummy = 0;
  - volatile int dummy;
  - volatile int dummy = 0;

man

4. **使用man** gcc中的-MD选项有什么作用？-Wall和-Werror有什么作用？为什么要使用-Wall和-Werror？

```
-MD -MD is equivalent to -M -MF file, except that -E is not implied.
    The driver determines file based on whether an -o option is given.
    If it is, the driver uses its argument but with a suffix of .d,
    otherwise it takes the name of the input file, removes any
    directory components and suffix, and applies a .d suffix.

    If -MD is used in conjunction with -E, any -o switch is understood
    to specify the dependency output file, but if used without -E, each
    -o is understood to specify a target object file.

    Since -E is not implied, -MD can be used to generate a dependency
    output file as a side-effect of the compilation process.
```

- Learn to use “man”, learn to use everything.

Makefile

5. **了解Makefile** 在Makefile中有一行

```
-include $(OBJS:.o=.d)
```

请解释这行代码的功能.

- include the contents of all *.d files
  - Where do *.d files come from?
  - What do *.d files contain?
  - Why do we include them?

Makefile

- Where do *.d files come from?

```
CFLAGS  = -m32 -static -ggdb -MD -Wall -I./include -O2 \
          -fno-builtin -fno-stack-protector -fno-omit-frame-pointer
```

- What do *.d files contain?

```
src/kernel/main.o: src/kernel/main.c include/common.h include/types.h \
include/const.h include/assert.h include/x86/x86.h include/x86/cpu.h \
include/x86/memory.h include/x86/io.h include/common.h include/x86/x86.h \
include/memory.h
```

- Why do we include them?
  - What happen if they are not included?

Makfile

6. **了解Makefile** 请描述你在终端敲入make后，make程序如何组织.c，.h和.S文件，最终生成disk.img.（这个问题包括两个方面：Makefile的工作方式和编译链接的过程. Hint：make过程中会用到 implicit rules.）

- fresh make & non-fresh make
- plenty of details
  - variables, functions, include, implicit rules...
- GNU Make Manual

fresh make

```
run: disk.img
	$(QEMU)  -serial stdio disk.img
```

fresh make

```
run: disk.img
	$(QEMU)  -serial stdio disk.img
```

1

```
disk.img: kernel
	@cd boot; make
	cat boot/bootblock kernel > disk.img
```

fresh make

```
run: disk.img
	$(QEMU)  -serial stdio disk.img
```

1

```
disk.img: kernel
	@cd boot; make
	cat boot/bootblock kernel > disk.img
```

2

```
kernel: $(OBJS)
	$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
```

fresh make

```
run: disk.img
        $(QEMU)  -serial stdio disk.img
```

1

```
disk.img: kernel
        @cd boot; make
        cat boot/bootblock kernel > disk.img
```

2

```
kernel: $(OBJS)
        $(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
```

3

```
main.o: main.c
        $(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
```

fresh make

```
run: disk.img
        $(QEMU)  -serial stdio disk.img
```

1

```
disk.img: kernel
        @cd boot; make
        cat boot/bootblock kernel > disk.img
```

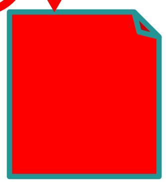
2

```
kernel: $(OBJS)
        $(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
```

3

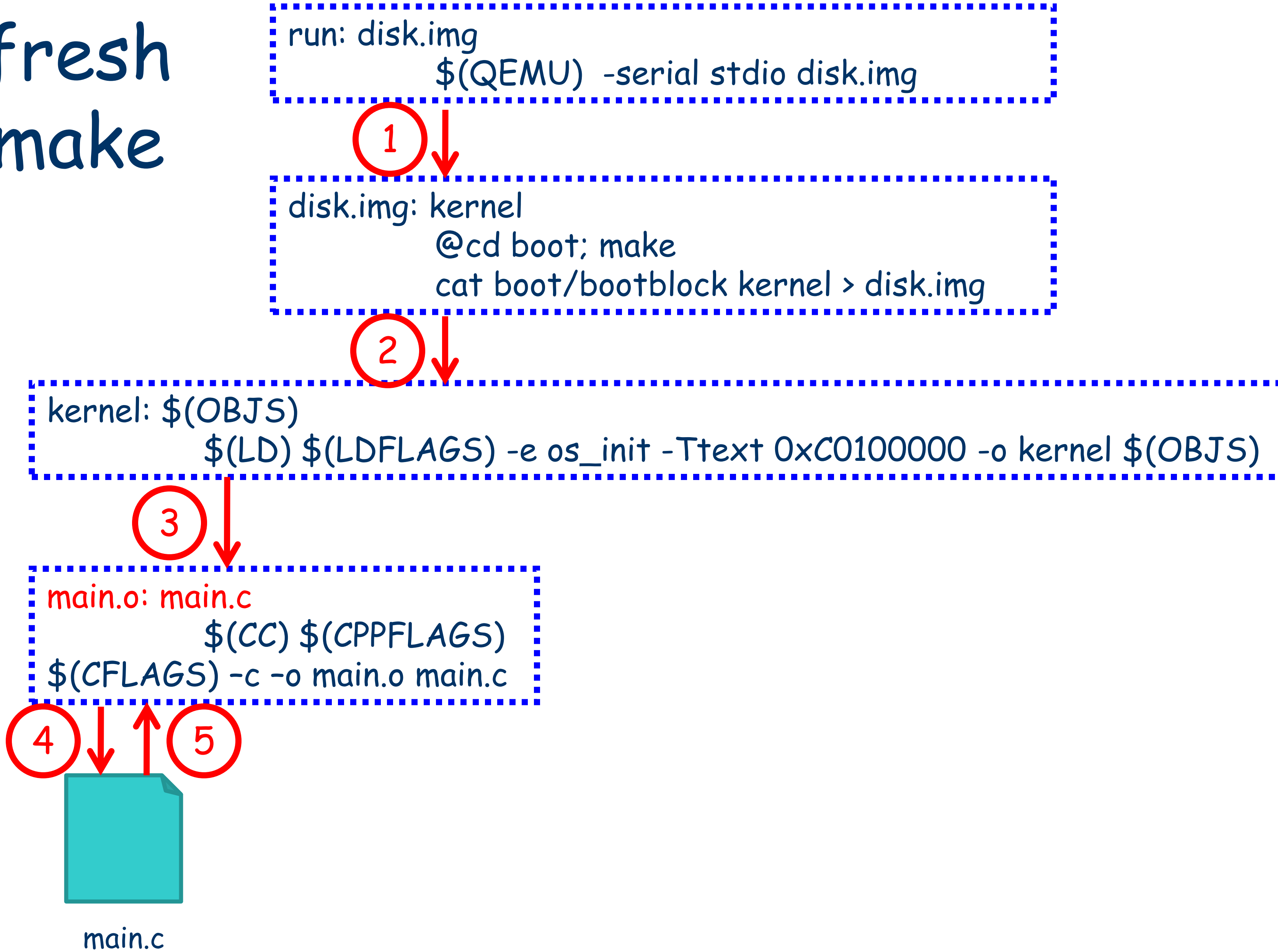
```
main.o: main.c
        $(CC) $(CPPFLAGS)
$(CFLAGS) –c –o main.o main.c
```

4



main.c

fresh make


run: disk.img
$(QEMU) -serial stdio disk.img
1
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
2
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
3
main.o: main.c
$(CC) $(CPPFLAGS) $(CFLAGS) -c -o main.o main.c
4
5
main.c

fresh make

```
run: disk.img
        $(QEMU)  -serial stdio disk.img
```

1

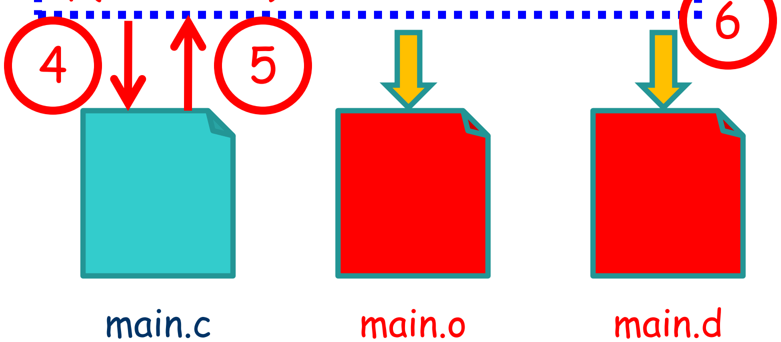
```
disk.img: kernel
        @cd boot; make
        cat boot/bootblock kernel > disk.img
```

2

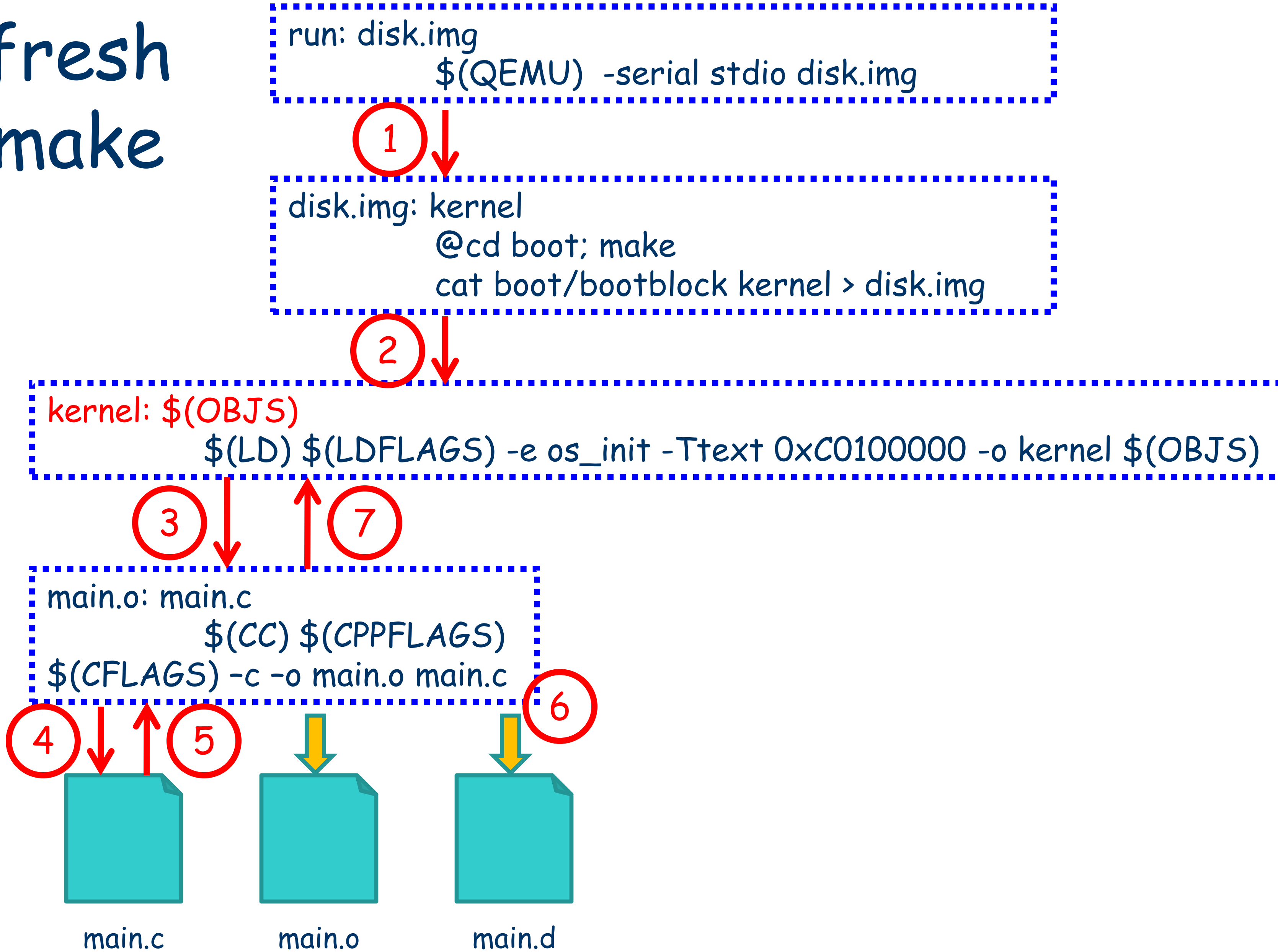
```
kernel: $(OBJS)
        $(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
```

3

```
main.o: main.c
        $(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
```

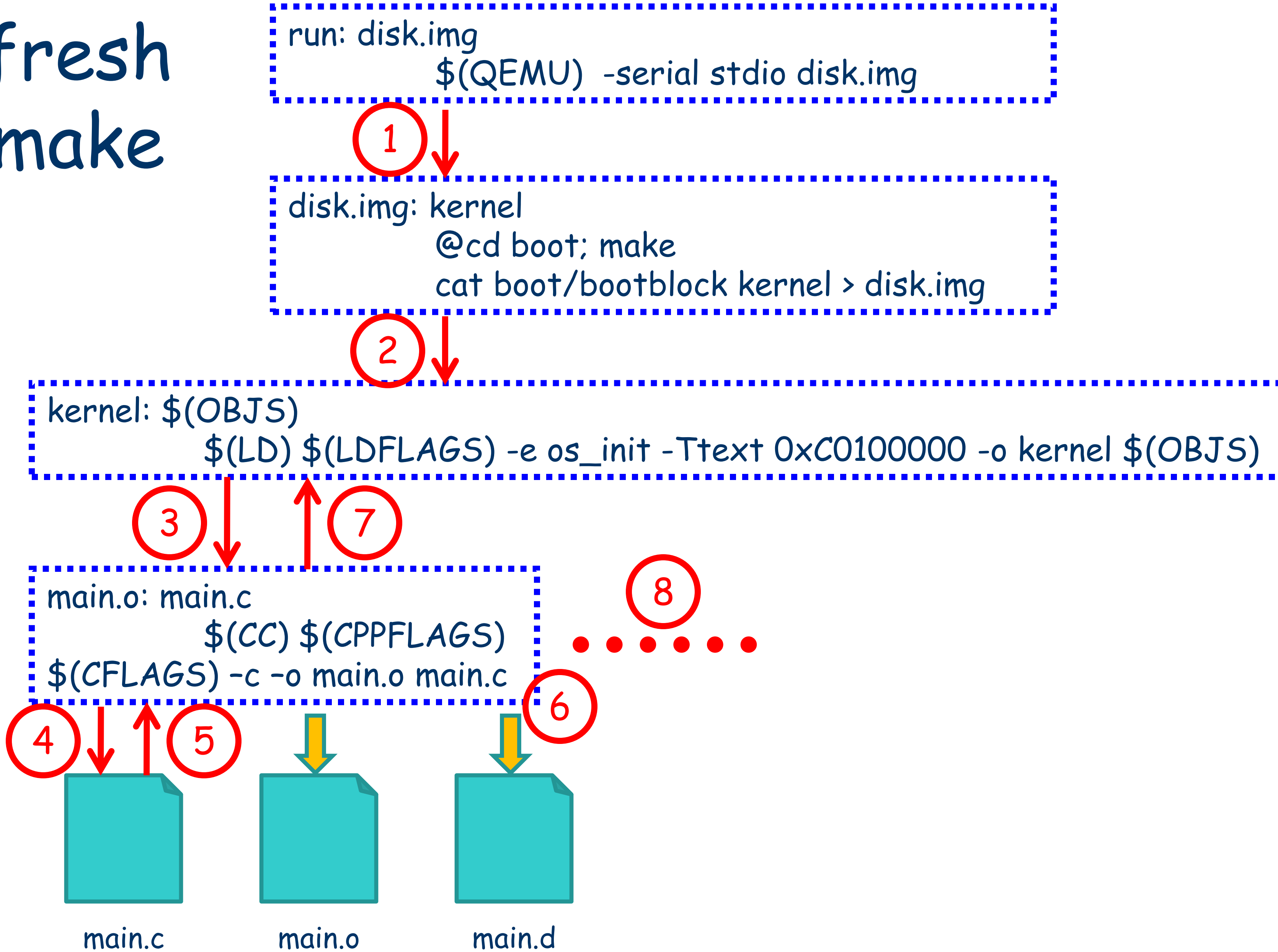


fresh make

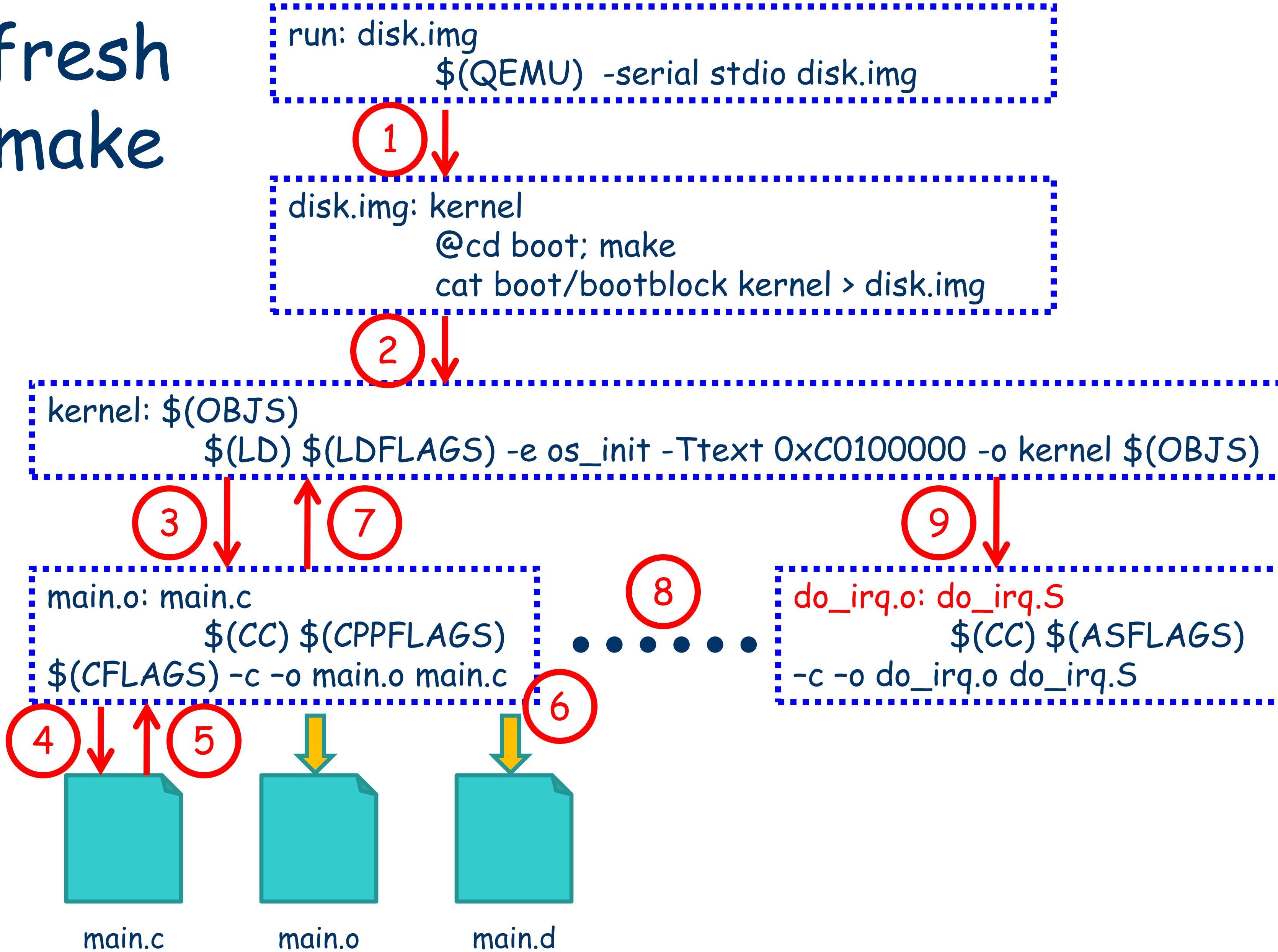


run: disk.img
$(QEMU) -serial stdio disk.img
1
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
2
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
3
7
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
4
5
6
main.c
main.o
main.d

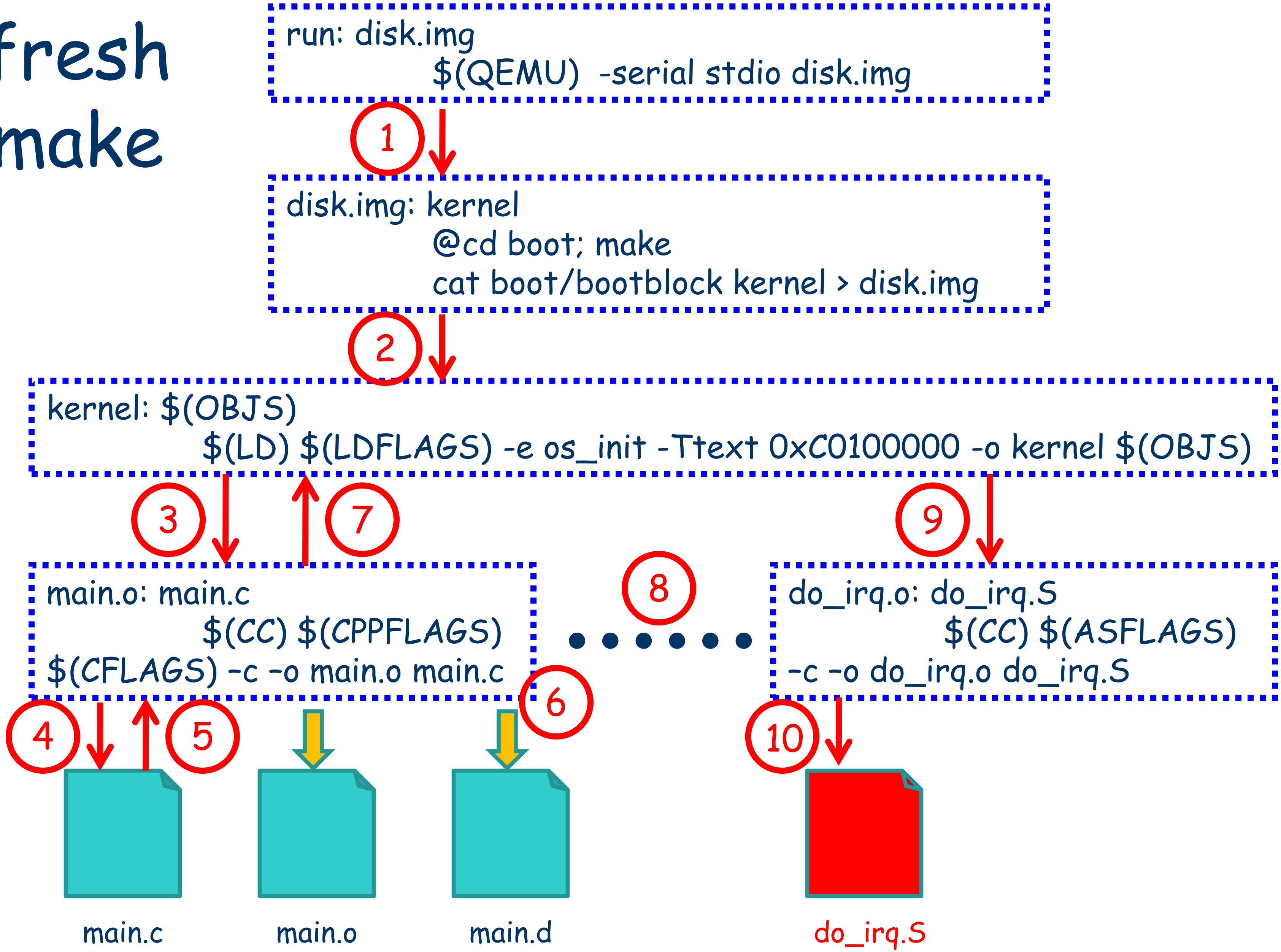
fresh make


run: disk.img
$(QEMU) -serial stdio disk.img
1
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
2
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
3
7
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
8
4
5
6
main.c
main.o
main.d

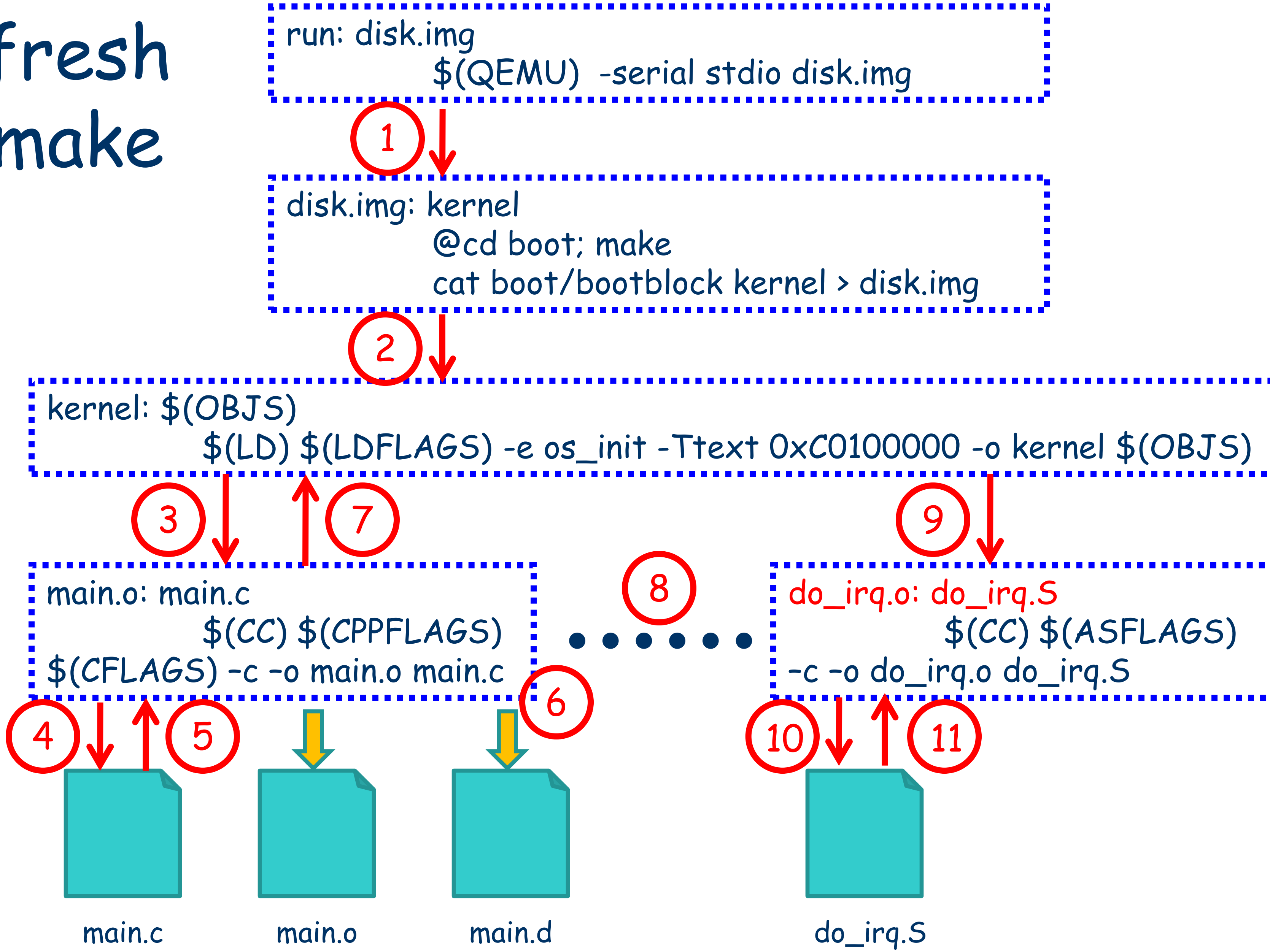
fresh make


run: disk.img
$(QEMU) -serial stdio disk.img
1
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
2
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
3
7
9
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
8
do_irq.o: do_irq.S
$(CC) $(ASFLAGS)
-c -o do_irq.o do_irq.S
4
5
6
main.c
main.o
main.d

fresh make

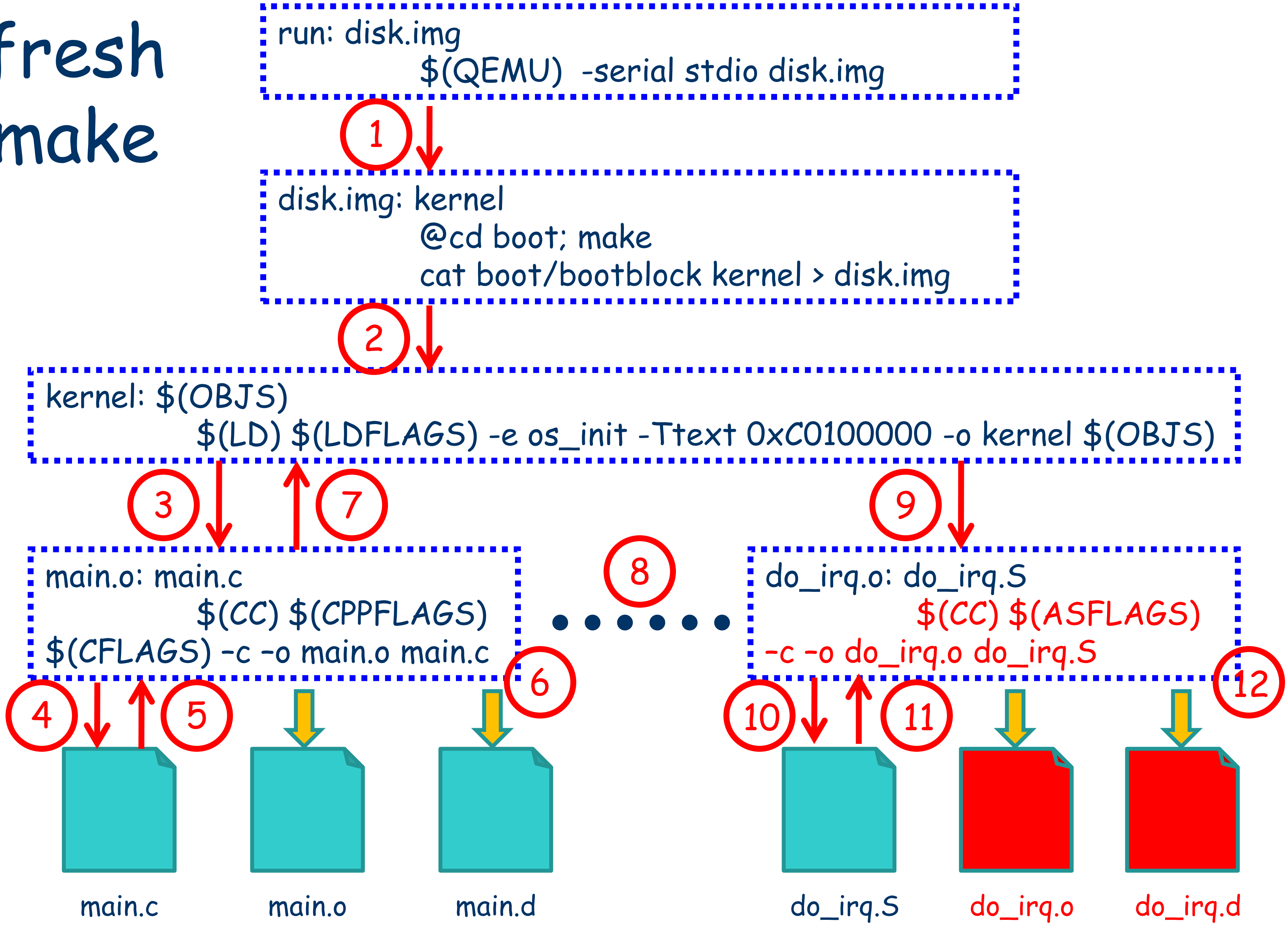

run: disk.img
$(QEMU) -serial stdio disk.img
1
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
2
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
3
7
9
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
8
do_irq.o: do_irq.S
$(CC) $(ASFLAGS)
-c -o do_irq.o do_irq.S
4
5
6
10
main.c
main.o
main.d
do_irq.S

fresh make

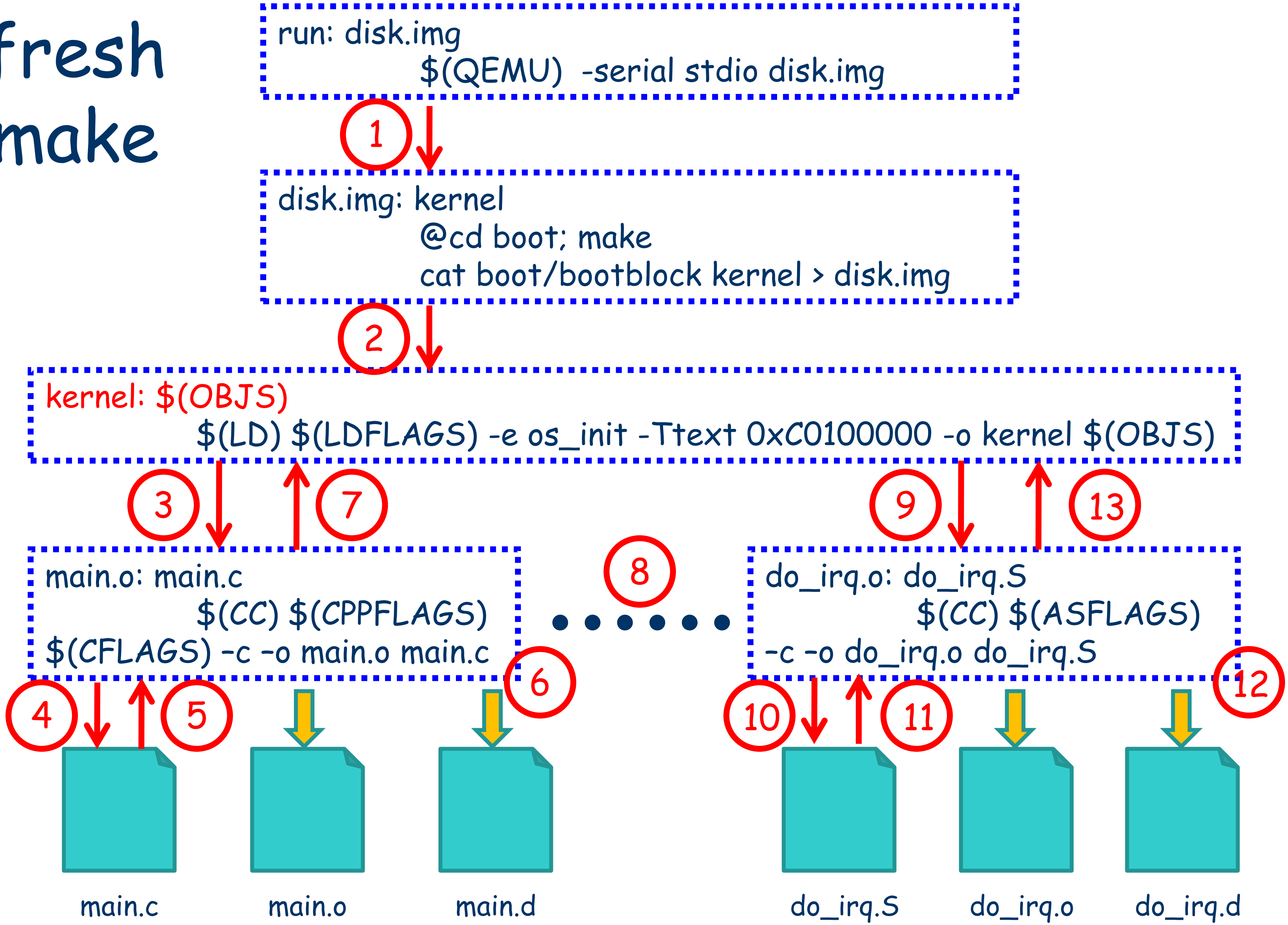


run: disk.img
$(QEMU) -serial stdio disk.img
1
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
2
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
3
7
9
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
8
do_irq.o: do_irq.S
$(CC) $(ASFLAGS)
-c -o do_irq.o do_irq.S
4
5
6
10
11
main.c
main.o
main.d
do_irq.S

fresh make

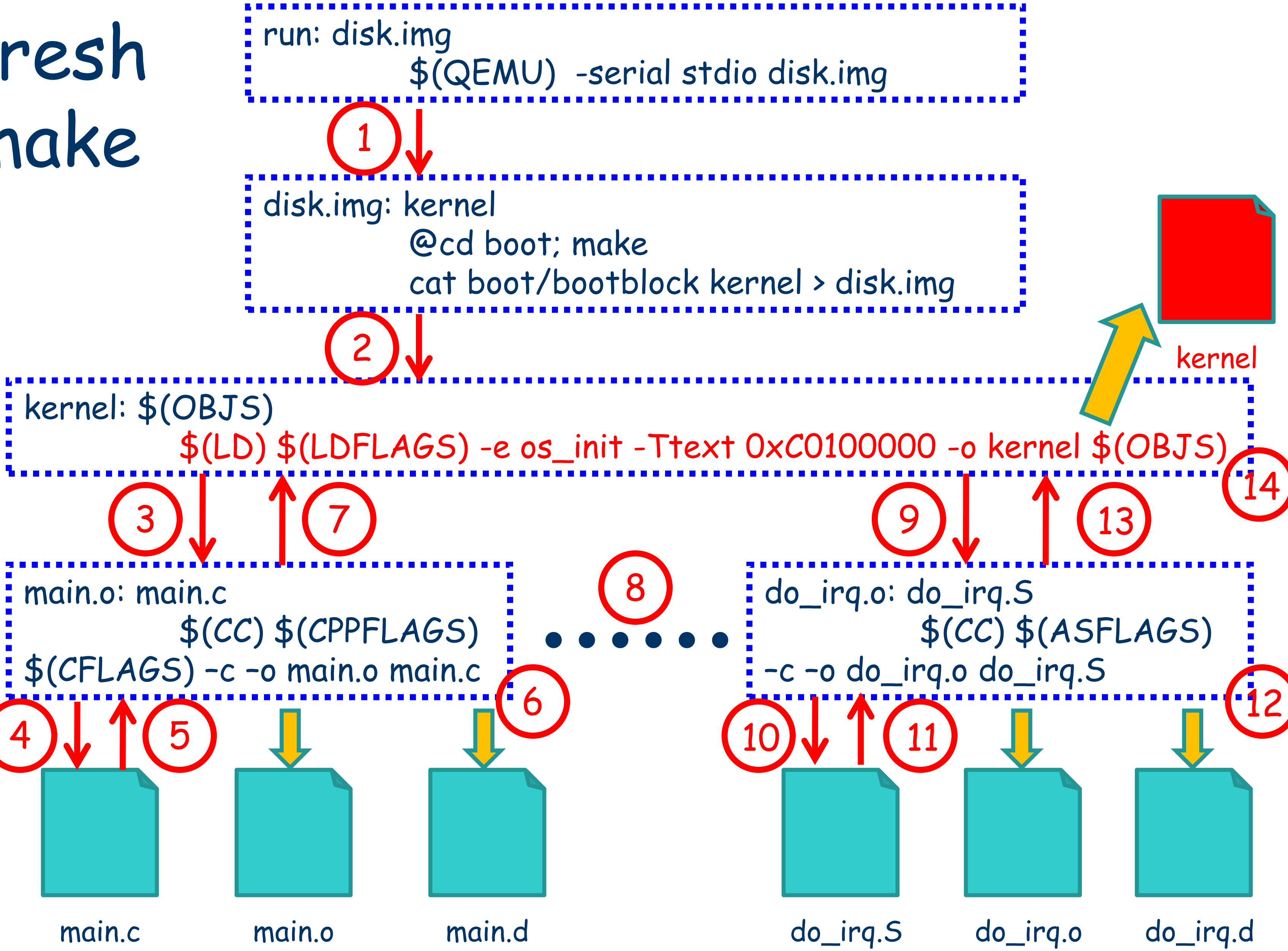

run: disk.img
$(QEMU) -serial stdio disk.img
1
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
2
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
3
7
9
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
8
do_irq.o: do_irq.S
$(CC) $(ASFLAGS)
-c -o do_irq.o do_irq.S
4
5
6
10
11
12
main.c
main.o
main.d
do_irq.S
do_irq.o
do_irq.d

fresh make

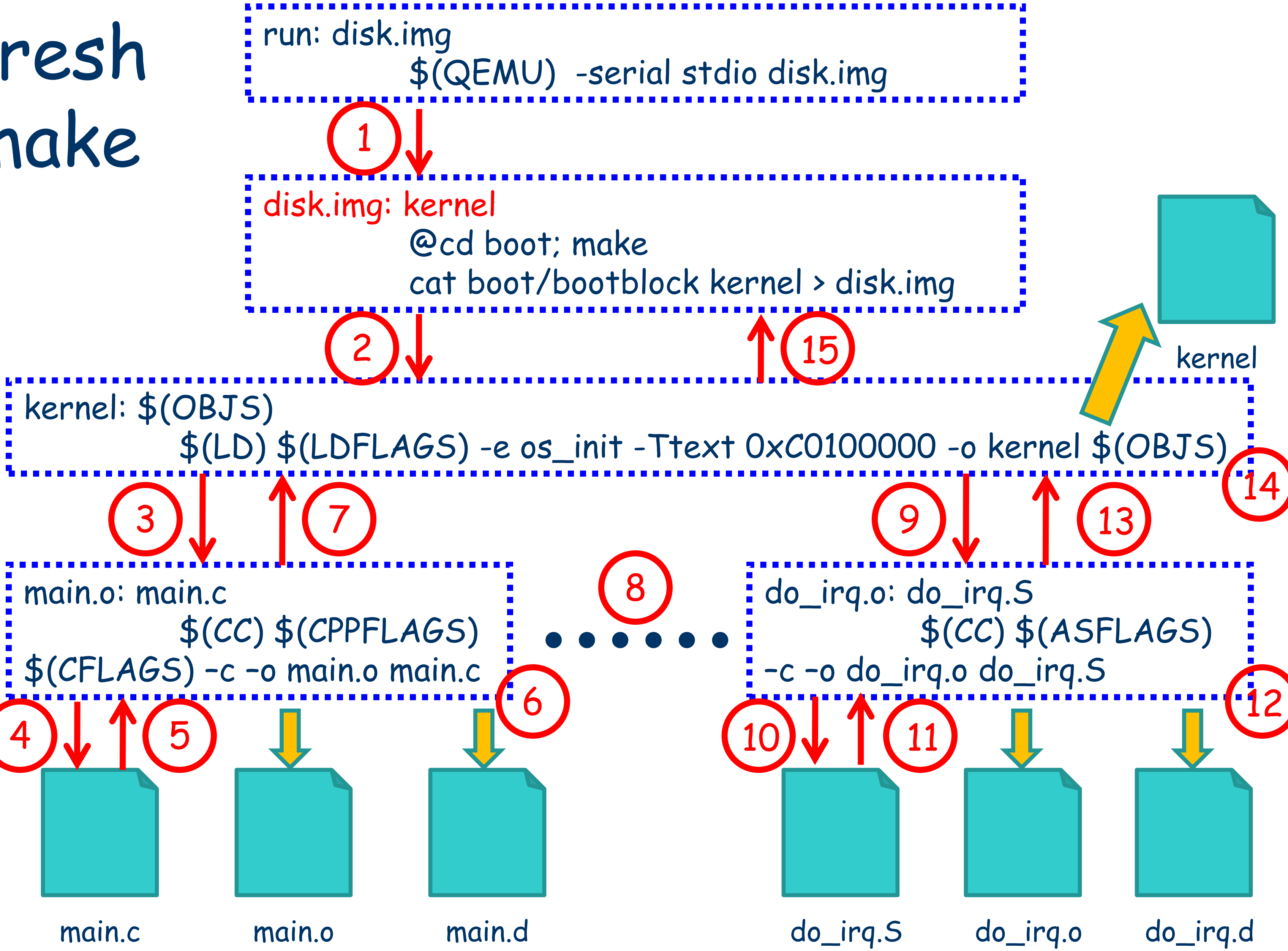


run: disk.img
$(QEMU) -serial stdio disk.img
1
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
2
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
3
7
9
13
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
8
do_irq.o: do_irq.S
$(CC) $(ASFLAGS)
-c -o do_irq.o do_irq.S
4
5
6
10
11
12
main.c
main.o
main.d
do_irq.S
do_irq.o
do_irq.d

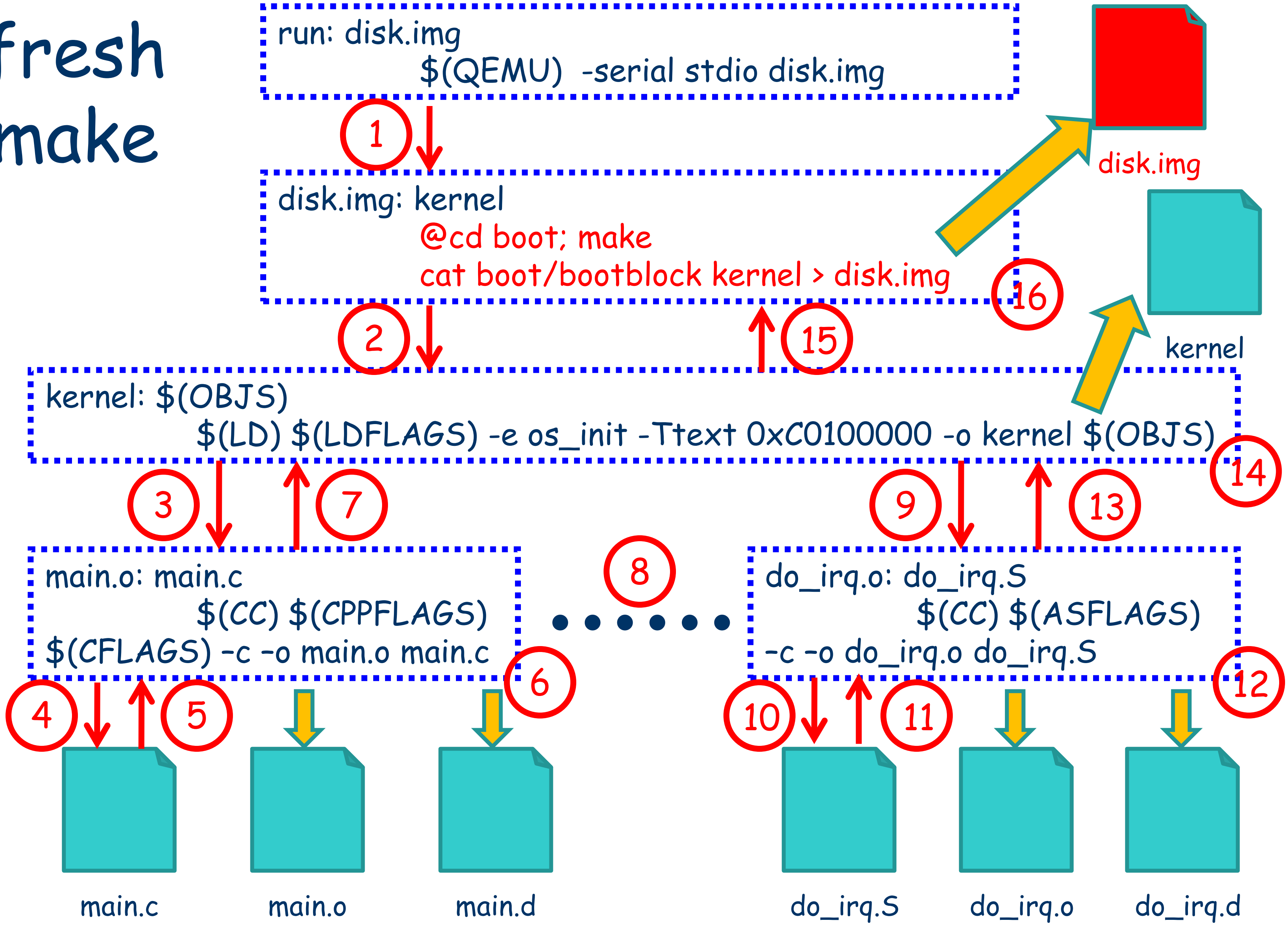
fresh make


run: disk.img
$(QEMU) -serial stdio disk.img
1
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
2
kernel
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
3
7
9
13
14
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
8
do_irq.o: do_irq.S
$(CC) $(ASFLAGS)
-c -o do_irq.o do_irq.S
4
5
6
10
11
12
main.c
main.o
main.d
do_irq.S
do_irq.o
do_irq.d

fresh make

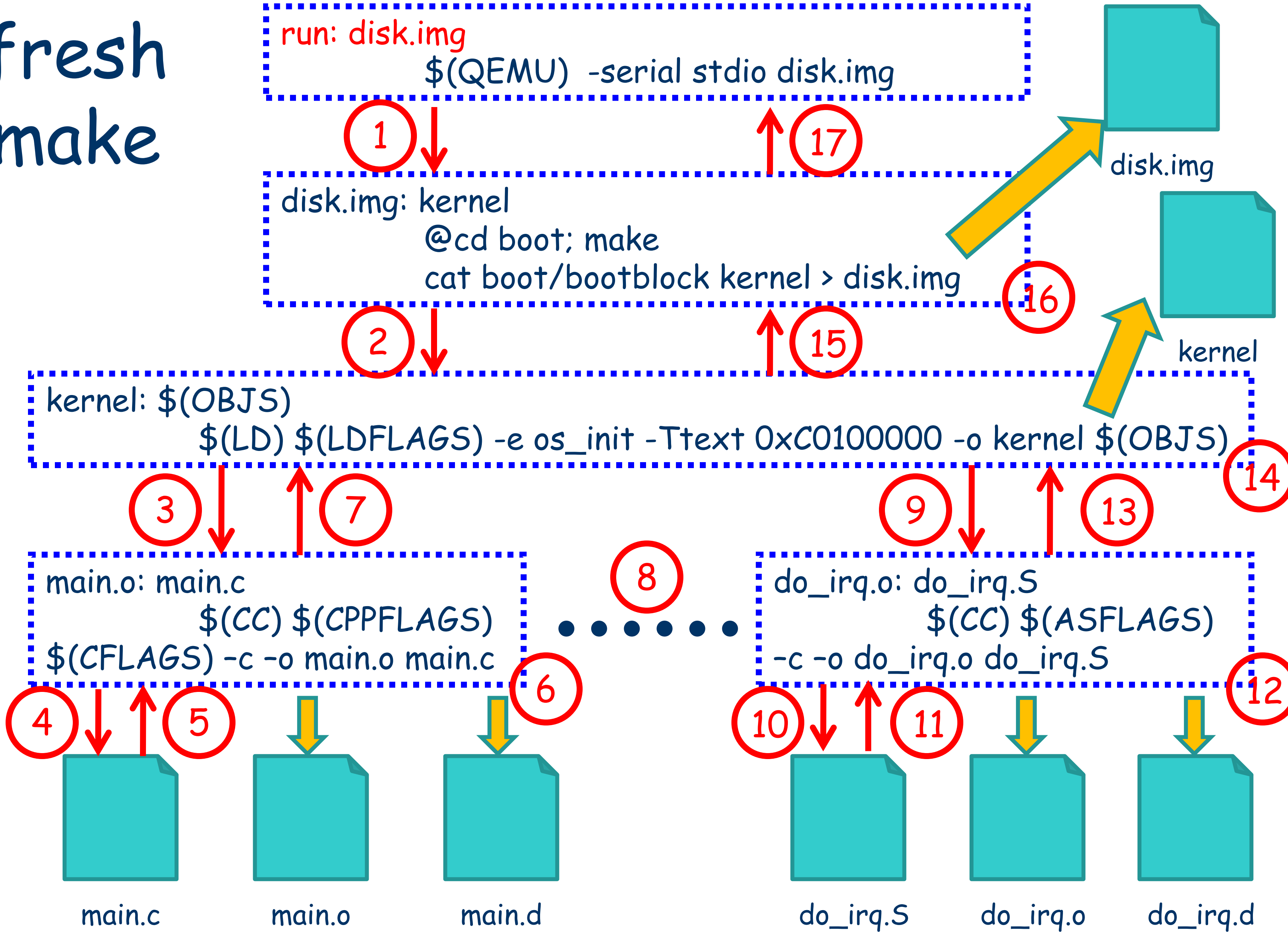

run: disk.img
$(QEMU) -serial stdio disk.img
1
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
2
15
kernel
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
14
3
7
9
13
8
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
do_irq.o: do_irq.S
$(CC) $(ASFLAGS)
-c -o do_irq.o do_irq.S
4
5
6
10
11
12
main.c
main.o
main.d
do_irq.S
do_irq.o
do_irq.d

fresh make



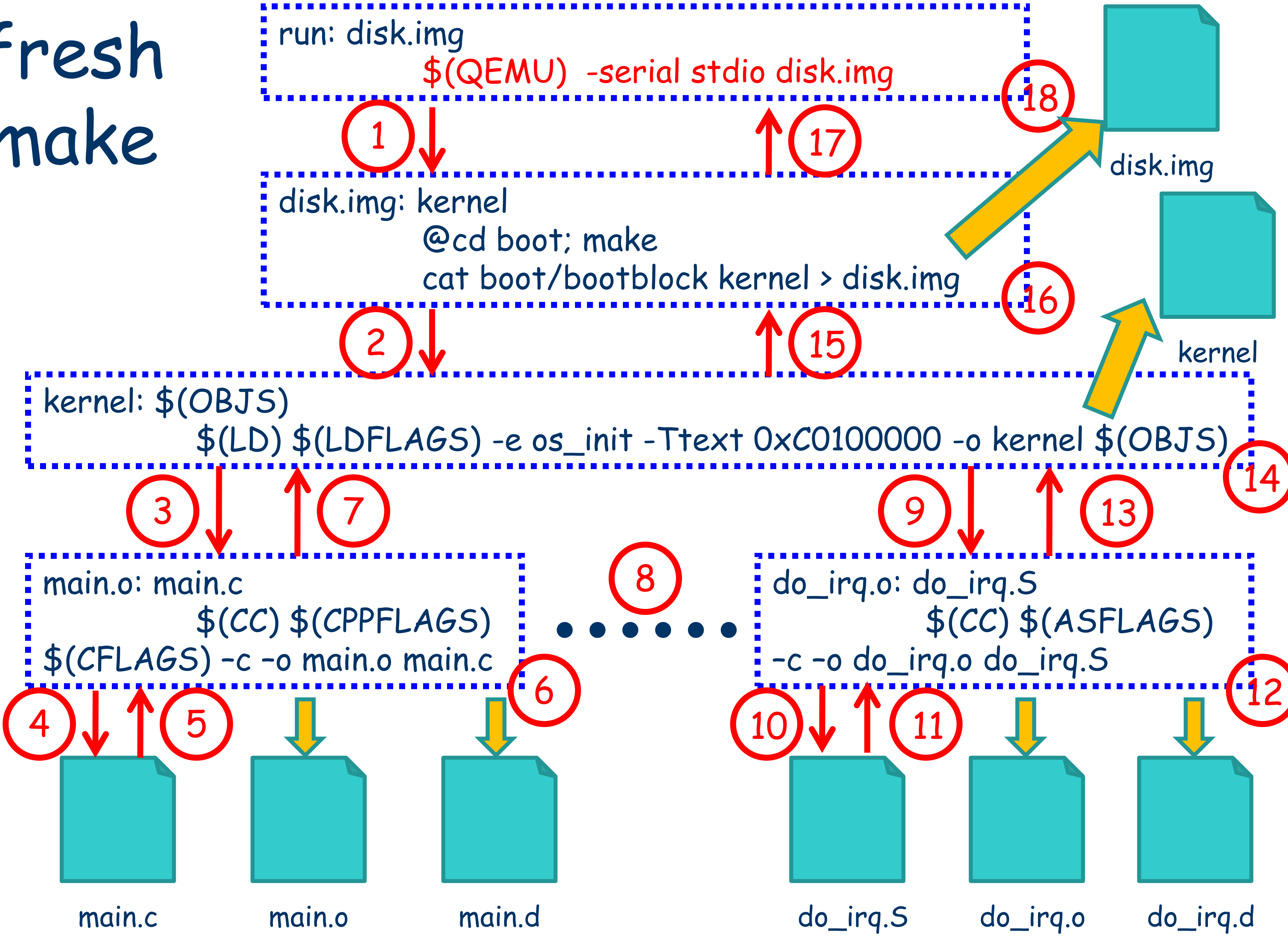
run: disk.img
$(QEMU) -serial stdio disk.img
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
do_irq.o: do_irq.S
$(CC) $(ASFLAGS)
-c -o do_irq.o do_irq.S
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
disk.img
kernel
main.c
main.o
main.d
do_irq.S
do_irq.o
do_irq.d

fresh make



run: disk.img
$(QEMU) -serial stdio disk.img
disk.img
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
kernel
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
do_irq.o: do_irq.S
$(CC) $(ASFLAGS)
-c -o do_irq.o do_irq.S
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
main.c
main.o
main.d
do_irq.S
do_irq.o
do_irq.d

fresh make



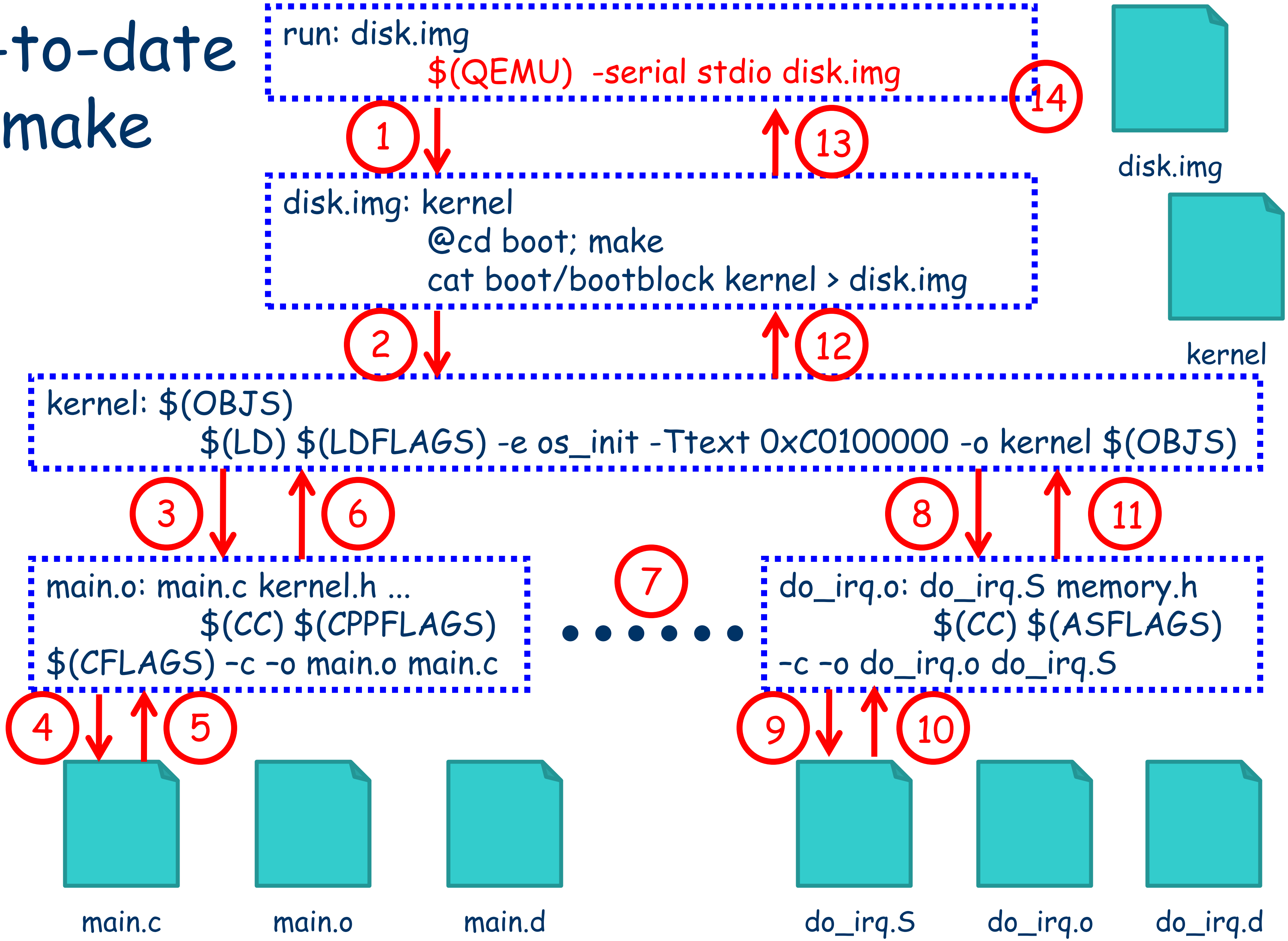
run: disk.img
$(QEMU) -serial stdio disk.img
disk.img: kernel
@cd boot; make
cat boot/bootblock kernel > disk.img
kernel: $(OBJS)
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
main.o: main.c
$(CC) $(CPPFLAGS)
$(CFLAGS) -c -o main.o main.c
do_irq.o: do_irq.S
$(CC) $(ASFLAGS)
-c -o do_irq.o do_irq.S
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
disk.img
kernel
main.c
main.o
main.d
do_irq.S
do_irq.o
do_irq.d

Makefile

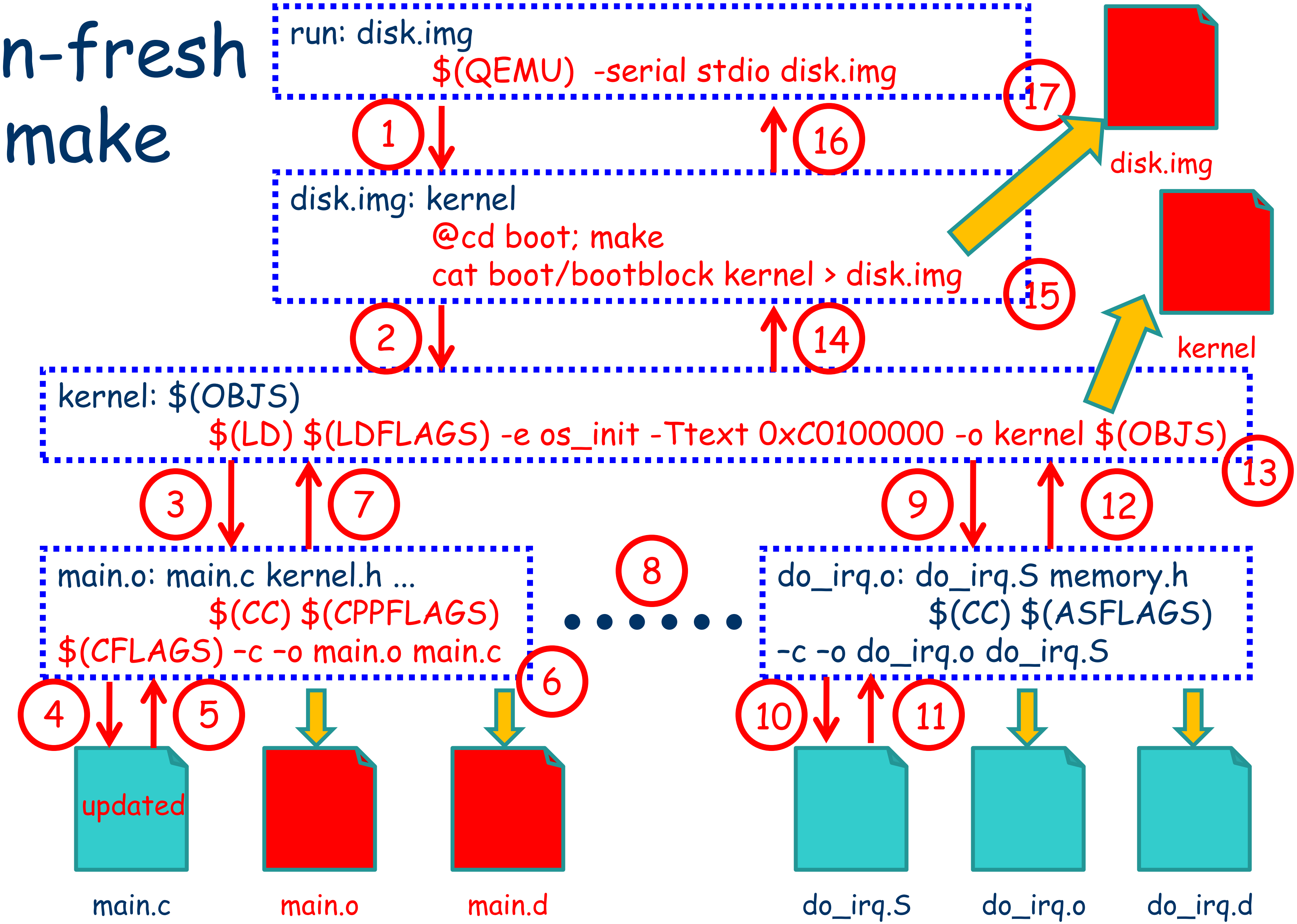
- What happen when
  - no file is updated?
  - a source file is updated?
  - a header file is updated?
  - boot/main.c is updated?

- Why "$(QEMU) -serial stdio disk.img" always executes?

up-to-date make



non-fresh make



volatile

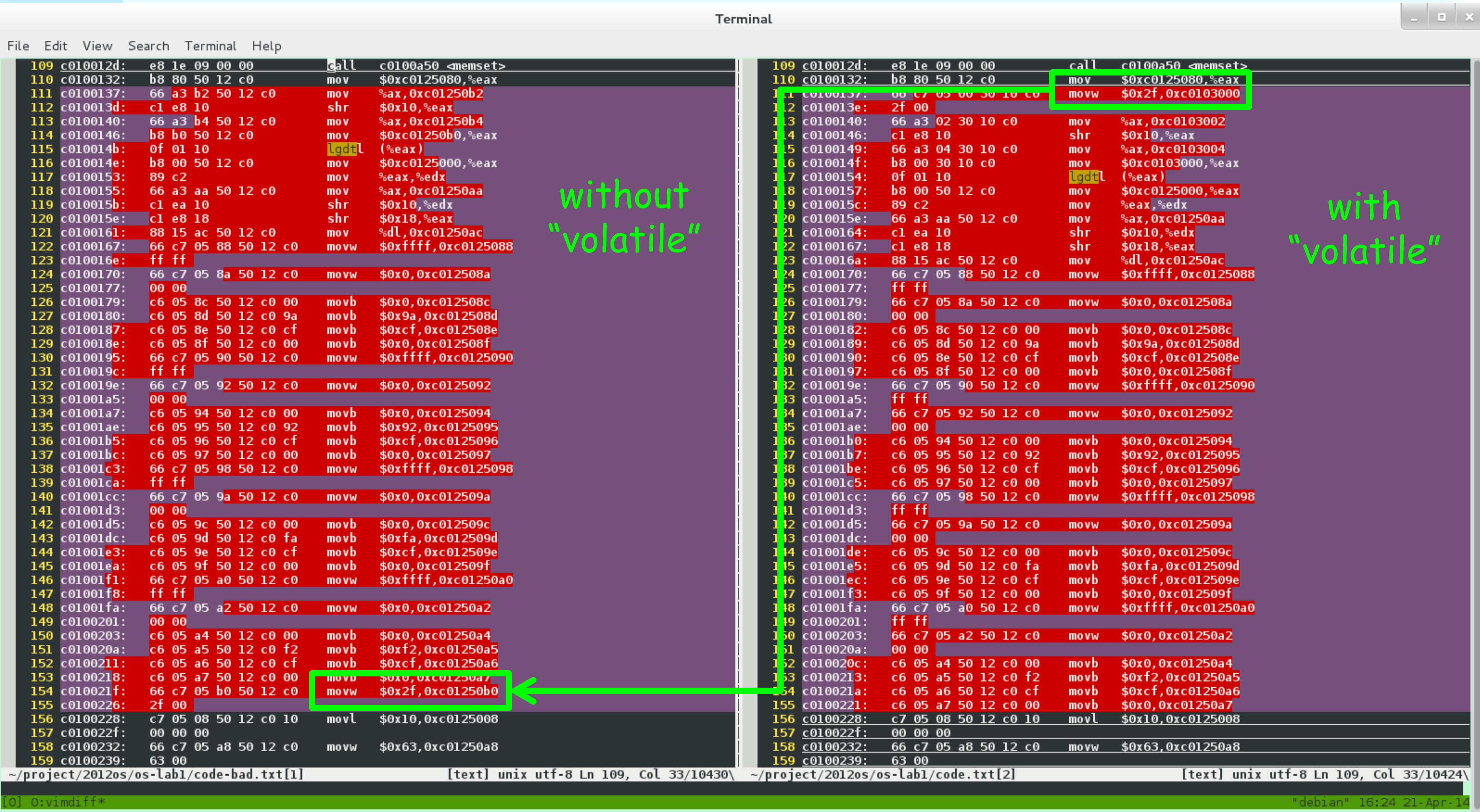
7. **<u>理解volatile</u>** 在include/x86/cpu.h中有两个函数write_gdtr()和write_idtr()，请选择其中一个函数来回答下列问题：函数体中有一行

```
static volatile uint16_t data[3];
```

尝试去掉volatile后重新编译并运行，你发现了什么问题？请对比去掉volatile前后编译结果的不同，并解释为什么去掉volatile之后会导致运行出现问题。（Hint：使用objdump命令，你可能还需要查阅i386手册。如果你使用gcc 4.7.3的版本，你可能不会观察到运行之后出现的问题，但你仍然可以对比编译结果的不同。）

- mysterious reboot
  - may not occur with gcc version later than 4.7.2

volatile

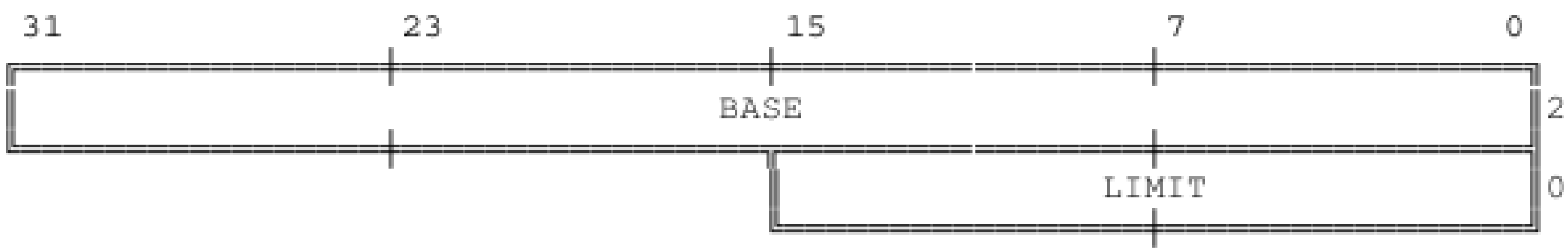


Terminal
File Edit View Search Terminal Help
109 c010012d: e8 1e 09 00 00 call c0100a50 <memset>
110 c0100132: b8 80 50 12 c0 mov $0xc0125080,%eax
111 c0100137: 66 a3 b2 50 12 c0 mov %ax,0xc01250b2
112 c010013d: c1 e8 10 shr $0x10,%eax
113 c0100140: 66 a3 b4 50 12 c0 mov %ax,0xc01250b4
114 c0100146: b8 b0 50 12 c0 mov $0xc01250b0,%eax
115 c010014b: 0f 01 10 lgdtl (%eax)
116 c010014e: b8 00 50 12 c0 mov $0xc0125000,%eax
117 c0100153: 89 c2 mov %eax,%edx
118 c0100155: 66 a3 aa 50 12 c0 mov %ax,0xc01250aa
119 c010015b: c1 ea 10 shr $0x10,%edx
120 c010015e: c1 e8 18 shr $0x18,%eax
121 c0100161: 88 15 ac 50 12 c0 mov %dl,0xc01250ac
122 c0100167: 66 c7 05 88 50 12 c0 movw $0xffff,0xc0125088
123 c010016e: ff ff
124 c0100170: 66 c7 05 8a 50 12 c0 movw $0x0,0xc012508a
125 c0100177: 00 00
126 c0100179: c6 05 8c 50 12 c0 00 movb $0x0,0xc012508c
127 c0100180: c6 05 8d 50 12 c0 9a movb $0x9a,0xc012508d
128 c0100187: c6 05 8e 50 12 c0 cf movb $0xcf,0xc012508e
129 c010018e: c6 05 8f 50 12 c0 00 movb $0x0,0xc012508f
130 c0100195: 66 c7 05 90 50 12 c0 movw $0xffff,0xc0125090
131 c010019c: ff ff
132 c010019e: 66 c7 05 92 50 12 c0 movw $0x0,0xc0125092
133 c01001a5: 00 00
134 c01001a7: c6 05 94 50 12 c0 00 movb $0x0,0xc0125094
135 c01001ae: c6 05 95 50 12 c0 92 movb $0x92,0xc0125095
136 c01001b5: c6 05 96 50 12 c0 cf movb $0xcf,0xc0125096
137 c01001bc: c6 05 97 50 12 c0 00 movb $0x0,0xc0125097
138 c01001c3: 66 c7 05 98 50 12 c0 movw $0xffff,0xc0125098
139 c01001ca: ff ff
140 c01001cc: 66 c7 05 9a 50 12 c0 movw $0x0,0xc012509a
141 c01001d3: 00 00
142 c01001d5: c6 05 9c 50 12 c0 00 movb $0x0,0xc012509c
143 c01001dc: c6 05 9d 50 12 c0 fa movb $0xfa,0xc012509d
144 c01001e3: c6 05 9e 50 12 c0 cf movb $0xcf,0xc012509e
145 c01001ea: c6 05 9f 50 12 c0 00 movb $0x0,0xc012509f
146 c01001f1: 66 c7 05 a0 50 12 c0 movw $0xffff,0xc01250a0
147 c01001f8: ff ff
148 c01001fa: 66 c7 05 a2 50 12 c0 movw $0x0,0xc01250a2
149 c0100201: 00 00
150 c0100203: c6 05 a4 50 12 c0 00 movb $0x0,0xc01250a4
151 c010020a: c6 05 a5 50 12 c0 f2 movb $0xf2,0xc01250a5
152 c0100211: c6 05 a6 50 12 c0 cf movb $0xcf,0xc01250a6
153 c0100218: c6 05 a7 50 12 c0 00
154 c010021f: 66 c7 05 b0 50 12 c0 movw $0x2f,0xc01250b0
155 c0100226: 2f 00
156 c0100228: c7 05 08 50 12 c0 10 movl $0x10,0xc0125008
157 c010022f: 00 00 00
158 c0100232: 66 c7 05 a8 50 12 c0 movw $0x63,0xc01250a8
159 c0100239: 63 00
~/project/2012os/os-lab1/code-bad.txt[1] [text] unix utf-8 Ln 109, Col 33/10430\
without "volatile"
109 c010012d: e8 1e 09 00 00 call c0100a50 <memset>
110 c0100132: b8 80 50 12 c0 mov $0xc0125080,%eax
movw $0x2f,0xc0103000
c010013e: 2f 00
c0100140: 66 a3 02 30 10 c0 mov %ax,0xc0103002
c0100146: c1 e8 10 shr $0x10,%eax
c0100149: 66 a3 04 30 10 c0 mov %ax,0xc0103004
c010014f: b8 00 30 10 c0 mov $0xc0103000,%eax
c0100154: 0f 01 10 lgdtl (%eax)
c0100157: b8 00 50 12 c0 mov $0xc0125000,%eax
c010015c: 89 c2 mov %eax,%edx
c010015e: 66 a3 aa 50 12 c0 mov %ax,0xc01250aa
c0100164: c1 ea 10 shr $0x10,%edx
c0100167: c1 e8 18 shr $0x18,%eax
c010016a: 88 15 ac 50 12 c0 mov %dl,0xc01250ac
c0100170: 66 c7 05 88 50 12 c0 movw $0xffff,0xc0125088
c0100177: ff ff
c0100179: 66 c7 05 8a 50 12 c0 movw $0x0,0xc012508a
c0100180: 00 00
c0100182: c6 05 8c 50 12 c0 00 movb $0x0,0xc012508c
c0100189: c6 05 8d 50 12 c0 9a movb $0x9a,0xc012508d
c0100190: c6 05 8e 50 12 c0 cf movb $0xcf,0xc012508e
c0100197: c6 05 8f 50 12 c0 00 movb $0x0,0xc012508f
c010019e: 66 c7 05 90 50 12 c0 movw $0xffff,0xc0125090
c01001a5: ff ff
c01001a7: 66 c7 05 92 50 12 c0 movw $0x0,0xc0125092
c01001ae: 00 00
c01001b0: c6 05 94 50 12 c0 00 movb $0x0,0xc0125094
c01001b7: c6 05 95 50 12 c0 92 movb $0x92,0xc0125095
c01001be: c6 05 96 50 12 c0 cf movb $0xcf,0xc0125096
c01001c5: c6 05 97 50 12 c0 00 movb $0x0,0xc0125097
c01001cc: 66 c7 05 98 50 12 c0 movw $0xffff,0xc0125098
c01001d3: ff ff
c01001d5: 66 c7 05 9a 50 12 c0 movw $0x0,0xc012509a
c01001dc: 00 00
c01001de: c6 05 9c 50 12 c0 00 movb $0x0,0xc012509c
c01001e5: c6 05 9d 50 12 c0 fa movb $0xfa,0xc012509d
c01001ec: c6 05 9e 50 12 c0 cf movb $0xcf,0xc012509e
c01001f3: c6 05 9f 50 12 c0 00 movb $0x0,0xc012509f
c01001fa: 66 c7 05 a0 50 12 c0 movw $0xffff,0xc01250a0
c0100201: ff ff
c0100203: 66 c7 05 a2 50 12 c0 movw $0x0,0xc01250a2
c010020a: 00 00
c010020c: c6 05 a4 50 12 c0 00 movb $0x0,0xc01250a4
c0100213: c6 05 a5 50 12 c0 f2 movb $0xf2,0xc01250a5
c010021a: c6 05 a6 50 12 c0 cf movb $0xcf,0xc01250a6
c0100221: c6 05 a7 50 12 c0 00 movb $0x0,0xc01250a7
156 c0100228: c7 05 08 50 12 c0 10 movl $0x10,0xc0125008
157 c010022f: 00 00 00
158 c0100232: 66 c7 05 a8 50 12 c0 movw $0x63,0xc01250a8
159 c0100239: 63 00
~/project/2012os/os-lab1/code.txt[2] [text] unix utf-8 Ln 109, Col 33/10424\
with "volatile"
[0] 0:vimdiff* "debian" 16:24 21-Apr-14

volatile

- Why mysterious reboot?
- What is data[0] for GDTR/IDTR?
  - P.156 in "i386 manual"

Figure 9-2. Pseudo-Descriptor Format for LIDT and SIDT



Loading kernel

8. **加载内核** 在boot/main.c的bootmain()函数中，最后计算出entry的值. 这个值是多少? 尝试使用不同的方法获取这个值.

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0xc01002c0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          133568 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         17
  Section header string table index: 14
```

- gdb
- readelf
  - How does entry point generate?

Loading kernel

9. <u>**加载内核**</u> 在boot/main.c的bootmain()函数中，有两处代码需要减去KOFFSET的值(分别在计算pa和entry时)，但在Lab0中相应代码并没有减去KOFFSET. 请尝试去掉这两处减去KOFFSET的操作，然后重新编译并运行，你发现了什么问题？请解释为什么在Lab1中需要减去KOFFSET的操作.

```
$(LD) $(LDFLAGS) -e os_init -Ttext 0xC0100000 -o kernel $(OBJS)
```

- kernel thinks it is located at 0xc0100000
  - makes the kernel mapping identical to the one in Linux
    - why?
  - but MBR loads the kernel at 0x100000
- All addresses in the kernel binary are virtual addresses.

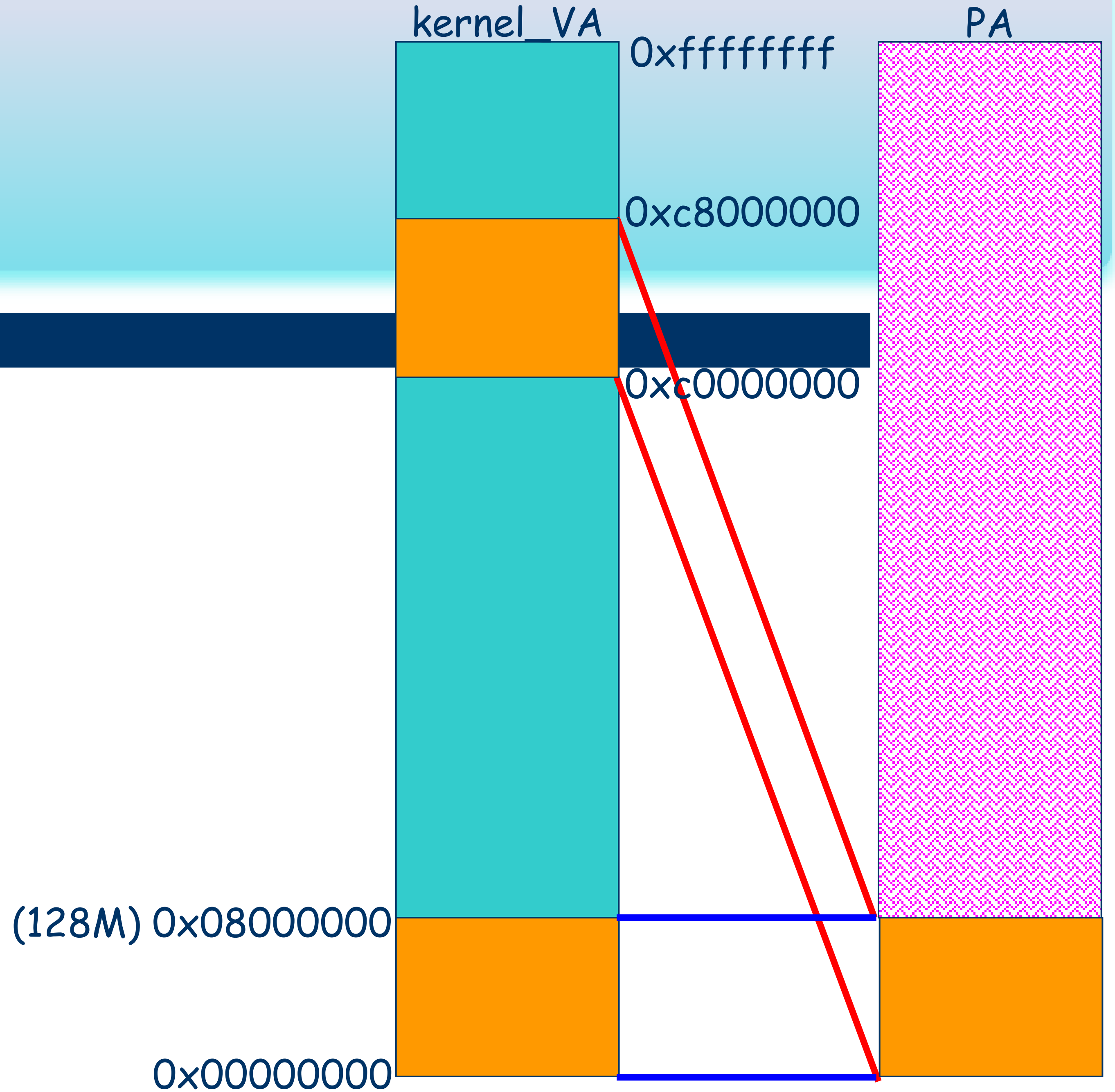
Loading kernel

- What about…

```
for(; ph < eph; ph ++) {
-               pa = (unsigned char*)(ph->paddr  - KOFFSET); /* physical address */
+               pa = (unsigned char*)(ph->paddr ); /* physical address */
                readseg(pa, ph->filesz, ph->off); /* load from disk */
                for (i = pa + ph->filesz; i < pa + ph->memsz; *i ++ = 0);
}
/* Here we go! */
entry = (void(*)(void))(elf->entry - KOFFSET);
```

```
for(; ph < eph; ph ++) {
                pa = (unsigned char*)(ph->paddr - KOFFSET); /* physical address */
                readseg(pa, ph->filesz, ph->off); /* load from disk */
                for (i = pa + ph->filesz; i < pa + ph->memsz; *i ++ = 0);
}
/* Here we go! */
-entry = (void(*)(void))(elf->entry - KOFFSET);
+entry = (void(*)(void))(elf->entry);
```

Paging



kernel_VA
PA
0xffffffff
0xc8000000
0xc0000000
(128M) 0x08000000
0x00000000

Paging

11. **分页机制** 在src/kernel/main.c的os_init()函数中有一处注释"Before setting up correct paging, no global variable can be used"。尝试在main.c中定义一个全局变量

```
volatile int x = 0;
```

然后在调用init_page()前使用这个全局变量

```
x = 10000;
```

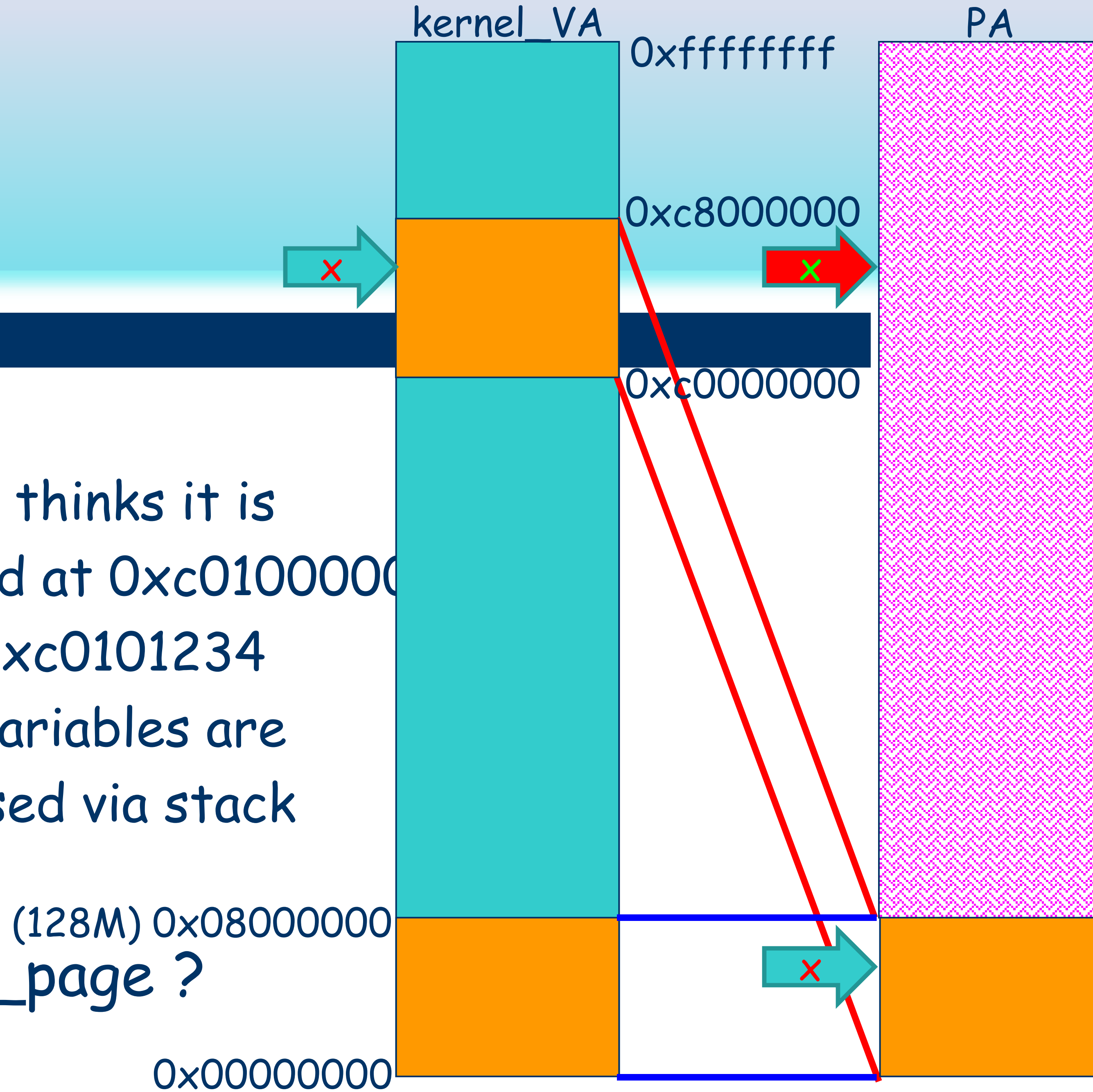
并在串口初始化结束后输出它的值

```
printk("x = %d\n", x);
```

重新编译并运行，你发现了什么问题？请解释为什么在开启分页之前不能使用全局变量，但却可以使用局部变量(在init_page()函数中使用了局部变量)。细心的你会发现，在开启分页机制之前，init_page()中仍然使用了一些全局变量，但却没有造成错误，这又是为什么？(回答完本题后可以删除添加的代码。)

Paging

- "x = 0"
  - kernel thinks it is located at 0xc0100000
  - &x = 0xc0101234
  - local variables are accessed via stack
- call init_page ?



- Lab1 feedback
  - 蓝框题

Initialize current state

- the initial value of esp in GPR can by arbitrary
  - why?
- P.364 in "i386 manual"

```
ELSE (* OperandSize = 32, instruction = POPAD *)
    EDI ← Pop();
    ESI ← Pop();
    EBP ← Pop();
    throwaway ← Pop (); (* Skip ESP *)
    EBX ← Pop();
    EDX ← Pop();
    ECX ← Pop();
    EAX ← Pop();
FI;
```

Hardware context switch

硬件实现的上下文切换
事实上，上下文切换分为两种，分别由硬件和软件实现. Nanos，Windows和Linux都是使用软件实现的上下文切换. 请搜索硬件实现上下文切换的相关信息，思考一下两者之间有什么不同，为何现代操作系统大多数都采用软件实现的方式?

- http://stackoverflow.com/questions/2711044/why-doesnt-linux-use-the-hardware-context-switch-via-the-tss

PCB definition

另类的PCB(这个问题有难度)

有一种PCB的定义如下:

```
#define KSTACK_SIZE 4096
union PCB {
        uint8_t kstack[KSTACK_SIZE];
        struct {
                void *tf;
                // other fields
        };
};
```

这样的定义方式有什么好处? 在SMP(对称多处理器)的环境下, 就必须采取类似这样的方式来定义PCB, 你知道为什么吗?

PCB definition

- Any other way to obtain "current"?
  - %esp & 0xfffff000
- For SMP
  - suppose we define "PCB* current[NR_CPU]"
  - How does one core obtain its "current"?

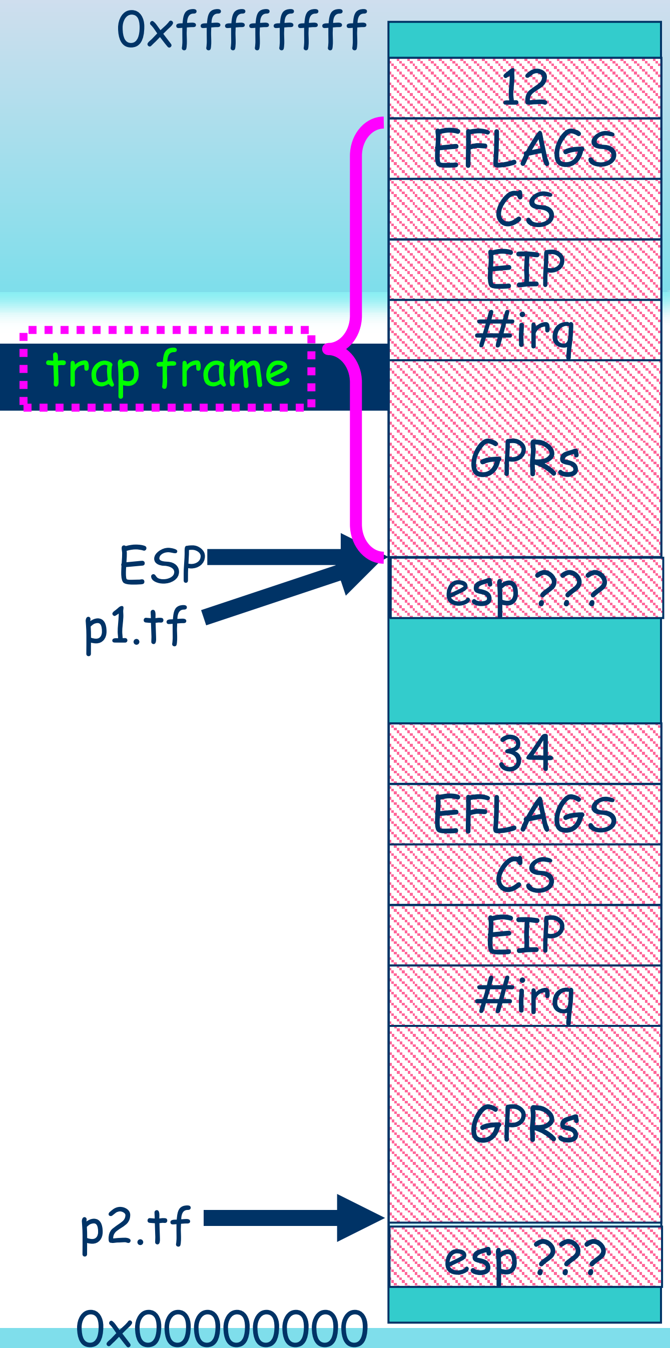
Context switch

对上下文切换过程的思考

- 在asm_do_irq调用irq_handle之前，有一条指令保存了当前栈顶指针的值。这条指令有什么目的？如果将其去掉，会有什么影响？
- 在你完成堆栈切换后，在src/kernel/irq/do_irq.S中有一条指令必须去掉，否则会发生错误。思考一下为什么在完成堆栈切换之前需要保留该指令，完成堆栈切换之后却必须将其去掉？该指令本来想干什么？现在其作用是否在哪里实现了？如果你觉得很晕，你可以用纸笔画出堆栈的变化，人工模拟上下文切换过程。

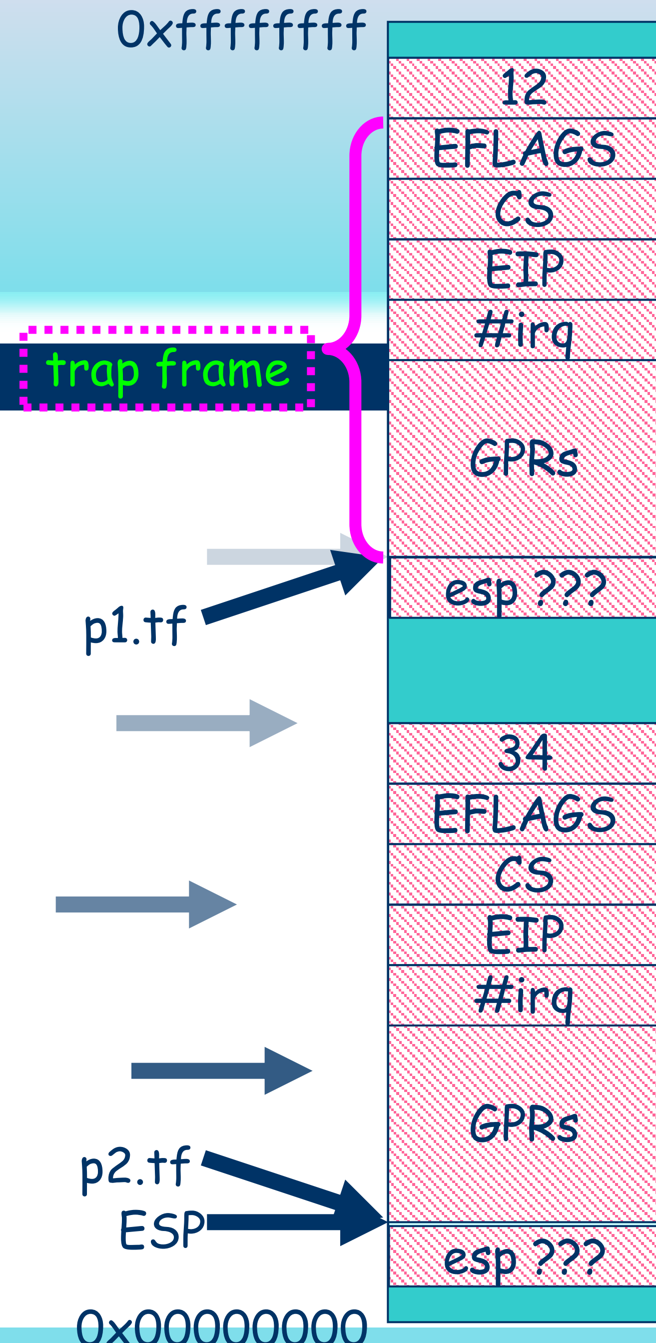
Context switch

```
    movw $SELECTOR_KERNEL(SEG_KERNEL_DATA), %ax
    movw %ax, %ds
    movw %ax, %es

    pushl %esp
    call irq_handle

# YOU NEED TO SWITCH STACK TO current->tf
# SO YOU NEED TWO ADD TWO LINES OF INTERRUPT CODE
# HINT:
#     1. USE movl INSTRUCTION
#     2. USE (address) CAN REFERENCE MEMORY LOCATION
#     3. YOU MAY FLUSH ANY GENRAL PURPOSE REGISTER AS
#     4. REGISTERS ARE REFERENCED BY "%", SUCH AS %es

################### your work ###################


#################################################
    #addl $4, %esp  #when you finish this task, this

    popal
```



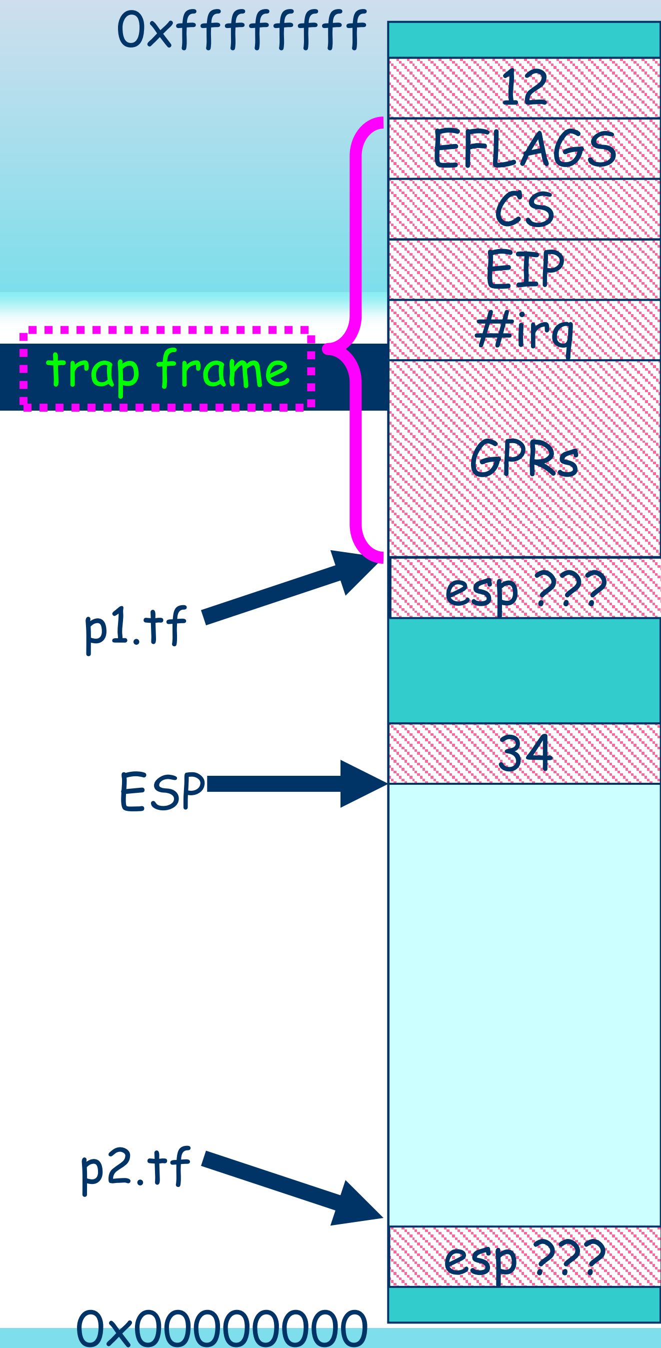
Context switch

```
    movw $SELECTOR_KERNEL(SEG_KERNEL_DATA), %ax
    movw %ax, %ds
    movw %ax, %es

    pushl %esp
    call irq_handle

# YOU NEED TO SWITCH STACK TO current->tf
# SO YOU NEED TWO ADD TWO LINES OF INTERRUPT CODE
# HINT:
#     1. USE movl INSTRUCTION
#     2. USE (address) CAN REFERENCE MEMORY LOCATION
#     3. YOU MAY FLUSH ANY GENRAL PURPOSE REGISTER AS
#     4. REGISTERS ARE REFERENCED BY "%", SUCH AS %es

################### your work ###################


#################################################
    #addl $4, %esp  #when you finish this task, this

    popal
```

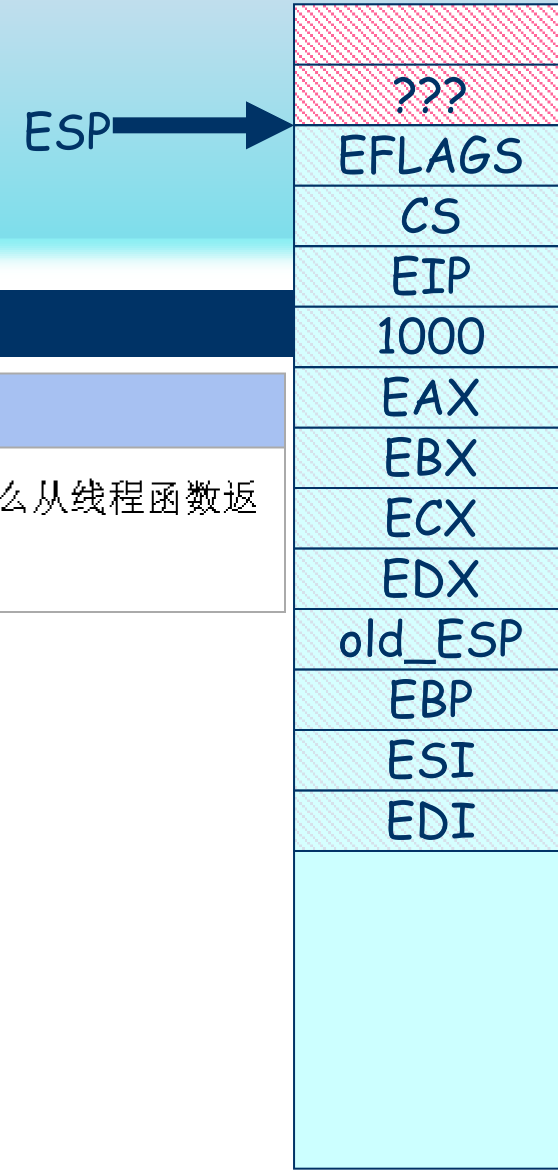


Context switch

```
    movw $SELECTOR_KERNEL(SEG_KERNEL_DATA), %ax
    movw %ax, %ds
    movw %ax, %es

    pushl %esp
    call irq_handle

# YOU NEED TO SWITCH STACK TO current->tf
# SO YOU NEED TWO ADD TWO LINES OF INTERRUPT CODE
# HINT:
#     1. USE movl INSTRUCTION
#     2. USE (address) CAN REFERENCE MEMORY LOCATION
#     3. YOU MAY FLUSH ANY GENRAL PURPOSE REGISTER AS
#     4. REGISTERS ARE REFERENCED BY "%", SUCH AS %es

################### your work ###################


#################################################
    #addl $4, %esp  #when you finish this task, this

    popal
```



Never-return function



不能返回的函数

目前你必须保证线程函数永远不会返回，否则将会发生错误. 为什么从线程函数返回就会发生错误?

Use printk in do_irq.S

在汇编代码中调用printk
你可能需要在asm_do_irq中调用printk来帮助你进行调试. 想一想怎么在汇编代码中调用它, 并尝试付诸实践.

```
.extern  printk
msg:
.asciz "esp = %x\n"

pushl %esp
pushl $msg
call printk
addl $8, %esp
```

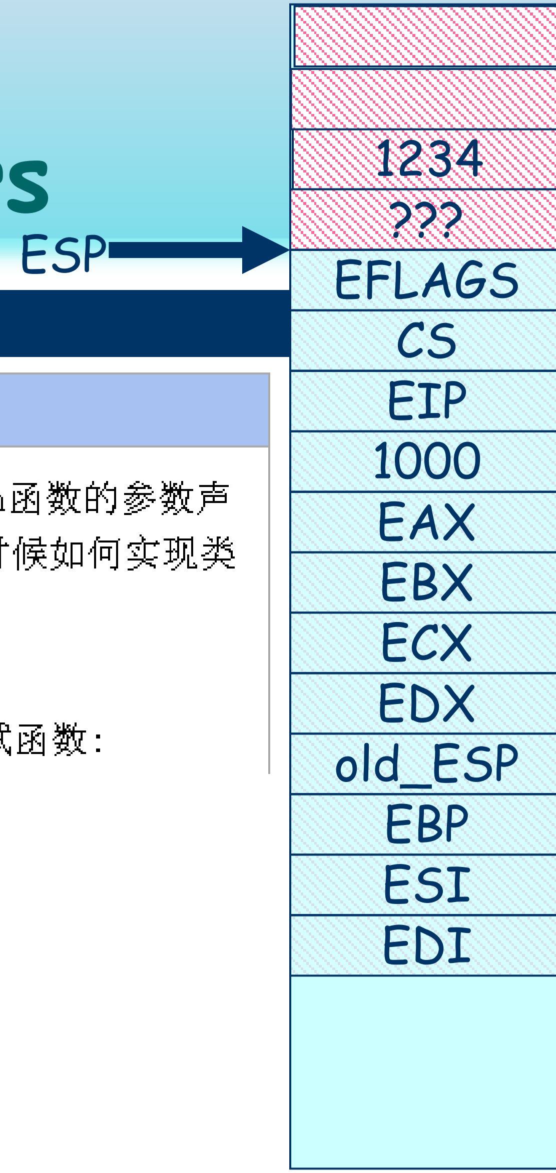
Stack overflow

```
void stackoverflow(int x)
{
    if(x==0)
        printk("%d ",x);
    if(x>0)
        stackoverflow(x-1);
}
void keep_stackoverflow()
{
    while(1){stackoverflow(16384*1000);}
}
```

Stack overflow

```
c01001f0 <stackoverflow>:
c01001f0:   55                              push   %ebp
c01001f1:   89 e5                           mov    %esp,%ebp
c01001f3:   83 ec 18                        sub    $0x18,%esp
c01001f6:   8b 45 08                        mov    0x8(%ebp),%eax
c01001f9:   83 f8 00                        cmp    $0x0,%eax
c01001fc:   74 07                           je     c0100205 <stackoverflow+0x15>
c01001fe:   7e 19                           jle    c0100219 <stackoverflow+0x29>
c0100200:   83 e8 01                        sub    $0x1,%eax
c0100203:   75 fb                           jne    c0100200 <stackoverflow+0x10>
c0100205:   c7 44 24 04 00 00 00            movl   $0x0,0x4(%esp)
c010020c:   00
c010020d:   c7 04 24 2c 2b 10 c0            movl   $0xc0102b2c,(%esp)
c0100214:   e8 d7 08 00 00                  call   c0100af0 <printk>
c0100219:   c9                              leave
c010021a:   c3                              ret
c010021b:   90                              nop
c010021c:   8d 74 26 00                     lea    0x0(%esi,%eiz,1),%esi
```

Threads with parameters



带有参数的线程(有些难度)

我们知道在Linux下可以编写从外部读入参数的程序，只需要把main函数的参数声明改为int main(int argc, char *argv[])即可。在创建线程的时候如何实现类似的功能?

先来个简单一点的吧，让create_kthread多接受一个整型参数:
create_kthread(void *fun, int arg)。然后只需要编写一个测试函数:

- Hints for implementing locking

Locking

- upgraded version of cli() & sti() to solve
  - nested locking
  - sleep during locking
  - locking in interrupt
- Have you triggered assertion fail/mysterious reboot when testing your implementation?

Where does the problem come from?

- Interrupt is not enabled when it should.
- Interrupt is enabled when it should not.

- How to find these bugs?

Assertion

- check the status of IF bit in EFLAGS

```
#define INTR assert(read_eflags() & IF_MASK)
#define NOINTR assert(~read_eflags() & IF_MASK)
```

- insert them in your code
- consistency
  - NOINTR when in critical region or during interrupt
  - INTR otherwise

Trap 1 – nested locking

```
void V(Sem *s) {
    INTR;
    lock(); NOINTR;
    // ...
    wakeup(p); NOINTR;
    // ...
    unlock();
    INTR;
}
```

not safe
any longer

```
void wakeup(PCB *p) {
    lock(); NOINTR;
    // ...
    unlock();
}
```

Trap 2 – sleep during locking

```
void P(Sem *s) {
    INTR;
    lock(); NOINTR;
    // ...
    if(counter == 0) { sleep(); NOINTR; }
    // ...
    unlock();
    INTR;
}
```

the consistency of locking should not be violated by other processes

Trap 3 – use locking in interrupts

```
void timer_handler() {
    // ...
    NOINTR;
    V(sem);
    NOINTR;
    // ...
}
```

```
void V(Sem *s) {
    lock(); NOINTR;
    // ...
    unlock();
}
```

should not sti()

Test case

```
void
test_consumer(void) {
    while (1) {
        P(&mutex); INTR;
        P(&full); INTR;
        // ...
        V(&empty); INTR;
        V(&mutex); INTR;
    }
}
```

Implementation

- 1. nested locking
- 2. sleep in locking
- 3. locking in interrupt

- (1) ➔ locking counter
  - assert(lock_cnt >= 0);
- (2) ➔ counter per thread
- (3) ➔ store IF before the first locking, restore it when leaving the most outside critical region

Lab2 is out!

- the second stage
  - implement message passing
  - create 4 kernel thread to print “abcdabcd…”
    - communicate with message passing
- the third stage
  - add device drivers
    - test your message passing
  - to be continue…
- Have fun!