1 系统调用

虽然在第一次报告中提到过系统调用，但是由于当时自己了解地不够全面，所以这次重新完整地学习系统调用。

1.1 系统调用的过程

用户空间的程序无法直接执行内容代码，也就无法直接调用内核空间中的函数。系统通过以下方式实现系统调用。

- 应用程序以软中断的方式通知内核自己需要一个系统调用。

  在 x86 上，软中断的中断向量号为 128，所以可以通过 int $0x80 指令触发软中断。int $0x80 之后，系统切换到内核态，并执行第 128 号异常处理程序 system_call()。这个程序在 entry_64.S 文件中用汇编语言编写。

- 内核代表应用程序在内核空间执行系统调用。

  在陷入内核前，应用程序将相应的系统调用号放入 eax 中，当系统调用处理程序运行时，可能直接从 eax 中得到数据。在执行相应的系统调用前，需要验证系统调用号的有效性。实现代码如下。

```
1    cmpl $__NR_syscall_max, %eax
2    ja 1f ; 如果系统调用号大于__NR_syscall_max，则返回—ENOSYS
```

  如果系统调用号有效，就执行系统调用表中相应的系统调用。

```
1    ; %rax存放着系统调用号
2    ; 因为是64位系统，系统调用表中的表项是以8字节类型存放的，所以需要将给定的
         系统调用号乘以8
3    call *sys_call_table(,%rax,8)
```

上述过程只是讲述了内核空间如何执行系统调用。除此之外，还需要明白应用程序在用户空间是怎么以软中断的方式通知内核自己需要一个系统调用。

Linux 在 unistd.h 中提供了一组宏，用于直接对系统调用进行访问。这组宏可以设置好寄存器并调用陷入指令。实现代码如下。

```
1    #define _syscall0(type, name) \
2    type name(void) \
3    {\
4        long __res; \
5        __asm__ volatile(
6            "int $0x80" \
7            : "=a" (__res)
8            : "0" (__NR_##name)); \
9        if(__res >= 0) \
```

```
10              return (type) __res; \
11          errno = -__res; \
12          return -1;\
13      }
```

假设现在有一个系统调用 foo()，我们就可以用下面的方式在用户空间调用它。

```
1   __syscall0(long, foo)
2   // 这里的宏会展开为 long foo(){/*...*/}
3
4   int main()
5   {
6       // ...
7       foo();
8       //...
9       return 0;
10  }
```

需要注意的是，系统不只拥有 _syscall0() 这一个宏，它提供了一组宏，这组宏是 _syscalln()。这里的 n 代表了需要传递给系统调用的参数个数。宏所需要的参数个数是 2+2n，第一个参数是系统调用函数返回类型，第二个参数是系统调用名称，随后是系统调用每个参数的类型和名称。

1.2 注册系统调用的步骤

下面通过将 foo() 加入为系统调用来说明这个步骤。

- 首先，需要在系统调用表的最后加入一个表项。现在要把 foo() 注册为一个正式的系统调用，就要把 sys_foo 加入到系统调用表中。实现代码如下：

```
1   ENTRY(sys_call_table)
2       .long sys_restart_syscall
3       ;...
4       .long sys_perf_event_open ; 最后一项
5       ; 将sys_foo加到这个表的末尾
6       .long sys_foo
```

因为我对 linux 汇编还不是很熟悉，所以去查询了以下几条指令的意思：

```
1   ENTRY()
2   ; 这是一个宏，定义于linux-2.6.35.5/include/Linux/linkage.h。格式如下：
3   #define ENTRY(name) \
4   .globl name \
5   ALIGN \
6   name:
7
8   ; .globl symbol 的含义是：
9   ; 定义该symbol为global的，也就是其他文件可以访问并使用该symbol
10
11  ; .long val 的含义是：
```

```
12      ; 该指令在当前区定义一个32位长整数的常数。需要注意该指令无法在bss段中使用
```

- 接下来，将 foo() 的系统调用号加到 <arch/arm/include/uapi/asm/unistd.h> 中。实现代码如下：

```
1   #if defined(__thumb) || defined(__ARM_EABI)
2   #define __NR_SYSCALL_BASE 0
3   #else
4   #define __NR_SYSCALL_BASE __NR_OABI_SYSCALL_BASE
5   #endif
6
7   #define __NR_restart_syscall (__NR_SYSCALL_BASE + 0)
8   // ...
9   // 最后一个系统调用号
10  #define __NR_pkey_free (__NR_SYSCALL_BASE + 396)
11  // 在最后一行添加foo()的系统调用号
12  #define __NR_foo (__NR_SYSCALL_BASE + 397)
```

- 最后在内核代码中定义这个系统调用函数。

```
1   asmlinkage long sys_foo()
2   {
3       return THREAD_SIZE;
4   }
5
6   // asmlinkage在linkage.h中有定义
7   #ifdef CONFIG_X86_32
8   #define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
9   #endif
10  // __attribute__((regparm(0)))告诉编译器参数只能通过堆栈来传递
11  // X86里面的系统调用都是先将参数压入stack以后调用sys_*函数的，所以必须告诉
        编译器只能通过堆栈传递参数
```

1.3 学习系统调用后的反思

在前面几个小节，我学会了如何在操作系统中新建一个系统调用。这一开始让我兴奋，随后而来的是疑惑，因为我并不了解系统是如何支持系统调用这个行为的。我也不了解 int $0x80 之后，系统具体发生了什么。我觉得这还需要我学习了系统的中断机制和特权保护机制后，知道如何去实现它们后，才能真正地理解系统调用。

2 LDT 的实现

2.1 LDT 的定义

首先我们需要在 GDT 表中增加局部描述符表的描述符。

```
        [SECTION .gdt]
LABEL_GDT: Descriptor 0,LDT
; 添加的描述符
LABEL_DESC_LDT: Descriptor 0,LDTLen-1,DA_LDT
; 相应的选择符
SelectorLDT equ LABEL_DESC_LDT - LABEL_GDT
```

然后我们需要定义一个 LDT 表。LDT 表和 GDT 表其实很类似，我在其中定义了一个指向 CODEA 代码段的段描述符。

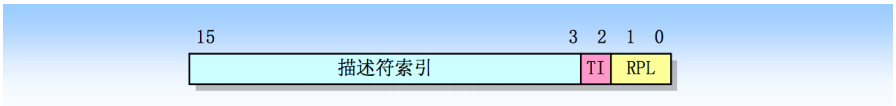
```
        ; 定义LDT表
        [SECTION .ldt]
        ALIGN 32
        LABEL_LDT:
        LABEL_LDT_DESC_CODEA: Descriptor 0,CodeALen-1,DA_C+DA_32
        LDTLEN equ $-LABEL_LDT

        SelectorLDTCodeA equ LABEL_LDT_DESC_CODEA-LABEL_LDT+4
        LDTLen equ $-LABEL_LDT

        ; 定义在LDT表中段描述符指向的代码段
        [SECTION .la]
        ALIGN 32
        [BITS 32]
        LABEL_CODE_A:
            mov ax,SelectorVideo
            mov gs,ax
            mov edi,(80*12+0)*2
            mov ah,0Ch
            mov al,'L'
            mov [gs:edi],ax
        CodeALen equ $-LABEL_CODE_A
```

上面代码段中，我们应该注意的是这个语句。

```
        SelectorLDTCodeA equ LABEL_LDT_DESC_CODEA-LABEL_LDT+4
```

这个选择符的定义和 GDT 选择符的定义有不同。在解释为什么这么写之前，我想重新说一下自己对段选择符的认识。因为在第二次学习报告中，我对它的描述实在过于粗糙。

段选择符是段的一个 16 位标志符。段选择符并不直接指向段，而是指向段描述符表中定义段的段描述符。段选择符的结构如下图。



从图中可以看出，段选择符有 3 个字段内容:

- 请求特权级 RPL。RPL 被用于特权级保护机制中。
- 表指示标志 TI。当 TI=0 时，表示描述符在 GDT 中。当 TI=1 时，表示描述符在 LDT 中。因为一个任务执行时，可以同时访问到 LDT 和 GDT，所以必须做这样的区别，以防在索引时放生混淆。
- 索引值。用于索引在 GDT 表或 LDT 表中的段描述符。

```
1        ; 这里特意加4，就是为了将段选择符中的第2位TI标志置一
2        SelectorLDTCodeA  equ  LABEL_LDT_DESC_CODEA-LABEL_LDT+4
```

2.2 LDT 的初始化

需要注意的是，既然在 GDT 表中添加了指向 LDT 表的段描述符，就应该在 16 位代码段中初始化它。

```
1        [SECTION  .16]
2        [BITS  16]
3
4            ; 初始化LDT在GDT中的描述符
5            xor  eax,eax
6            mov  ax,ds
7            shl  eax,4
8            add  eax,LABEL_LDT
9            mov  word  [LABEL_DESC_LDT + 2],  ax
10           shr  eax,16
11           mov  byte  [LABEL_DESC_LDT + 4],  al
12           mov  byte  [LABEL_DESC_LDT + 7],  ah
```

LDT 表和 GDT 表区别仅仅在于全局和局部的不同，所以初始化 LDT 表中描述符和之前的操作很类似。具体情况看下面的代码。

```
1        [SECTION  .16]
2        [BITS  16]
3            ; 初始化LDT表中的描述符
4            xor  eax,eax
5            mov  ax,ds
6            shl  eax,4
7            add  eax,LABEL_CODE_A
8            mov  word  [LABEL_LDT_DESC_CODEA + 2],  ax
9            shr  eax,16
10           mov  byte  [LABEL_LDT_DESC_CODEA + 4],  al
```

```
11          mov byte [LABEL_LDT_DESC_CODEA + 7], ah
```

2.3 调用 LDT 中的代码段

```
1       [SECTION .s32]
2       [BITS 32]
3           ; 将LDT表的段选择符加载进LDTR寄存器中
4           ; 在LDTR寄存器中的LDT段描述符存放着LDT表的基址
5           ; 对LDT表中的代码段描述符进行寻址时，以LDTR中LDT表的描述符中的基址为准
6           ; 偏移量由段选择符制定，用于索引LDT表中存放着的代码段描述符
7           ; 当发生任务切换时，LDTR会更换新任务的LDT
8           mov ax, SelectorLDT
9           lldt ax
10          ; CPU根据代码段描述符中的TI标志判断是索引GDT表还是索引LDT表
11          jmp SelectorLDTCodeA:0
```

2.4 总结如何添加 LDT 表

- 添加一个 LDT 表，里面可以类似于 GDT 表，存放代码段、数据段或堆栈段。
- 在 GDT 表中添加新 LDT 表的段描述符。
- 在 16 位代码段中初始化新 LDT 表的段描述符。同时初始化 LDT 表中存放着的所有段描述符。

3 保护机制的实现

3.1 保护机制的相关介绍

虽然我在第二次学习报告中也提到了保护机制，但是过于粗糙。我自己感觉对它的认识也不够深入，所以在这里重新整理一下对保护机制的认识。

3.1.1 段级保护

段限长检查。段描述符的段限长字段用于防止程序寻址到段外内存位置。当 G=0 时，段限长最大为 1MB。当 G=1 时，段限长最大为 4GB。除了检查段限长，处理器也会检查描述符表的长度。GDTR、IDTR 和 LDTR 寄存器都包含有描述符的限长值，用于防止程序在描述符表的外面选择描述符。

段类型检查。段描述符中有 S 标志和 TYPE 字段用于标识段的类型。S=1 时，为系统类型的段。S=0 时，为代码或数据类型的段。TYPE 字段的 4 个比特位用于定义代码、数据和系统描述符的各种类型。处理器在以下两种情况会检查段的类型信息:

- 当段选择符加载进一个段寄存器时需要检查段的类型，因为某些段寄存器只能存放特定类型的描述符。
- 一些指令操作不被允许在某些类型的段上执行。比如任何指令不能写一个可执行段。

特权级检查。段保护机制有 4 个特权级，由段描述符的 DPL 字段中定义。另外还需要知道，CPL 是当前任务的特权级，RPL 是段描述符的特权级。特权级检查可以分为 3 种情况。

- 访问数据段时的特权级检查。

  为了访问数据段中的操作数，数据段中的段选择符需要加载进数据段寄存器或堆栈寄存器中，此时就需要特权级检查。处理器会比较 CPL、RPL 和 DPL，只有当 DPL 的数值大等于 CPL 和 RPL 两者时，处理器才会把选择符加载进段寄存器，否则就会产生一个一般保护异常。需要注意的是，当使用堆栈选择符加载 SS 段寄存器时，DPL、RPL 和 CPL 都需要相同，否则就会产生一个一般保护异常。

- 直接调用或跳转到代码段时的特权级检查。

  首先说明一下一致代码段与非一致代码段的概念。一致代码段允许当前特权级任务跳转到更高特权级的代码段，并且当前任务的 CPL 也会设置为更高特权级代码段的 DPL 值。而非一致代码段不允许这样的行为。段描述符中的 C 标志用于标识一致代码段和非一致代码段。C=1 时，为一致代码段。C=0 时，为非一致代码段。需要注意的是，一致代码段和非一致代码段都不允许当前任务跳转到更低特权级的代码段。

当访问一致代码段时，处理器会忽略对 RPL 的检查，只要求 CPL 大等于 DPL。当访问非一致代码段时，CPL 必须等于 DPL，而 RPL 必须小等于 DPL。

- 通过调用门访问代码段时的特权级检查。

  因为处理器通过调用门描述符去访问代码段描述符，所以还需要对调用门描述符进行特权级检查。处理器要求 CPL 和 RPL 都要小于调用门描述符的 DPL。随后进行对代码段的特权级检查。当使用 CALL 指令时，对于一致代码段和非一致代码段都只要求 DPL 小等于 CPL。当使用 JMP 指令时，对于一致代码段要求 DPL 小等于 CPL，对于非一致代码段要求 DPL 等于 CPL。

3.1.2 页级保护

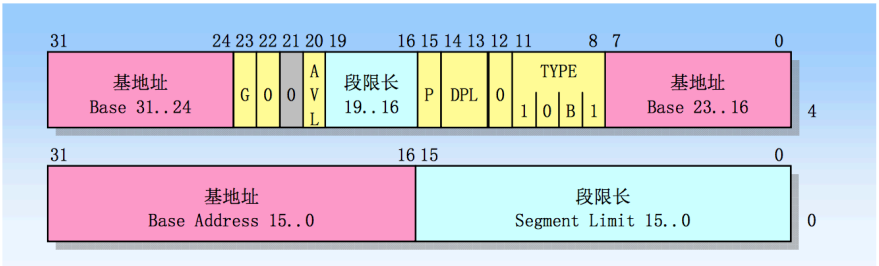
只有当所有的段级保护通过后，才会进行页级保护检查。

页级保护分为两类权限，分别为超级用户级和普通用户级。特权级 0、1、2 的任务被归类为超级用户级，特权级 3 的任务被归类为普通用户级。页面也分为超级用户级和普通用户级。页目录和页表项中的 U/S 标志用于标识该页面的级别。只有当两者的 U/S 都为 0，页面才是超级用户级，否则是普通用户级。普通用户级的程序不可访问超级用户级的页面，超级用户级的程序可以读/写/执行所有页面。

普通用户级的程序只能访问普通用户级的页面，但是不一定可以写。只有当页面和页目录的读写标志 R/W 都为 1 时，普通用户级的程序才能写普通用户级的页面。

3.2 编写保护机制

在编写保护机制前，我觉得有必要先把段描述符的属性搞清楚。先来回忆一下段描述符的格式，如下图。



可以清楚地看到，描述符的第 6 和第 7 个字节是属性与段限长的混合。对这 16 位进行赋值。假设有 20 位的段限长，首先给描述符低 16 位的段限长赋值，代码为 LEN & 0FFFFh。然后将其右移 8 位，剩 12 位，最后取剩余的 12 位的高 4 位，并对第 7 字节中的段限长字段赋值。代码为 ((LEN » 8) & 0F00h)。现在第 6 和第 7 字节仅剩属性还未赋值，代码为 ATTR & 0F0FFh。现在重温一下段描述符的数据结构，我觉得现在对这个数据结构的理解完全掌握了。

```
%marco Descriptor 3
    dw %2 & 0FFFFh
    dw %1 & 0FFFFh
    dw (%1 >> 16) & 0FFh
    dw ((%2 >> 8) & 0F00h) | (%3 & 0F0FFh)
    db (%1 >> 24) & 0FFh
%endmarco
```

3.2.1 对段描述符属性的编程

根据宏定义，段的属性由第三个参数决定。下面定义一些宏，用于定义段的属性。在代码中我会注释为什么这么定义宏。

```
; 第三个参数的高4位分别为G、D/B、保留比特位和AVL位。
; 4h=0100b，相当于将D/B设置为1。
; 对于代码段，这个标志用于指出该段中的指令引用有效地址和操作数的默认长度。
; D/B为1时，有效地址为32位，操作数长度为32位或8位。
; D/B为0时，有效地址为16位，操作数长度为16位或8位。
; DA_32代表段为32位段
DA_32 EQU 4000h

; 第三参数的低8位为P、DPL、S和TYPE。
; P表示段是否存在在内存中
; DPL表示段描述符的特权级
; S表示该段是系统段描述符或门描述符还是代码或数据段

; 给出存储段描述符的属性

; TYPE有4位，最高位为0时，为数据段，最高位为1时，为代码段。
; 为数据段时，TYPE为0、E(扩展方向)、W(可写)和A(已访问)。
; 为代码段时，TYPE为1、C(一致性)、R(可读)和A(已访问)。
; 后面会有一张表，用于整理TYPE的各种情况。

; 数据段
DA_DR EQU 90h ; 存在的只读数据段
DA_DRA EQU 91h ; 存在的已访问只读数据段
DA_DRW EQU 92h ; 存在的可读写数据段
DA_DRWA EQU 93h ; 存在的已访问的可读写数据段
DA_C EQU 98h ; 存在的仅执行代码段

; 代码段
DA_CA EQU 99h ; 存在的已访问的仅执行代码段
DA_CR EQU 9Ah ; 存在的可执行可读代码段
DA_CCO EQU 9Ch ; 存在的可执行的一致代码段
DA_CCOR EQU 9Eh ; 存在的可执行的可读一致代码段

; 给出系统段描述符的属性

; 此时TYPE有16种情况，除了3个保留情况以外，有13种段描述符
; 后面会有一张表，用于整理系统段描述符的各种情况

DA_LDT EQU 82h ; LDT表描述符
DA_TaskGate EQU 85h ; 任务门描述符
```

```
41      DA_386TSS EQU 89h ; 32位TSS描述符
42      DA_386CGate EQU 8Ch ; 32位调用门描述符
43      DA_386IGate EQU 8Eh ; 32位中断门描述符
44      DA_386TGate EQU 8Fh ; 32位陷阱门描述符
```

下面是各种类型的代码段描述符和数据段描述符。

类型（TYPE）字段					描述符类型	说明
十进制	位 11	位 10	位 9	位 8		
		E	W	A		
0	0	0	0	0	数据	只读
1	0	0	0	1	数据	只读，已访问
2	0	0	1	0	数据	可读/写
3	0	0	1	1	数据	可读/写，已访问
4	0	1	0	0	数据	向下扩展，只读
5	0	1	0	1	数据	向下扩展，只读，已访问
6	0	1	1	0	数据	向下扩展，可读/写
7	0	1	1	1	数据	向下扩展，可读/写，已访问
		C	R	A		
8	1	0	0	0	代码	仅执行
9	1	0	0	1	代码	仅执行，已访问
10	1	0	1	0	代码	执行/可读
11	1	0	1	1	代码	执行/可读，已访问
12	1	1	0	0	代码	一致性段，仅执行
13	1	1	0	1	代码	一致性段，仅执行，已访问
14	1	1	1	0	代码	一致性段，执行/可读
15	1	1	1	1	代码	一致性段，执行/可读，已访问

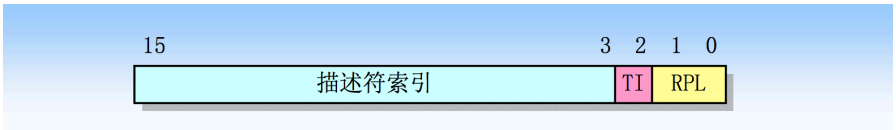
下面是各种类型的系统段描述符和门描述符。

类型（TYPE）字段					说明	
十进制	位 11	位 10	位 9	位 8		
0	0	0	0	0	Reserved	保留
1	0	0	0	1	16-Bit TSS (Available)	16 位 TSS（可用）
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16-Bit TSS (Busy)	16 位 TSS（忙）
4	0	1	0	0	16-Bit Call Gate	16 位调用门
5	0	1	0	1	Task Gate	任务门
6	0	1	1	0	16-Bit Interrupt Gate	16 位中断门
7	0	1	1	1	16-Bit Trap Gate	16 位陷阱门
8	1	0	0	0	Reserved	保留
9	1	0	0	1	32-Bit TSS (Available)	32 位 TSS（可用）
10	1	0	1	0	Reserved	保留
11	1	0	1	1	32-Bit TSS (Busy)	32 位 TSS（忙）
12	1	1	0	0	32-Bit Call gate	32 位调用门
13	1	1	0	1	Reserved	保留
14	1	1	1	0	32-Bit Interrupt Gate	32 位中断门
15	1	1	1	1	32-Bit Trap Gate	32 位陷阱门

到此，我自认为自己对段描述符是理解得比较透彻了。接下来进行对段选择符的编程。

3.2.2 对段选择符属性的编程

段选择符相对于段描述符而言简单很多，因为它只由三部分组成，分别是描述符索引、TI 标志和 RPL 标志。TI 标志表示段描述符在 GDT 中还是在 LDT 中。RPL 标志用于表示段选择符的请求特权级。段选择符的结构如下图。



下面是实现代码。

```
    ; 用于设置TI位
    SA_TIG  EQU  0
    SA_TIL  EQU  4

    ; 用于设置RPL字段
    SA_RPL0 EQU  0
```

```
7       SA_RPL1  EQU  1
8       SA_RPL2  EQU  2
9       SA_RPL3  EQU  3
```

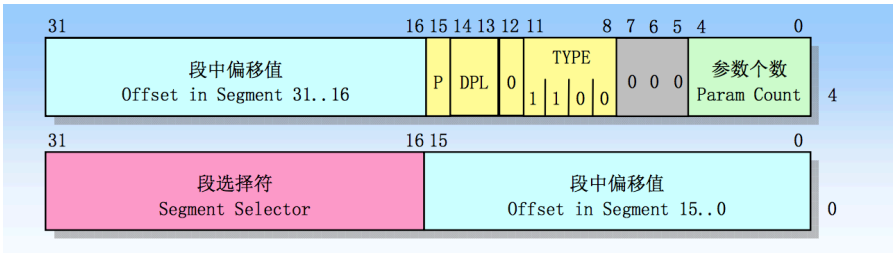
3.2.3 小结

有了对段描述符、段选择符的属性编程，x86 的保护机制也就可以实现了。

4 调用门的实现

虽然处理器可以通过 jmp 或 call 直接进行转移，但是这样所能进行的代码段间转移是非常有限的。对于非一致代码段，只能在相同特权级代码段之间转移。对于一致代码段，也最多从低特权级代码段向高特权级代码段进行转移。如果想自由地进行不同特权级之间的转移，则需要门描述符或 TSS。

首先看一下门描述符的结构，见下图。



有了对段描述符的编程经历，门描述符的编程就变得简单很多了。它的数据结构代码和段描述符有相似之处。

```
%marco Gate 4
    dw %2 & 0FFFFh
    dw %1
    db (%3 & 1Fh) | ((%4 << 8) & 0FF00h)
    dw (%2 >> 16) & 0FFFFh
%endmarco
```

4.1 添加门描述符

首先在 GDT 表中添加门描述符。

```
[SECTION .gdt]
LABEL_GDT: Descriptor 0, 0, 0
LABEL_DESC_CODE: Descriptor 0, SegCodeDestLen-1, DA_C + DA_32
; ...
; 这里的SelctorCodeDest，是指向LABEL_DESC_CODE段描述符的
; 门描述符的特点就是间接调用代码段
LABEL_CALL_GATE_TEST: Gate SelectorCodeDest, 0, 0, DA_386CGate + DA_DPL0
; ...
SelectorCodeDest equ LABEL_DESC_CODE - LABEL_GDT
SelectorCallGateTest equ LABEL_CALL_GATE_TEST - LABEL_GDT
```

我发现自己有个东西忘记交代了，就是 GDT 表的第一个描述符是空描述符，所有总是有

```
LABEL_GDT: Descriptor 0, 0, 0
```

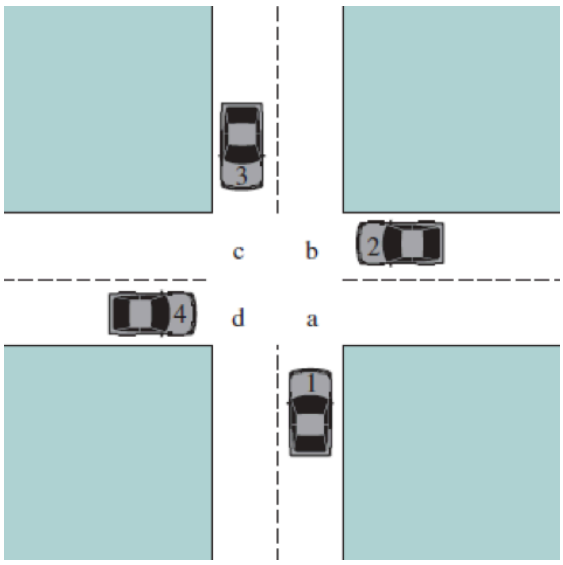
5 迟到的实验一

之前一直以为实验一只是编译内核，周四上课的时候才知道实验一还有同步互斥问题，所以在此尝试做一下，如果代码有问题请老师指出错误。

5.1 问题描述

5.1.1 现状

有两条道路双向两个车道，即每条路每个方向只有一个车道，两条道路十字交叉。假设车辆只能向前直行，而不允许转弯和后退。如下图所示。



5.1.2 资源需求

车辆为了向前，对资源有如下需求。

- 向北行驶的车辆 1 需要象限 a 和象限 b。
- 向西行驶的车辆 2 需要象限 b 和象限 c。
- 向南行驶的车辆 3 需要象限 c 和象限 d。
- 向东行驶的车辆 4 需要象限 d 和象限 a。

5.1.3 交通规则

每个车辆都需要满足如下的规则。

- 有多个方向的车辆同时到达十字路口时，需要遵循右边车辆优先通行规则，除非该车在十字路口等待时收到一个立即通行的信号。
- 同一车道上的车辆必须依照顺序通过十字路口。
- 向北行驶的车辆只有在占用象限 a 的情况下才能判断象限 b 是否有空。对于其他方向的车辆也是如此。

5.1.4 要求

- 避免产生死锁。
- 避免产生饥饿。

5.2 解决方案

周日一天都坐在电脑前面写这个代码，不得不说，虽然这个问题描述起来简单，实际写起来需要考虑的东西确实很多。也是自己对多线程编程不够熟悉，这次算是积累了经验。

5.2.1 对死锁的仿真

为了达到死锁的效果，就必须仿真现实中的情况。现实中汽车过十字路口总是要时间的，所以我写了一个 sleep() 函数，实现代码如下：

```
1  void sleep(int seconds)
2  {
3      pthread_mutex_t mutex;
4      pthread_mutex_init(&mutex, NULL);
5      pthread_cond_t cond;
6      pthread_cond_init(&cond, NULL);
7      struct timeval now;
8      struct timespec outtime;
9      gettimeofday(&now, NULL);
10     outtime.tv_sec = now.tv_sec + seconds;
11     outtime.tv_nsec = now.tv_usec * 1000;
12     pthread_mutex_lock(&mutex);
13     pthread_cond_timedwait(&cond, &mutex, &outtime);
14     pthread_mutex_unlock(&mutex);
15     pthread_mutex_destroy(&mutex);
16     pthread_cond_destroy(&cond);
17 }
```

为了仿真四个方向来车，我编写了四个函数，分别是 from_east()、from_west()、from_south() 和 from_north() 函数。在函数中，我首先写了让四辆小车同时到达路口时死锁。

```
void * from_north(void *arg)
{
    car_num num_struct = *(car_num*)arg;
    int num = num_struct.total_num;
    int north_num = num_struct.direction_num;

    pthread_mutex_lock(&source_c);
    printf("car %d from North arrives at crossing\n", num);

    pthread_mutex_lock(&source_d);
    sleep(1);
    pthread_mutex_unlock(&source_c);

    printf("car %d from North leaving crossing\n", num);
    pthread_mutex_unlock(&source_d);

    return (void*)0;
}
```

其他三个函数类似于上述函数的形式，在 main() 分别创建四个函数的线程。运行之后，能得到死锁的结果。这样就达成了死锁的仿真。

5.2.2 死锁的解决

这个总结是写在完成代码之后的，所以从此时看来，我觉得我解决死锁的方法并不好。在给出当前解决方案后，我会分析可能发生的错误。

在一开始，我觉得可以用计数信号量完成，所以声明了一个全局变量 remaining_sources。变量名很直观，就是四个路口剩余的数量。我简单得考虑了，只要有车到达，remaining_sources 就减一。只要有车离开，remaining_sources 就加一。当 remaining_sources 为 0 时，则通知 manager() 函数处理，自己将当前路口解锁，并被 pthread_cond_wait() 函数挂起。

manager() 函数等待着 remaining_sources 为 0 的情况。一旦 remaining_sources 为 0，它首先发出一行字符串，说明检测到了死锁。然后使用 pthread_cond_wait() 等待当前其他三个方向的车离开。当其他三个线程的任意一个线程离开路口时，会对 remaining_sources 进行条件检查，如果发现正处于 remaining==0 的情况，则用 pthread_cond_signal() 唤醒 manager() 函数。manager() 函数被唤醒后，接着唤醒之前被挂起的线程。

这个想法在大部分情况是有效的，但是会有特殊情况。实现代码如下：

```
void * from_north(void *arg)
{
    // ...
    pthread_mutex_lock(&source_c);
    printf("car %d from North arrives at crossing\n", num);

    --remaining_sources;
    if(remaining_sources == 0)
    {
```

```
10              direction = NORTH;
11              pthread_cond_signal(&manager_lock_cond);
12              pthread_mutex_unlock(&source_c);
13              pthread_cond_wait(&north_wait_manager, &manager_lock);
14              pthread_mutex_lock(&source_c);
15          }
16
17          pthread_mutex_lock(&source_d);
18          sleep(1);
19          pthread_mutex_unlock(&source_c);
20
21          if(remaining_sources == 0)
22              pthread_cond_signal(&manager_unlock_cond);
23
24          printf("car %d from North leaving crossing\n", num);
25          ++remaining_sources;
26          pthread_mutex_unlock(&source_d);
27          // ...
28
29          return (void*)0;
30      }
31
32      // ...
33
34      void * manager(void *arg)
35      {
36          while(1)
37          {
38              pthread_mutex_lock(&manager_lock);
39              pthread_cond_wait(&manager_lock_cond, &manager_lock);
40              switch(direction)
41              {
42                  case NORTH:
43                      printf("DEADLOCK: car jam detected, signalling East to go\n");
44                      break;
45                  case EAST:
46                      printf("DEADLOCK: car jam detected, signalling South to go\n");
47                      break;
48                  case WEST:
49                      printf("DEADLOCK: car jam detected, signalling North to go\n");
50                      break;
51                  case SOUTH:
52                      printf("DEADLOCK: car jam detected, signalling West to go\n");
53                      break;
54                  default:
55                      err_exit();
56                      break;
57              }
58              pthread_cond_wait(&manager_unlock_cond, &manager_lock);
59              switch(direction)
60              {
61                  case NORTH:
62                      pthread_cond_signal(&north_wait_manager);
63                      break;
64                  case EAST:
65                      pthread_cond_signal(&east_wait_manager);
```

```
66                      break;
67                  case WEST:
68                      pthread_cond_signal(&west_wait_manager);
69                      break;
70                  case SOUTH:
71                      pthread_cond_signal(&south_wait_manager);
72                      break;
73                  default:
74                      err_exit();
75                      break;
76              }
77              pthread_mutex_unlock(&manager_lock);
78          }
79          return (void*)0;
80      }
```

这个方法乍一想真的很好，但是之后运行的时候有点问题，这让我开始思考其中的错误。现在谈一种情况，当三个路口 north、east 和 south 都被占用了，此时 remaining_sources 为 1。此时 north 方向的线程特别快，一下子都离开了路口，但是还没来得及将 remaining_sources 加一。此时 west 方向的车来了，而且及时将 remaining_sources 减一并判断了，此时 remaining_sources 为 0，于是执行我预先的操作。

可惜的是，因为 north 方向的车已经走了，所以实际上死锁情况并不存在，remaining_sources 之后也不会等于 0。此时，如果 manager() 被挂起，就不会被 pthread_cond_signal() 唤醒。这样 west 方向的车也就不会被唤醒，也就出现了问题。

为了避免问题，我已经在代码中修改，让 north 方向的车，在对资源解锁前，先将 remaining_sources 加一，这样就不会出现上述情况了。但是我感觉还是不太安全，这种判断变量值的方法和我之后使用的专门函数管理的方法相比，还是有点不靠谱。

5.2.3 让同方向的车按序通过路口

我定义了一个函数 north_manager() 专门管理同一方向车的通行顺序。为了让车按顺序通过，我首先将同一方向的车进行编号，如下所示：

```
1   struct car_num
2   {
3       int total_num;
4       int direction_num;
5   };
6   typedef struct car_num car_num;
7
8   // ...
9
10  int main()
11  {
12      for(i = 0; i < len; ++i)
13      {
14          car_nums[i].total_num = i + 1;
15          switch(cars[i])
16          {
```

```
                case 'n':
                    car_nums[i].direction_num = north_num;
                    ++north_num;
                    break;
                case 'e':
                    car_nums[i].direction_num = east_num;
                    ++east_num;
                    break;
                case 'w':
                    car_nums[i].direction_num = west_num;
                    ++west_num;
                    break;
                case 's':
                    car_nums[i].direction_num = south_num;
                    ++south_num;
                    break;
                default:
                    err_exit();
                    break;
            }
        }
    }
```

得到各个方向的车的编号后，就可以让 north_manager() 发挥作用了。首先 north 方向的所有线程一开始都会调用 pthread_cond_wait() 被挂起。为了让车能在 north_manager() 函数发出信号前已经在 wait，我给每个线程都配备了相应的锁。并且让 north_manager() 先睡眠 1 秒，确保线程都持有锁或者在等待中了。为了让 north_manager() 函数能够唤醒特定的线程，我为每个线程声明了相应的条件变量。

只要看 north_manager() 就很容易发现，函数只是依次在唤醒线程，这也就达到了同一方向的车按顺序通过的目的。当然，函数需要等当前线程离开后才能唤醒下一线程。所以当 north_manager() 唤醒一个线程后，就通过 pthread_cond_wait() 被挂起。当该线程离开时，通过 pthread_cond_signal() 唤醒 north_manager()。

```
    pthread_mutex_t* north_order_lock;
    pthread_mutex_t* east_order_lock;
    pthread_mutex_t* west_order_lock;
    pthread_mutex_t* south_order_lock;

    pthread_cond_t* north_order;
    pthread_cond_t* east_order;
    pthread_cond_t* west_order;
    pthread_cond_t* south_order;

    void * from_north(void *arg)
    {
        car_num num_struct = *(car_num*)arg;
        int num = num_struct.total_num;
        int north_num = num_struct.direction_num;

        pthread_mutex_lock(north_order_lock+north_num);
        pthread_cond_wait(north_order+north_num, north_order_lock+north_num);
        pthread_mutex_unlock(north_order_lock+north_num);
```

```
20
21          pthread_mutex_lock(&source_c);
22
23          pthread_mutex_lock(&source_d);
24          sleep(1);
25          pthread_mutex_unlock(&source_c);
26
27          pthread_mutex_lock(north_order_lock+north_num);
28          pthread_mutex_unlock(north_order_lock+north_num);
29          pthread_cond_signal(&north_manager_cond);
30
31          pthread_mutex_unlock(&source_d);
32
33          return (void*)0;
34      }
35      void * north_manager(void *arg)
36      {
37          int i;
38          sleep(1);
39          for(i = 0; i < north_num; ++i)
40          {
41              pthread_mutex_lock(north_order_lock+i);
42              pthread_mutex_unlock(north_order_lock+i);
43              pthread_cond_signal(north_order+i);
44
45              pthread_mutex_lock(north_order_lock+i);
46              pthread_cond_wait(&north_manager_cond, north_order_lock+i);
47              pthread_mutex_unlock(north_order_lock+i);
48          }
49
50          return (void*)0;
51      }
```

5.2.4 右边优先以及解决饥饿

我发现对右边优先的理解不同，可能给出的解决方案有比较大的偏差。所以我先叙述一下自己对“右边优先”这一概念的理解。当 north 方向的车和 east 方向的车都在请求 c 路口时，north 方向的车有较高的优先级。因为 north 方向有较高的优先级，所以如果 north 方向有一个很长的车队，那么 c 路口就会一直被 north 方向的车占用，east 方向的车也就发生了饥饿。

为了避免饥饿，当 north 方向的车首先占用 c 路口后，把 east 方向的车调为较高的优先级。如果之后还有 north 方向的车和 east 方向的车同时请求，依据优先级，east 方向的车就可以请求得到 c 路口资源。

为了实现这一想法，我首先定义了一个请求队列的数据结构。这个结构只有两个变量，一个是队列长度，一个是请求队列中的车。根据实际情况，队列长度是不可能超过 2 的。如果超过了 2，就调用 err_exit() 返回异常情况。

north 方向的车在占有 c 路口之前，先更新请求队列，并通过 pthread_cond_signal() 函数向 source_c_manager() 函数发出请求。source_c_manager() 函数根据队列长度和队列中

各方向车的优先级做出相应的安排。值得注意的是，当请求队列只有一辆车时，处理完该情况后，需要将各方向车的优先级还原为初始情况。

实现代码如下：

```
struct request_queue
{
    int directions[2];
    int len;
};

typedef struct request_queue request_queue;

void err_exit()
{
    printf("somthing wrong!!!\n");
    exit(1);
}

void * from_north(void *arg)
{
    pthread_mutex_lock(&c_queue_lock);
    request_c_queue.directions[request_c_queue.len] = NORTH;
    ++request_c_queue.len;
    pthread_cond_signal(&request_source_c);
    pthread_cond_wait(&north_wait_c, &c_queue_lock);
    pthread_mutex_unlock(&c_queue_lock);

    pthread_mutex_lock(&source_c);

    pthread_mutex_lock(&d_queue_lock);
    request_d_queue.directions[request_d_queue.len] = NORTH;
    ++request_d_queue.len;
    pthread_cond_signal(&request_source_d);
    pthread_cond_wait(&north_wait_d, &d_queue_lock);
    pthread_mutex_unlock(&d_queue_lock);

    pthread_mutex_lock(&source_d);
    sleep(1);
    pthread_mutex_unlock(&source_c);

    pthread_mutex_lock(&c_queue_lock);
    if(request_c_queue.len != 0)
        pthread_cond_signal(&request_source_c);
    pthread_mutex_unlock(&c_queue_lock);

    pthread_mutex_unlock(&source_d);

    pthread_mutex_lock(&d_queue_lock);
    if(request_d_queue.len != 0)
        pthread_cond_signal(&request_source_d);
    pthread_mutex_unlock(&d_queue_lock);

    return (void*)0;
}

```

```
52  void * source_a_manager(void *arg)
53  {
54      pthread_mutex_t lock;
55      int south_level = CPL0;
56      int west_level = CPL1;
57
58      pthread_mutex_init(&lock, NULL);
59      pthread_cond_signal(&south_wait_a);
60      request_a_queue.len = 0;
61      while(1)
62      {
63          pthread_mutex_lock(&lock);
64          pthread_cond_wait(&request_source_a, &lock);
65
66          pthread_mutex_lock(&a_queue_lock);
67          if(request_a_queue.len == 1)
68          {
69              --request_a_queue.len;
70              south_level = CPL0;
71              west_level = CPL1;
72              switch(request_a_queue.directions[0])
73              {
74                  case SOUTH:
75                      pthread_cond_signal(&south_wait_a);
76                      break;
77                  case WEST:
78                      pthread_cond_signal(&west_wait_a);
79                      break;
80                  default:
81                      err_exit();
82                      break;
83              }
84          }
85          else if(request_a_queue.len == 2)
86          {
87              --request_a_queue.len;
88              if(south_level == CPL0)
89              {
90                  south_level = CPL1;
91                  west_level = CPL0;
92                  request_a_queue.directions[0] = WEST;
93                  pthread_cond_signal(&south_wait_a);
94              }
95              else
96              {
97                  south_level = CPL0;
98                  west_level = CPL1;
99                  request_a_queue.directions[0] = SOUTH;
100                 pthread_cond_signal(&west_wait_a);
101             }
102         }
103         else
104             err_exit();
105         pthread_mutex_unlock(&a_queue_lock);
106
107         pthread_mutex_unlock(&lock);
```

```
108         }
109         return (void*)0;
110     }
```

5.2.5 总结

将死锁、按顺序通过、右边优先和饥饿等问题解决后，实验一算是完成了。不过我始终觉得死锁问题的解决方案不够好，不够稳定，可能在我某些没考虑到的情况，它就出错了。

5.3 实现代码

```
1  //
2  //  main.c
3  //
4  //  Created by pengsida on 2016/11/20.
5  //  Copyright © 2016年 pengsida. All rights reserved.
6  //
7  //
8
9  #include<stdio.h>
10 #include<stdlib.h>
11 #include<string.h>
12 #include<pthread.h>
13 #include<sys/time.h>
14 #include <errno.h>
15
16 #define MAX 1000
17
18 #define NORTH 1
19 #define EAST 2
20 #define WEST 3
21 #define SOUTH 4
22
23 #define CPL0 0
24 #define CPL1 0
25
26 int remaining_sources;
27 int direction;
28 size_t len;
29 int north_num;
30 int east_num;
31 int west_num;
32 int south_num;
33
34 pthread_mutex_t manager_lock;
35 pthread_mutex_t count_lock;
36 pthread_mutex_t remaining_lock;
37
38 pthread_mutex_t source_a;
39 pthread_mutex_t source_b;
```

```
40 pthread_mutex_t source_c;
41 pthread_mutex_t source_d;
42
43 pthread_mutex_t first_north_lock;
44 pthread_mutex_t first_east_lock;
45 pthread_mutex_t first_west_lock;
46 pthread_mutex_t first_south_lock;
47
48 pthread_mutex_t* north_order_lock;
49 pthread_mutex_t* east_order_lock;
50 pthread_mutex_t* west_order_lock;
51 pthread_mutex_t* south_order_lock;
52
53 pthread_mutex_t north_manager_lock;
54 pthread_mutex_t east_manager_lock;
55 pthread_mutex_t west_manager_lock;
56 pthread_mutex_t south_manager_lock;
57
58 pthread_mutex_t a_queue_lock;
59 pthread_mutex_t b_queue_lock;
60 pthread_mutex_t c_queue_lock;
61 pthread_mutex_t d_queue_lock;
62
63 pthread_cond_t manager_lock_cond;
64 pthread_cond_t manager_unlock_cond;
65
66 pthread_cond_t* north_order;
67 pthread_cond_t* east_order;
68 pthread_cond_t* west_order;
69 pthread_cond_t* south_order;
70
71 pthread_cond_t first_north;
72 pthread_cond_t first_east;
73 pthread_cond_t first_west;
74 pthread_cond_t first_south;
75
76 pthread_cond_t north_wait_manager;
77 pthread_cond_t east_wait_manager;
78 pthread_cond_t west_wait_manager;
79 pthread_cond_t south_wait_manager;
80
81 pthread_cond_t north_manager_cond;
82 pthread_cond_t east_manager_cond;
83 pthread_cond_t west_manager_cond;
84 pthread_cond_t south_manager_cond;
85
86 pthread_cond_t request_source_a;
87 pthread_cond_t request_source_b;
88 pthread_cond_t request_source_c;
89 pthread_cond_t request_source_d;
90
91 pthread_cond_t south_wait_a;
92 pthread_cond_t west_wait_a;
93 pthread_cond_t east_wait_b;
94 pthread_cond_t south_wait_b;
95 pthread_cond_t north_wait_c;
```

```
96  pthread_cond_t east_wait_c;
97  pthread_cond_t west_wait_d;
98  pthread_cond_t north_wait_d;
99
100 struct car_num
101 {
102     int total_num;
103     int direction_num;
104 };
105
106 typedef struct car_num car_num;
107
108 struct request_queue
109 {
110     int directions[2];
111     int len;
112 };
113
114 typedef struct request_queue request_queue;
115
116 request_queue request_a_queue;
117 request_queue request_b_queue;
118 request_queue request_c_queue;
119 request_queue request_d_queue;
120
121 void sleep(int seconds)
122 {
123     pthread_mutex_t mutex;
124     pthread_mutex_init(&mutex, NULL);
125     pthread_cond_t cond;
126     pthread_cond_init(&cond, NULL);
127     struct timeval now;
128     struct timespec outtime;
129     gettimeofday(&now, NULL);
130     outtime.tv_sec = now.tv_sec + seconds;
131     outtime.tv_nsec = now.tv_usec * 1000;
132     pthread_mutex_lock(&mutex);
133     pthread_cond_timedwait(&cond, &mutex, &outtime);
134     pthread_mutex_unlock(&mutex);
135     pthread_mutex_destroy(&mutex);
136     pthread_cond_destroy(&cond);
137 }
138
139 void err_exit()
140 {
141     printf("somthing wrong!!!\n");
142     exit(1);
143 }
144
145 void * from_north(void *arg)
146 {
147     car_num num_struct = *(car_num*)arg;
148     int num = num_struct.total_num;
149     int north_num = num_struct.direction_num;
150
151     pthread_mutex_lock(north_order_lock+north_num);
```

```
152     pthread_cond_wait(north_order+north_num, north_order_lock+north_num);
153     pthread_mutex_unlock(north_order_lock+north_num);
154 
155     pthread_mutex_lock(&c_queue_lock);
156     request_c_queue.directions[request_c_queue.len] = NORTH;
157     ++request_c_queue.len;
158     pthread_cond_signal(&request_source_c);
159     pthread_cond_wait(&north_wait_c, &c_queue_lock);
160     pthread_mutex_unlock(&c_queue_lock);
161 
162     pthread_mutex_lock(&source_c);
163     printf("car %d from North arrives at crossing\n", num);
164 
165     --remaining_sources;
166     if(remaining_sources == 0)
167     {
168         direction = NORTH;
169         pthread_cond_signal(&manager_lock_cond);
170         pthread_mutex_unlock(&source_c);
171         pthread_cond_wait(&north_wait_manager, &manager_lock);
172         pthread_mutex_lock(&source_c);
173     }
174 
175     pthread_mutex_lock(&d_queue_lock);
176     request_d_queue.directions[request_d_queue.len] = NORTH;
177     ++request_d_queue.len;
178     pthread_cond_signal(&request_source_d);
179     pthread_cond_wait(&north_wait_d, &d_queue_lock);
180     pthread_mutex_unlock(&d_queue_lock);
181 
182     pthread_mutex_lock(&source_d);
183     sleep(1);
184     pthread_mutex_unlock(&source_c);
185 
186     pthread_mutex_lock(&c_queue_lock);
187     if(request_c_queue.len != 0)
188         pthread_cond_signal(&request_source_c);
189     pthread_mutex_unlock(&c_queue_lock);
190 
191 //    pthread_mutex_lock(&north_manager_lock);
192     pthread_mutex_lock(north_order_lock+north_num);
193     pthread_mutex_unlock(north_order_lock+north_num);
194     pthread_cond_signal(&north_manager_cond);
195 //    pthread_mutex_unlock(&north_manager_lock);
196 
197     if(remaining_sources == 0)
198         pthread_cond_signal(&manager_unlock_cond);
199 
200     printf("car %d from North leaving crossing\n", num);
201     ++remaining_sources;
202     pthread_mutex_unlock(&source_d);
203 
204     pthread_mutex_lock(&d_queue_lock);
205     if(request_d_queue.len != 0)
206         pthread_cond_signal(&request_source_d);
207     pthread_mutex_unlock(&d_queue_lock);
```

```
208
209     return (void*)0;
210 }
211
212 void * from_west(void *arg)
213 {
214     car_num num_struct = *(car_num*)arg;
215     int num = num_struct.total_num;
216     int west_num = num_struct.direction_num;
217
218     pthread_mutex_lock(west_order_lock+west_num);
219     pthread_cond_wait(west_order+west_num, west_order_lock+west_num);
220     pthread_mutex_unlock(west_order_lock+west_num);
221
222     pthread_mutex_lock(&d_queue_lock);
223     request_d_queue.directions[request_d_queue.len] = WEST;
224     ++request_d_queue.len;
225     pthread_cond_signal(&request_source_d);
226     pthread_cond_wait(&west_wait_d, &d_queue_lock);
227     pthread_mutex_unlock(&d_queue_lock);
228
229     pthread_mutex_lock(&source_d);
230     printf("car %d from West arrives at crossing\n", num);
231
232     // pthread_mutex_lock(&remaining_sources_lock);
233     --remaining_sources;
234     if(remaining_sources == 0)
235     {
236         direction = WEST;
237         pthread_cond_signal(&manager_lock_cond);
238         pthread_mutex_unlock(&source_d);
239         pthread_cond_wait(&west_wait_manager, &manager_lock);
240         pthread_mutex_lock(&source_d);
241     }
242
243     pthread_mutex_lock(&a_queue_lock);
244     request_a_queue.directions[request_a_queue.len] = WEST;
245     ++request_a_queue.len;
246     pthread_cond_signal(&request_source_a);
247     pthread_cond_wait(&west_wait_a, &a_queue_lock);
248     pthread_mutex_unlock(&a_queue_lock);
249
250     pthread_mutex_lock(&source_a);
251     sleep(1);
252     pthread_mutex_unlock(&source_d);
253
254     pthread_mutex_lock(&d_queue_lock);
255     if(request_d_queue.len != 0)
256         pthread_cond_signal(&request_source_d);
257     pthread_mutex_unlock(&d_queue_lock);
258
259 //    pthread_mutex_lock(&west_manager_lock);
260     pthread_mutex_lock(west_order_lock+west_num);
261     pthread_mutex_unlock(west_order_lock+west_num);
262     pthread_cond_signal(&west_manager_cond);
263 //    pthread_mutex_unlock(&west_manager_lock);
```

```
264
265     if(remaining_sources == 0)
266         pthread_cond_signal(&manager_unlock_cond);
267
268     printf("car %d from West leaving crossing\n", num);
269     ++remaining_sources;
270     pthread_mutex_unlock(&source_a);
271
272     pthread_mutex_lock(&a_queue_lock);
273     if(request_a_queue.len != 0)
274         pthread_cond_signal(&request_source_a);
275     pthread_mutex_unlock(&a_queue_lock);
276
277     return (void*)0;
278 }
279
280 void * from_south(void *arg)
281 {
282     car_num num_struct = *(car_num*)arg;
283     int num = num_struct.total_num;
284     int south_num = num_struct.direction_num;
285
286     pthread_mutex_lock(south_order_lock+south_num);
287     pthread_cond_wait(south_order+south_num, south_order_lock+south_num);
288     pthread_mutex_unlock(south_order_lock+south_num);
289
290     pthread_mutex_lock(&a_queue_lock);
291     request_a_queue.directions[request_a_queue.len] = SOUTH;
292     ++request_a_queue.len;
293     pthread_cond_signal(&request_source_a);
294     pthread_cond_wait(&south_wait_a, &a_queue_lock);
295     pthread_mutex_unlock(&a_queue_lock);
296
297     pthread_mutex_lock(&source_a);
298     printf("car %d from South arrives at crossing\n", num);
299
300     --remaining_sources;
301     if(remaining_sources == 0)
302     {
303         direction = SOUTH;
304         pthread_cond_signal(&manager_lock_cond);
305         pthread_mutex_unlock(&source_a);
306         pthread_cond_wait(&south_wait_manager, &manager_lock);
307         pthread_mutex_lock(&source_a);
308     }
309
310     pthread_mutex_lock(&b_queue_lock);
311     request_b_queue.directions[request_b_queue.len] = SOUTH;
312     ++request_b_queue.len;
313     pthread_cond_signal(&request_source_b);
314     pthread_cond_wait(&south_wait_b, &b_queue_lock);
315     pthread_mutex_unlock(&b_queue_lock);
316
317     pthread_mutex_lock(&source_b);
318     sleep(1);
319     pthread_mutex_unlock(&source_a);
```

```
320
321     pthread_mutex_lock(&a_queue_lock);
322     if(request_a_queue.len != 0)
323         pthread_cond_signal(&request_source_a);
324     pthread_mutex_unlock(&a_queue_lock);
325
326 //    pthread_mutex_lock(&south_manager_lock);
327     pthread_mutex_lock(south_order_lock+south_num);
328     pthread_mutex_unlock(south_order_lock+south_num);
329     pthread_cond_signal(&south_manager_cond);
330 //    pthread_mutex_unlock(&south_manager_lock);
331
332     if(remaining_sources == 0)
333         pthread_cond_signal(&manager_unlock_cond);
334
335     printf("car %d from South leaving crossing\n", num);
336     ++remaining_sources;
337     pthread_mutex_unlock(&source_b);
338
339     pthread_mutex_lock(&b_queue_lock);
340     if(request_b_queue.len != 0)
341         pthread_cond_signal(&request_source_b);
342     pthread_mutex_unlock(&b_queue_lock);
343
344     return (void*)0;
345 }
346
347 void * from_east(void *arg)
348 {
349     car_num num_struct = *(car_num*)arg;
350     int num = num_struct.total_num;
351     int east_num = num_struct.direction_num;
352
353     pthread_mutex_lock(east_order_lock+east_num);
354     pthread_cond_wait(east_order+east_num, east_order_lock+east_num);
355     pthread_mutex_unlock(east_order_lock+east_num);
356
357     pthread_mutex_lock(&b_queue_lock);
358     request_b_queue.directions[request_b_queue.len] = EAST;
359     ++request_b_queue.len;
360     pthread_cond_signal(&request_source_b);
361     pthread_cond_wait(&east_wait_b, &b_queue_lock);
362     pthread_mutex_unlock(&b_queue_lock);
363
364     pthread_mutex_lock(&source_b);
365     printf("car %d from East arrives at crossing\n", num);
366
367     --remaining_sources;
368     if(remaining_sources == 0)
369     {
370         direction = EAST;
371         pthread_cond_signal(&manager_lock_cond);
372         pthread_mutex_unlock(&source_b);
373         pthread_cond_wait(&east_wait_manager, &manager_lock);
374         pthread_mutex_lock(&source_b);
375     }
```

```
376
377     pthread_mutex_lock(&c_queue_lock);
378     request_c_queue.directions[request_c_queue.len] = EAST;
379     ++request_c_queue.len;
380     pthread_cond_signal(&request_source_c);
381     pthread_cond_wait(&east_wait_c, &c_queue_lock);
382     pthread_mutex_unlock(&c_queue_lock);
383
384     pthread_mutex_lock(&source_c);
385     sleep(1);
386     pthread_mutex_unlock(&source_b);
387
388     pthread_mutex_lock(&b_queue_lock);
389     if(request_b_queue.len != 0)
390         pthread_cond_signal(&request_source_b);
391     pthread_mutex_unlock(&b_queue_lock);
392
393 //    pthread_mutex_lock(&east_manager_lock);
394     pthread_mutex_lock(east_order_lock+east_num);
395     pthread_mutex_unlock(east_order_lock+east_num);
396     pthread_cond_signal(&east_manager_cond);
397 //    pthread_mutex_unlock(&east_manager_lock);
398
399     if(remaining_sources == 0)
400         pthread_cond_signal(&manager_unlock_cond);
401
402     printf("car %d from East leaving crossing\n", num);
403     ++remaining_sources;
404     pthread_mutex_unlock(&source_c);
405
406     pthread_mutex_lock(&c_queue_lock);
407     if(request_c_queue.len != 0)
408         pthread_cond_signal(&request_source_c);
409     pthread_mutex_unlock(&c_queue_lock);
410
411     return (void*)0;
412 }
413
414 void * source_a_manager(void *arg)
415 {
416     pthread_mutex_t lock;
417     int south_level = CPL0;
418     int west_level = CPL1;
419
420     pthread_mutex_init(&lock, NULL);
421     pthread_cond_signal(&south_wait_a);
422     request_a_queue.len = 0;
423     while(1)
424     {
425         pthread_mutex_lock(&lock);
426         pthread_cond_wait(&request_source_a, &lock);
427
428         pthread_mutex_lock(&a_queue_lock);
429         if(request_a_queue.len == 1)
430         {
431             --request_a_queue.len;
```

```
432             south_level = CPL0;
433             west_level = CPL1;
434             switch(request_a_queue.directions[0])
435             {
436                 case SOUTH:
437                     pthread_cond_signal(&south_wait_a);
438                     break;
439                 case WEST:
440                     pthread_cond_signal(&west_wait_a);
441                     break;
442                 default:
443                     err_exit();
444                     break;
445             }
446         }
447         else if(request_a_queue.len == 2)
448         {
449             --request_a_queue.len;
450             if(south_level == CPL0)
451             {
452                 south_level = CPL1;
453                 west_level = CPL0;
454                 request_a_queue.directions[0] = WEST;
455                 pthread_cond_signal(&south_wait_a);
456             }
457             else
458             {
459                 south_level = CPL0;
460                 west_level = CPL1;
461                 request_a_queue.directions[0] = SOUTH;
462                 pthread_cond_signal(&west_wait_a);
463             }
464         }
465         else
466             err_exit();
467         pthread_mutex_unlock(&a_queue_lock);
468 
469         pthread_mutex_unlock(&lock);
470     }
471     return (void*)0;
472 }
473 
474 void * source_b_manager(void *arg)
475 {
476     pthread_mutex_t lock;
477     int east_level = CPL0;
478     int south_level = CPL1;
479 
480     pthread_mutex_init(&lock, NULL);
481     pthread_cond_signal(&east_wait_b);
482     request_b_queue.len = 0;
483     while(1)
484     {
485         pthread_mutex_lock(&lock);
486         pthread_cond_wait(&request_source_b, &lock);
487 
```

```
488         pthread_mutex_lock(&b_queue_lock);
489         if(request_b_queue.len == 1)
490         {
491             --request_b_queue.len;
492             east_level = CPL0;
493             south_level = CPL1;
494             switch(request_b_queue.directions[0])
495             {
496                 case SOUTH:
497                     pthread_cond_signal(&south_wait_b);
498                     break;
499                 case EAST:
500                     pthread_cond_signal(&east_wait_b);
501                     break;
502                 default:
503                     err_exit();
504                     break;
505             }
506         }
507         else if(request_b_queue.len == 2)
508         {
509             --request_b_queue.len;
510             if(east_level == CPL0)
511             {
512                 east_level = CPL1;
513                 south_level = CPL0;
514                 request_b_queue.directions[0] = SOUTH;
515                 pthread_cond_signal(&east_wait_b);
516             }
517             else
518             {
519                 east_level = CPL0;
520                 south_level = CPL1;
521                 request_b_queue.directions[0] = EAST;
522                 pthread_cond_signal(&south_wait_b);
523             }
524         }
525         else
526             err_exit();
527         pthread_mutex_unlock(&b_queue_lock);
528 
529         pthread_mutex_unlock(&lock);
530     }
531     return (void*)0;
532 }
533 
534 void * source_c_manager(void *arg)
535 {
536     pthread_mutex_t lock;
537     int north_level = CPL0;
538     int east_level = CPL1;
539 
540     pthread_mutex_init(&lock, NULL);
541     pthread_cond_signal(&north_wait_c);
542     request_c_queue.len = 0;
543     while(1)
```

```
544     {
545         pthread_mutex_lock(&lock);
546         pthread_cond_wait(&request_source_c, &lock);
547 
548         pthread_mutex_lock(&c_queue_lock);
549         if(request_c_queue.len == 1)
550         {
551             --request_c_queue.len;
552             north_level = CPL0;
553             east_level = CPL1;
554             switch(request_c_queue.directions[0])
555             {
556                 case EAST:
557                     pthread_cond_signal(&east_wait_c);
558                     break;
559                 case NORTH:
560                     pthread_cond_signal(&north_wait_c);
561                     break;
562                 default:
563                     err_exit();
564                     break;
565             }
566         }
567         else if(request_c_queue.len == 2)
568         {
569             --request_c_queue.len;
570             if(north_level == CPL0)
571             {
572                 north_level = CPL1;
573                 east_level = CPL0;
574                 request_c_queue.directions[0] = EAST;
575                 pthread_cond_signal(&north_wait_c);
576             }
577             else
578             {
579                 north_level = CPL0;
580                 east_level = CPL1;
581                 request_c_queue.directions[0] = NORTH;
582                 pthread_cond_signal(&east_wait_c);
583             }
584         }
585         else
586             err_exit();
587         pthread_mutex_unlock(&c_queue_lock);
588 
589         pthread_mutex_unlock(&lock);
590     }
591     return (void*)0;
592 }
593 
594 void * source_d_manager(void *arg)
595 {
596     pthread_mutex_t lock;
597     int west_level = CPL0;
598     int north_level = CPL1;
599 
```

```
600     pthread_mutex_init(&lock, NULL);
601     pthread_cond_signal(&west_wait_d);
602     request_d_queue.len = 0;
603     while(1)
604     {
605         pthread_mutex_lock(&lock);
606         pthread_cond_wait(&request_source_d, &lock);
607 
608         pthread_mutex_lock(&d_queue_lock);
609         if(request_d_queue.len == 1)
610         {
611             --request_d_queue.len;
612             west_level = CPL0;
613             north_level = CPL1;
614             switch (request_d_queue.directions[0])
615             {
616                 case NORTH:
617                     pthread_cond_signal(&north_wait_d);
618                     break;
619                 case WEST:
620                     pthread_cond_signal(&west_wait_d);
621                     break;
622                 default:
623                     err_exit();
624                     break;
625             }
626         }
627         else if(request_d_queue.len == 2)
628         {
629             --request_d_queue.len;
630             if(west_level == CPL0)
631             {
632                 west_level = CPL0;
633                 north_level = CPL1;
634                 request_d_queue.directions[0] = NORTH;
635                 pthread_cond_signal(&west_wait_d);
636             }
637             else
638             {
639                 west_level = CPL1;
640                 north_level = CPL0;
641                 request_d_queue.directions[0] = WEST;
642                 pthread_cond_signal(&north_wait_d);
643             }
644         }
645         else
646             err_exit();
647         pthread_mutex_unlock(&d_queue_lock);
648 
649         pthread_mutex_unlock(&lock);
650     }
651     return (void*)0;
652 }
653 
654 void * north_manager(void *arg)
655 {
```

```
656     int i;
657     sleep(1);
658     for(i = 0; i < north_num; ++i)
659     {
660 //        pthread_mutex_lock(&north_manager_lock);
661         pthread_mutex_lock(north_order_lock+i);
662         pthread_mutex_unlock(north_order_lock+i);
663         pthread_cond_signal(north_order+i);
664 
665         pthread_mutex_lock(north_order_lock+i);
666         pthread_cond_wait(&north_manager_cond, north_order_lock+i);
667         pthread_mutex_unlock(north_order_lock+i);
668 //        pthread_mutex_unlock(&north_manager_lock);
669     }
670 
671     return (void*)0;
672 }
673 
674 void * east_manager(void *arg)
675 {
676     int i;
677     sleep(1);
678     for(i = 0; i < east_num; ++i)
679     {
680 //        pthread_mutex_lock(&east_manager_lock);
681         pthread_mutex_lock(east_order_lock+i);
682         pthread_mutex_unlock(east_order_lock+i);
683         pthread_cond_signal(east_order+i);
684 
685         pthread_mutex_lock(east_order_lock+i);
686         pthread_cond_wait(&east_manager_cond, east_order_lock+i);
687         pthread_mutex_unlock(east_order_lock+i);
688 //        pthread_mutex_unlock(&east_manager_lock);
689     }
690 
691     return (void*)0;
692 }
693 
694 void * west_manager(void *arg)
695 {
696     int i;
697     sleep(1);
698     for(i = 0; i < west_num; ++i)
699     {
700 //        pthread_mutex_lock(&west_manager_lock);
701         pthread_mutex_lock(west_order_lock+i);
702         pthread_mutex_unlock(west_order_lock+i);
703         pthread_cond_signal(west_order+i);
704 
705         pthread_mutex_lock(west_order_lock+i);
706         pthread_cond_wait(&west_manager_cond, west_order_lock+i);
707         pthread_mutex_unlock(west_order_lock+i);
708 //        pthread_mutex_unlock(&west_manager_lock);
709     }
710 
711     return (void*)0;
```

```
712 }
713
714 void * south_manager(void *arg)
715 {
716     int i;
717     sleep(1);
718     for(i = 0; i < south_num; ++i)
719     {
720 //         pthread_mutex_lock(&south_manager_lock);
721         pthread_mutex_lock(south_order_lock+i);
722         pthread_mutex_unlock(south_order_lock+i);
723         pthread_cond_signal(south_order+i);
724
725         pthread_mutex_lock(south_order_lock+i);
726         pthread_cond_wait(&south_manager_cond, south_order_lock+i);
727         pthread_mutex_unlock(south_order_lock+i);
728 //         pthread_mutex_unlock(&south_manager_lock);
729     }
730
731     return (void*)0;
732 }
733
734 void * manager(void *arg)
735 {
736     while(1)
737     {
738         pthread_mutex_lock(&manager_lock);
739         pthread_cond_wait(&manager_lock_cond, &manager_lock);
740         switch(direction)
741         {
742             case NORTH:
743                 printf("DEADLOCK: car jam detected, signalling East to go\n");
744                 break;
745             case EAST:
746                 printf("DEADLOCK: car jam detected, signalling South to go\n");
747                 break;
748             case WEST:
749                 printf("DEADLOCK: car jam detected, signalling North to go\n");
750                 break;
751             case SOUTH:
752                 printf("DEADLOCK: car jam detected, signalling West to go\n");
753                 break;
754             default:
755                 err_exit();
756                 break;
757         }
758         pthread_cond_wait(&manager_unlock_cond, &manager_lock);
759         switch(direction)
760         {
761             case NORTH:
762                 pthread_cond_signal(&north_wait_manager);
763                 break;
764             case EAST:
765                 pthread_cond_signal(&east_wait_manager);
766                 break;
767             case WEST:
```

```
768                 pthread_cond_signal(&west_wait_manager);
769                 break;
770             case SOUTH:
771                 pthread_cond_signal(&south_wait_manager);
772                 break;
773             default:
774                 err_exit();
775                 break;
776         }
777         pthread_mutex_unlock(&manager_lock);
778     }
779     return (void*)0;
780 }
781
782 int main(int argc, char ** argv)
783 {
784     int i = 0;
785     int err;
786     pthread_t manager_tid;
787     pthread_t north_manager_tid;
788     pthread_t east_manager_tid;
789     pthread_t west_manager_tid;
790     pthread_t south_manager_tid;
791     pthread_t source_a_manager_tid;
792     pthread_t source_b_manager_tid;
793     pthread_t source_c_manager_tid;
794     pthread_t source_d_manager_tid;
795     len = strlen(argv[1]);
796
797     north_num = 0;
798     east_num = 0;
799     west_num = 0;
800     south_num = 0;
801
802     request_a_queue.len = 0;
803     request_a_queue.directions[0] = 0;
804     request_a_queue.directions[1] = 0;
805     request_b_queue.len = 0;
806     request_b_queue.directions[0] = 0;
807     request_b_queue.directions[1] = 0;
808     request_c_queue.len = 0;
809     request_c_queue.directions[0] = 0;
810     request_c_queue.directions[1] = 0;
811     request_d_queue.len = 0;
812     request_d_queue.directions[0] = 0;
813     request_d_queue.directions[1] = 0;
814
815     char* cars = (char*)malloc((len+1) * sizeof(char));
816     pthread_t * tids = (pthread_t*)malloc((len+1) * sizeof(pthread_t));
817 //    int* nums = (int*)malloc(len * sizeof(int));
818     car_num* car_nums = (car_num*)malloc(len * sizeof(car_num));
819
820     remaining_sources = 4;
821
822     pthread_mutex_init(&manager_lock, NULL);
823     pthread_mutex_init(&count_lock, NULL);
```

```
824     pthread_mutex_init(&remaining_lock, NULL);
825 
826     pthread_mutex_init(&source_a, NULL);
827     pthread_mutex_init(&source_b, NULL);
828     pthread_mutex_init(&source_c, NULL);
829     pthread_mutex_init(&source_d, NULL);
830 
831     pthread_mutex_init(&first_north_lock, NULL);
832     pthread_mutex_init(&first_east_lock, NULL);
833     pthread_mutex_init(&first_west_lock, NULL);
834     pthread_mutex_init(&first_south_lock, NULL);
835 
836     pthread_mutex_init(&north_manager_lock, NULL);
837     pthread_mutex_init(&east_manager_lock, NULL);
838     pthread_mutex_init(&west_manager_lock, NULL);
839     pthread_mutex_init(&south_manager_lock, NULL);
840 
841     pthread_mutex_init(&a_queue_lock, NULL);
842     pthread_mutex_init(&b_queue_lock, NULL);
843     pthread_mutex_init(&c_queue_lock, NULL);
844     pthread_mutex_init(&d_queue_lock, NULL);
845 
846     pthread_cond_init(&manager_lock_cond, NULL);
847     pthread_cond_init(&manager_unlock_cond, NULL);
848     pthread_cond_init(&north_wait_manager, NULL);
849     pthread_cond_init(&east_wait_manager, NULL);
850     pthread_cond_init(&west_wait_manager, NULL);
851     pthread_cond_init(&south_wait_manager, NULL);
852 
853     pthread_cond_init(&first_north, NULL);
854     pthread_cond_init(&first_east, NULL);
855     pthread_cond_init(&first_south, NULL);
856     pthread_cond_init(&first_west, NULL);
857 
858     pthread_cond_init(&north_manager_cond, NULL);
859     pthread_cond_init(&east_manager_cond, NULL);
860     pthread_cond_init(&west_manager_cond, NULL);
861     pthread_cond_init(&south_manager_cond, NULL);
862 
863     pthread_cond_init(&request_source_a, NULL);
864     pthread_cond_init(&request_source_b, NULL);
865     pthread_cond_init(&request_source_c, NULL);
866     pthread_cond_init(&request_source_d, NULL);
867 
868     pthread_cond_init(&south_wait_a, NULL);
869     pthread_cond_init(&south_wait_b, NULL);
870     pthread_cond_init(&east_wait_b, NULL);
871     pthread_cond_init(&east_wait_c, NULL);
872     pthread_cond_init(&north_wait_c, NULL);
873     pthread_cond_init(&north_wait_d, NULL);
874     pthread_cond_init(&west_wait_d, NULL);
875     pthread_cond_init(&west_wait_a, NULL);
876 
877     for(i = 0; i < len; ++i)
878     {
879         cars[i] = argv[1][i];
```

```
880 //          nums[i] = i + 1;
881         car_nums[i].total_num = i + 1;
882         switch(cars[i])
883         {
884             case 'n':
885                 car_nums[i].direction_num = north_num;
886                 ++north_num;
887                 break;
888             case 'e':
889                 car_nums[i].direction_num = east_num;
890                 ++east_num;
891                 break;
892             case 'w':
893                 car_nums[i].direction_num = west_num;
894                 ++west_num;
895                 break;
896             case 's':
897                 car_nums[i].direction_num = south_num;
898                 ++south_num;
899                 break;
900             default:
901                 err_exit();
902                 break;
903         }
904     }
905     cars[i] = '\0';
906 
907     north_order = (pthread_cond_t*)malloc(north_num * sizeof(pthread_cond_t));
908     east_order = (pthread_cond_t*)malloc(east_num * sizeof(pthread_cond_t));
909     west_order = (pthread_cond_t*)malloc(west_num * sizeof(pthread_cond_t));
910     south_order = (pthread_cond_t*)malloc(south_num * sizeof(pthread_cond_t));
911 
912     north_order_lock = (pthread_mutex_t*)malloc(north_num * sizeof(pthread_mutex_t)
        );
913     east_order_lock = (pthread_mutex_t*)malloc(east_num * sizeof(pthread_mutex_t));
914     west_order_lock = (pthread_mutex_t*)malloc(west_num * sizeof(pthread_mutex_t));
915     south_order_lock = (pthread_mutex_t*)malloc(south_num * sizeof(pthread_mutex_t)
        );
916 
917     for(i = 0; i < north_num; ++i)
918     {
919         pthread_cond_init(north_order+i, NULL);
920         pthread_mutex_init(north_order_lock+i, NULL);
921     }
922 
923     for(i = 0; i < east_num; ++i)
924     {
925         pthread_cond_init(east_order+i, NULL);
926         pthread_mutex_init(east_order_lock+i, NULL);
927     }
928 
929     for(i = 0; i < south_num; ++i)
930     {
931         pthread_cond_init(south_order+i, NULL);
932         pthread_mutex_init(south_order_lock+i, NULL);
933     }
```

```
934
935     for(i = 0; i < west_num; ++i)
936     {
937         pthread_cond_init(west_order+i, NULL);
938         pthread_mutex_init(west_order_lock+i, NULL);
939     }
940
941     err = pthread_create(&manager_tid, NULL, manager, NULL);
942     if(err != 0)
943         err_exit();
944
945     for(i = 0; i < len; ++i)
946     {
947         switch(cars[i])
948         {
949             case 'n':
950                 err = pthread_create(tids+i, NULL, from_north, (void*)(car_nums + i
                        ));
951                 if(err != 0)
952                     err_exit();
953                 break;
954             case 'w':
955                 err = pthread_create(tids+i, NULL, from_west, (void*)(car_nums + i)
                        );
956                 if(err != 0)
957                     err_exit();
958                 break;
959             case 's':
960                 err = pthread_create(tids+i, NULL, from_south, (void*)(car_nums + i
                        ));
961                 if(err != 0)
962                     err_exit();
963                 break;
964             case 'e':
965                 err = pthread_create(tids+i, NULL, from_east, (void*)(car_nums + i)
                        );
966                 if(err != 0)
967                     err_exit();
968                 break;
969             default:
970                 err_exit();
971                 break;
972         }
973     }
974
975     if(0 != pthread_create(&north_manager_tid, NULL, north_manager, NULL))
976         err_exit();
977     if(0 != pthread_create(&east_manager_tid, NULL, east_manager, NULL))
978         err_exit();
979     if(0 != pthread_create(&west_manager_tid, NULL, west_manager, NULL))
980         err_exit();
981     if(0 != pthread_create(&south_manager_tid, NULL, south_manager, NULL))
982         err_exit();
983
984     if(0 != pthread_create(&source_a_manager_tid, NULL, source_a_manager, NULL))
985         err_exit();
```

```
986     if(0 != pthread_create(&source_b_manager_tid, NULL, source_b_manager, NULL))
987         err_exit();
988     if(0 != pthread_create(&source_c_manager_tid, NULL, source_c_manager, NULL))
989         err_exit();
990     if(0 != pthread_create(&source_d_manager_tid, NULL, source_d_manager, NULL))
991         err_exit();
992 
993     for(i = 0; i < len; ++i)
994     {
995         err = pthread_join(tids[i], NULL);
996         if(err != 0)
997             err_exit();
998     }
999 
1000    return 0;
1001 }
```