1. (a) 在流水线处理器中，什么是冒险(hazard)？在什么情况下出现冒险？对于冒险一般如何处理？

(b) 什么是 CISC？什么是 RISC？它们各自有什么特点和区别？

(c) 什么是虚拟地址？虚拟地址有什么作用？

(d) 解释下列名词：大端（big endian）、小端(little endian)、主机序、网络序

(a) CISC 是"复杂指令集"的简称，复杂指令集指令数量很多，部分指令的延迟很长，编码是可变长度的。RISC 是"精简指令集"的简称，精简指令集指令数量少得多，没有较长延迟的指令，编码是固定长度的。

CISC	早期的 RISC
指令数量很多。Intel 描述全套指令的文档 [28，29] 有 1200 多页。	指令数量少得多。通常少于 100 个。
有些指令的延迟很长。包括将一个整块从存储器的一个部分复制到另一部分的指令，以及其他一些将多个寄存器的值复制到存储器或从存储器复制到多个寄存器的指令。	没有较长延迟的指令。有些早期的 RISC 机器甚至没有整数乘法指令，要求编译器通过一系列加法来实现乘法。
编码是可变长度的。IA32 的指令长度可以是 1 ～ 15 个字节。	编码是固定长度的。通常所有的指令都编码为 4 个字节。
指定操作数的方式很多样。在 IA32 中，存储器操作数指示符可以有许多不同的组合，这些组合由偏移量、基址和变址寄存器以及伸缩因子组成。	简单寻址方式。通常只有基址和偏移量寻址。
可以对存储器和寄存器操作数进行算术和逻辑运算。	只能对寄存器操作数进行算术和逻辑运算。允许使用存储器引用的只有 load 和 store 指令，load 是从存储器读到寄存器，store 是从寄存器写到存储器。这种方法被称为 load/store 体系结构。
对机器级程序来说实现细节是不可见的。ISA 提供了程序和如何执行程序之间的清晰的抽象。	对机器级程序来说实现细节是可见的。有些 RISC 机器禁止某些特殊的指令序列，而有些跳转要到下一条指令执行完了以后才会生效。编译器必须在这些约束条件下进行性能优化。
有条件码。作为指令执行的副产品，设置了一些特殊的标志位，可以用于条件分支检测。	没有条件码。相反，对条件检测来说，要用明确的测试指令，这些指令会将测试结果放在一个普通的寄存器中。
栈密集的过程链接。栈被用来存取过程参数和返回地址。	寄存器密集的过程链接。寄存器被用来存取过程参数和返回地址。因此有些过程能完全避免存储器引用。通常处理器有更多的（最多的有 32 个）寄存器。

(b) 虚拟地址是操作系统提供的对主存的抽象，它为每个进程提供了一个大的、一致的和私有的地址空间。它的作用是 1) 它将主存看成一个存储在磁盘上的地址空间的高速缓存，在主存中只保存活动区域，并根据需要在主存和磁盘之间来回传送数据，通过这种方式，它高效的利用了主存 2）它为每个进程提供了一致的地址空间，从而简化了存储器管理 3）它保护了每个进程的地址空间不被其它进程破坏。

(c) 最低有效字节在最前面的方式，称为小端法，最低有效字节在最后面的方式，称为大端法。主机序是指机器 CPU 采用的字节表示方法。网络序是 TCP/IP 中规定的数据表示格式，与 CPU 类型和操作系统无关，主机序采用大端排列。

2. 有如下假设：

(1) 存储器是字节寻址的；

(2) 存储器访问的是 1 字节的字（不是 4 字节的字）；

(3) 地址宽度为 13 位；

(4) 高速缓存是 2 路组相联的（E=2），块大小为 4 字节（B=4），有 8 个组（S=8）；

高速缓存的内容如下，所有数字都是以十六进制来表示的：

2 路组相联高速缓存

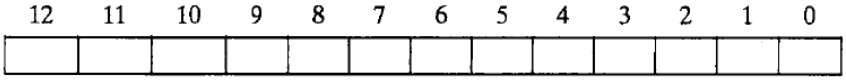
组索引	行 0						行 1					
	标记位	有效位	字节 0	字节 1	字节 2	字节 3	标记位	有效位	字节 0	字节 1	字节 2	字节 3
0	09	1	86	30	3F	10	00	0	—	—	—	—
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	—	—	—	—	0B	0	—	—	—	—
3	06	0	—	—	—	—	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	—	—	—	—
6	91	1	A0	B7	26	2D	F0	0	—	—	—	—
7	46	0	—	—	—	—	DE	1	12	C0	88	37

（a）下面的放开展示的是地址格式（每个小方框一个位）。指出（在图中标出）用来确定下列内容的字段：

（1） CO 高速缓存块偏移

（2） CI 高速缓存组索引

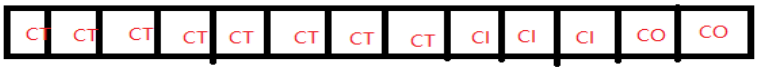
（3） CT 高速缓存标记



（b）假设一个程序运行在上述机器上，它引用 0xE34 处的 1 个字节的字。指出访问的高速缓存条目和十六进制表示的返回的高速缓存字节值。指出是否会发生缓存不命中，如果出现缓存不命中，用“-”来表示“返回的高速缓存字节”。

参数	值
高速缓存块偏移（CO）	0x_____
高速缓存组索引（CI）	0x_____
高速缓存标记（CT）	0x_____
高速缓存命中？（是/否）	
返回的高速缓存字节	0x_____

(a) 由于块大小是 4(2^2)，那么需要两位来表示块内偏移；由于缓存一共有 8(2^3)组，那么需要 3 位来表示组索引；由于地址一共 13 位，那么标记位为 8(13 – 2 - 3)位；按照高速缓存地址划分的规则（从高位到低位依次是标记，组索引，块内偏移），13 位地址可标记为：

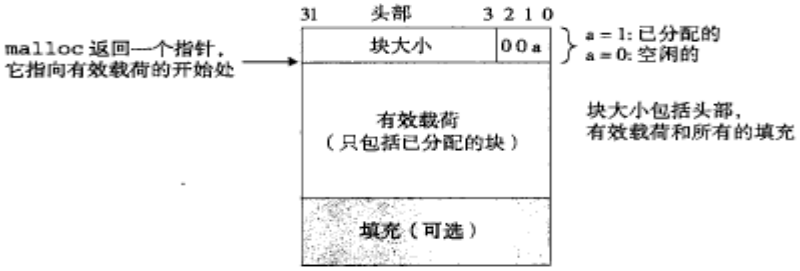


(b) 地址 0xE34 的二进制表示为：0b111000110100；根据上面的讨论，末两位为块内偏移 0b00(0x0)，末两位之前三位为组索引 0b101(0x5)，末五位之前是标记 0b1110001(0x71)；通过查表可以找到组号为 5，标记为 0x71 的位置，该位置有效位为 1，读取第一个字节得到数据 0xB。

3. （a） 确定下面的 malloc 请求序列得到的块大小和头部值。假设：

（1） 分配器维持双字对齐，使用隐式空闲链表，格式如下图所示。

（2） 块大小向上摄入为最接近的 8 字节的倍数。



请求	块大小（十进制字节）	块头部（十六进制）
malloc(3)		
malloc(11)		
malloc(20)		
malloc(21)		

(b) 确定下面每种对齐要求和块格式的组合的最小块大小。假设：隐式空闲链表，不允许有效载荷为零，头部和脚部放在 4 字节的字中。

对齐要求	已分配的块	空闲块	最小块大小（字节）
单字	头部和脚部	头部和脚部	
单字	头部，但是无脚部	头部和脚部	
双字	头部和脚部	头部和脚部	
双字	头部，但是没有脚部	头部和脚部	

（a） 块大小包括块头部大小（如果块有尾部的话，那么也包含尾部大小），当请求 3 字节大小的块时，需要 3+4=7 字节的内容，而根据条件的 8 字节倍数要求，块的大小确定为 8 字节（最后一字节虽然没有申请，但也分配了）；根据图片显示的块结构，块头部值的最后三位为元数据，元数据头两位始终为 0，最后一位为 1 表示已占用的块；块头部值可计算为块大小加上块元数据值，即 0x8+0b001，即头部值为 9。

根据上面的方法，malloc(11)需要分配 16 字节的块，头部值为 17(0x11) ;malloc(20)需要分配 24 字节的块，头部值为 25（0x19）;malloc(21)需要分配 32 字节的块，头部值为 33(0x21)

(c) 第一项，块对齐单字，已分配和空闲块都有头部和尾部，那么头部和尾部数据需要 8 字节；而有效载荷不允许为 0，那么大于 8 且能被 4 整除的最小正整数为 12，因此最小块大小为 12。

第二项，块对齐单字，已分配块可以省略脚部，那么仅考虑已分配块，头部数据需要 4 个字节；有小载荷不允许为 0，那么大于 4 且能被 4 整除的最小正整数为 8，因此最小块大小为 8。

第三项，大于 8 且能被 8 整除的最小整数为 16。

第四项，大于 4 且能被 8 整除的最小整数为 8。

4. (a) 下面是一个并行求和算法，将其补充完整。

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

#define MAXTHREADS 32

/* Global shared variables */
long psum[MAXTHREADS];
long nelems_per_thread;

void *sum(void *vargp)
{
    /* Extract the thread id */
    int myid = *((int *)vargp);
    /* Start element index */
    long start = myid * nelems_per_thread;
    /* End element index */
    long end = start + nelems_per_thread;

    long i, sum = 0;
    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[myid] = sum;

    return NULL;
}

int main(int argc, char **argv)
{
    long i, nelems, log_nelems, nthreads, result = 0;
    pthread_t tid[MAXTHREADS];
    int myid[MAXTHREADS];

    /* Get input arguments */
    if (argc != 3) {
        printf("Usage:%s <nthreads> <log_nelems>\n", argv[0]);
        exit(0);
    }
    nthreads = atoi(argv[1]);
    log_nelems = atoi(argv[2]);
    nelems = (1L << log_nelems);
```

```
    nelems_per_thread = nelems / nthreads;

    /* Create peer threads and wait for them to finish */
    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        ___(1)___;
    }
    for (i = 0; i < nthreads; i++) {
        ___(2)___;
    }

    /* Add up the partial sums computed by each thread */
    for (i = 0; i < nthreads; i++) {
        ____(3)____;
    }

    /* Check final answer */
    if (result != (nelems *(nelems - 1))/2) {
        printf("Error: result=%ld\n", result);
    }

    exit(0);
}
```

(b) 思考下面的程序，它视图使用一对信号量来实现互斥。

初始时： s = 1, t = 0

线程 1：P(s)→V(s)→P(t)→V(t)

线程 2：P(s)→V(s)→P(t)→*(t)

(1) 画出这个程序的进度图。

(2) 它总是死锁吗？请分析原因

(3) 如果是，那么对初始信号量的值做那些改变就能消除潜在的死锁呢？

(4) 画出得到的无死锁程序的进度条。

(a) (1) pthread_create(&tid[i], NULL, sum, &myid[i]);

(2) pthread_join(tid[i], NULL);

(3) result += psum[i];

(b) (1)进度图如下

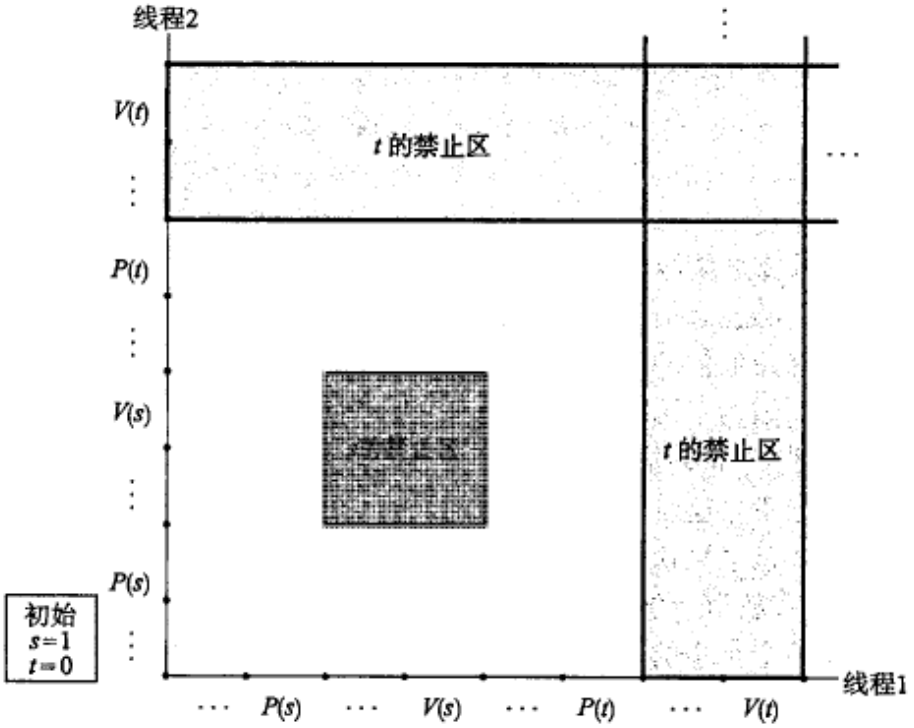


图 12-46　一个有死锁的程序进度图

（2）因为任何可行轨迹最终都将陷入死锁状态，所以这个程序总会发生死锁。

（3）　为了消除死锁，可以将变量 t 初始化为 1。

（4）修正后的进度图如下。

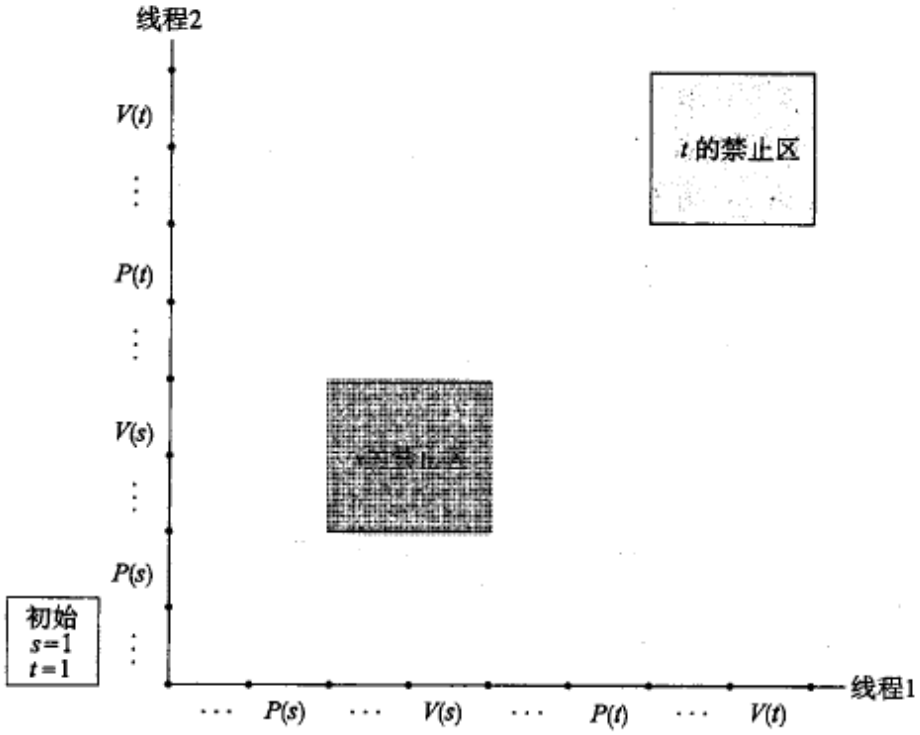


图 12-47　改正后的无死锁的程序的进度图