1.请阅读以下代码

```
int main(){
    int x = 0;
    float y = 0.0;
    while((y - x < 1) && (y - x > -1)){
        x += 1;
        y += 1;
    }
    printf("%d %f\n", x, y);
}
```

问最终的输出为：

（提示 1：我们这里认为 int 型变量与 float 型变量运算会把 int 型转化为 float 型再进行运算）

（提示 2：2^24 = 16777216）

（提示 3：本题中的舍入方式为向偶数舍入）

A.16777218 16777217.000000

B.16777218 16777216.000000

C.16777217 16777216.000000

D.16777217 16777215.000000

2.现定义了如下变量：

```
int x, y, z;
float fx, fy, fz;
```

则以下表达式不恒为真的是（其中 fx,fy 均不为无穷或非数）：

A.x * y == y * x

B.(x + y) * z == x * z + y * z

C.fx + fy == fy + fx

D.(fx + fy) * fz == fx * fz + fy * fz

3.下列选项中是合法 x86-64 汇编指令的是（）

A.movq %rax, $1

B. xorq %rsp, %rsp

C. movq (%rax), (%rbx)
D. movq (%rax, %rbx, 3), %rcx

4. 现假定所有函数均使用%rbp 寄存器描述栈帧，并且函数体进入时一定以 pushq %rbp; movq %rsp, %rbp 开始，函数体退出前一定以 leaveq; retq 结束。有编译器内置函数 __builtin_return_address(1)表示获取调用当前函数的函数的返回地址，举例来说，若有如下代码：

```
long foo(void)
{
        return (long) __builtin_return_address(1);
}
```

同时函数 bar()调用了函数 foo()，则函数 foo()的返回值是函数 bar()的返回地址(即 bar()调用 完成后控制流转向的地址)，保证 bar()和 foo()都不是内联函数，则将函数 foo()编译，生成的 汇编指令可能是（）

A.

```
foo:
        pushq %rbp
        movq %rsp, %rbp
        movq 8(%rbp), %rax
        leaveq
        retq
```

B.

```
foo:
        pushq %rbp
        movq %rsp, %rbp
        movq 0(%rbp), %rax
        movq 0(%rax), %rax
        leaveq
        retq
```

C.

```
foo:
        pushq %rbp
        movq %rsp, %rbp
        movq 0(%rbp), %rax
```

```
        movq -8(%rax), %rax
        leaveq
        retq
```

D.

```
foo:
        pushq %rbp
        movq %rsp, %rbp
        movq 0(%rbp), %rax
        movq 8(%rax), %rax
        leaveq
        retq
```

5.下列关于 RISC 和 CISC 的描述中，正确的是：

A.在 RISC 指令集的发展过程中，其指令数量始终少于 100 条。

B.CISC 指令集中的指令比 RISC 指令集更复杂，因此其指令长度总是比 RISC 指令集中的指令要长。

C.早期的 RISC 指令集没有条件码，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。

D.RISC 指令集中的所有过程都需要进行内存引用。

6.在 Y86-64 PIPE 处理器中（不考虑数据前递），以下哪个指令序列会造成数据冒险？

A.

```
irmovq $10, %rdx
addq %rdx, %rax
```

B.

```
irmovq $10, %rdx
nop
addq %rdx, %rax
```

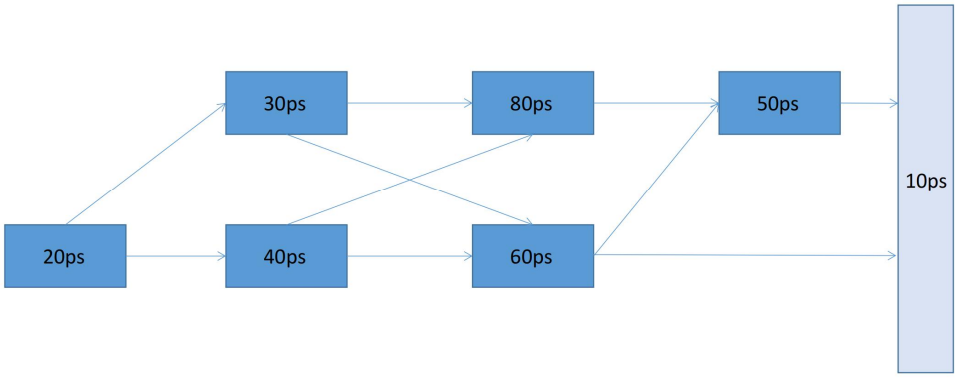
C.

```
irmovq $10, %rdx
nop
nop
```

addq %rdx, %rax

D. 以上三个选项都会引发数据冒险

7.根据下图所示的流水线结构，判断错误的选项是?



A.当前流水线的延迟为(20ps + 40ps + 80ps + 50ps + 10ps)=200ps，吞吐率为 5GIPS。
B.无法通过只插入一个寄存器的方式来使这个流水线变为二级流水线。
C.倘若只能插入两个寄存器(延迟均为 10ps)，那么该流水线处理每条指令的平均时间最多减少 20ps (假设每条指令平均使用一个周期的时间)。
D.流水线的级数越深，处理器就一定能获得越好的性能。

8.以下关于程序设计优化的说法，错误的是:
A.循环展开有助于进一步变换代码，减少关键路径上的操作数量。
B.分离多个累积变量计算可以提高并行性。
C.大多数编译器会改变浮点数的合并运算（如加法和乘法）顺序以提高程序性能。
D.循环展开的程度增加并不一定能改善程序运行效率，反而会变差。

9.以下关于存储设备与存储器层次结构的说法，正确的是:
A.一个有 5 个盘片，每个面 10000 条磁道，每条磁道平均有 400 个扇区，每个扇区有 512 个 字 节的硬盘容量为 20.48GB。

B.SRAM 的存取速度快，但是断电后数据会丢失，DRAM 的存取速度较慢，但断电后数据不会丢失。

C.在存储器层次结构中，上一层存储的信息一定是下一层存储的信息的一个子集。

D.基于缓存的存储器层次结构只利用了程序的时间局部性，没有利用程序的空间局部性，因为缓存不会存储我们没访问过的元素。

10. 布局是数字芯片设计自动化流程的必要步骤，下面哪一项不是布局算法需要优化的目标?

A. 互连线长（Wirelength）
B. 可布线性（Routability）
C. 时序（Timing）
D. 逻辑深度（Logic depth）

答案与题解：

1.答案：B。由代码可以知道，循环会在 x 与 y 的值首次不相等的时候终止，这当然会在 y 首次发生舍入附近。由于当 y 达到值 16777216 后就不会发生变化，因此之后就只考察 x 的变化。x 在值为 16777217 时类型转换为 float 会向偶数舍入至 16777216，因此最终 x 的值为 16777218。

2.答案：D。大数吃小数。

3.答案：B。立即数不能作为 mov 指令的目标，A 错误；mov 指令无法操作两次内存，C 错误；内存寻址比例因子必须是 1、2、4、8，D 错误。

4.答案：D。在 foo()中读取 0(%rbp)可以获得 foo()开头保存的%rbp，也就是外层函数 bar()的栈帧地址，向上偏移 8 即可获得函数 bar()的返回地址。

5.答案：C

解析：A 选项，RISC 指令集有复杂化的趋势，早期 RISC 通常指令数量小于 100 条；B，CISC 的指令是不定长的，有些长度比 RISC 短；C 正确；D，CISC 一定会将返回地址压入栈中，不可能避免内存引用。

本题考查对两种指令集的理解。

6.答案：D。要想不引起数据冒险，需要改变数据的指令在使用数据的指令进入译码阶段时已经完成写回阶段。因此至少需要插入 3 个 nop 指令。

书本 P295

7.答案为 D。送分。

8.答案：C。浮点数运算没有结合律，大多数编译器不会尝试优化。

9.答案：A。B 中 DRAM 掉电后也会丢失，C 中上层存储的内容可以独立于下层，例如加载到寄存器中的值不一定在内存中存在。D 缓存利用了空间局部性，会把访问元素在同一块内的元素全部取到缓存里。

10．答案为 D，见讲座 PPT25 页，凭直觉也可以获得答案：布局算法会影响互连线长和走线方式，因此 A 和 B 需要考虑，线长会影响延时，所以 C 时序也需要考虑。D 的逻辑深度在布局时不变。因此正确答案选 D

第二题 请结合教材第六章"存储器层次结构"的有关知识回答问题**(10 分)**.

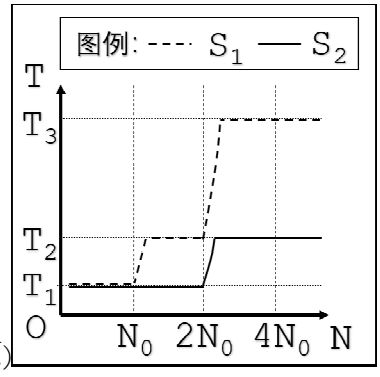
1. (2 分)高速缓存(cache)的先进先出替换策略(FIFO)指的是在发生缓存不命中时,最先进入高速缓存的行(cache line)将最先被替换出.考虑两种遵循先进先出替换策略且块大小(block size)相同的全相联高速缓存 C1 和 C2.其中 C1 是 3 路全相联的,C2 是 4 路全相联的.假设初始情况下 C1,C2 所有行的有效位都为 0.则让它们分别连续访问标记(tag)为 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4 的行之后, C1 发生的缓存不命中次数**(1)**_____C2 发生的缓存不命中次数**(填">""="或"<")**.

2. ((2)(3)(4)每空 2 分,(5)(6)每空 1 分)已知某单核处理器有 L1,L2,L3 三级高速缓存且遵循先进先出替换策略,你希望通过触发行驱逐(cache line eviction)的方法测量该处理器的 L1 数据缓存(L1 d-cache)的相联度,实验前已知 L1 d-cache 的容量在 16KiB 到 64KiB 之间(含 16KiB 和 64KiB)且字节数为 2 的幂,且已知高速缓存块的字节数小于 1KiB.你的实验步骤如下:

Step #1: 开辟一块足够大(如 8MiB)的内存空间.
Step #2: 逐步调整访问的步长 S(S 依次取 1KiB, 2KiB, 4KiB, …, 32KiB,64KiB)和访问的元素数量 N(N 依次取 1, 2, 3, …, 127, 128).对于给定的 S 和 N,连续两次从所开辟内存的起始位置开始,以 S 为步长顺序访问 N 个内存地址.记录第二次访问时平均一个内存地址的访问时间 T.
Step #3: 多次实验取平均值以减小误差.作图并分析实验结果.
这里是给定 S 和 N 后访问内存的一个例子:当 S=64KiB,N=6 时,先按顺序访问一遍所开辟内存空间的第 0B, 64KiB, 128KiB, 192KiB, 256KiB, 320KiB 处的一个字节的值.清空寄存器,接着计时并第二遍按顺序访问所开辟内存空间的第 0B, 64KiB, 128KiB, 192KiB, 256KiB, 320KiB 处的一个字节的值.访问完成后立即停止计时,并记录 T = (结束计时的时刻 - 开始计时的时刻) ÷ 6.0.

(a) **Step #2** 中,给定 S 和 N 后第一遍访问 N 个内存地址的过程被称为暖身(warm up),这是为了避免教材中所提到的三种缓存不命中之一的**(2)**_____不命中带来的影响.

(b) 假设访问内存的过程都在高速缓存中顺次进行,且不考虑预取(prefetch)机制的影响.你选取了步长为 S_1, S_2 时的部分实验结果如图所示(此图为示意图).你注意到访存速度越快,则访问单个元素的时间越短.因此从图中可以看出 T_3 最接近**(3)**_____**(填"L1""L2""L3")**级缓存的访存时间,S_1 **(4)**_____S_2**(填">""=""或**



"<")。

(c) 在(b)的条件下，进一步推理可知该处理器的 L1 d-cache 的容量(capacity)为(5)_____(用代数式表示，S_1，S_2，N_0 为已知量，下同。注意计算容量时不考虑有效位和标记位)，相联度为(6)_____。

标准答案(每空 2 分)：

(1) <
(2) 冷/强制性
(3) L3
(4) >
(5) $2 \cdot N_0 \cdot S_2$
(6) N_0

评分标准：

(1) 写"小于"的酌情扣 1 分，回答其他答案（如≤）的不得分。
(2) 写"冷""强制性"也正确。写英文也正确。字写错或有拼写错误的，若不影响意思表示不额外扣分。
(3) 写"L_3"也正确，写"三级"的酌情扣一分。
(4) 写"大于"的酌情扣 1 分。
(5) 角标书写不规范，如写成"2 ·N0 ·S2"，若不影响意思表示不额外扣分。
(6) 角标书写不规范，如写成"N0"，若不影响意思表示不额外扣分。

解析：本题设简单题 8 分，难题 2 分。预测平均分 7.5 分。

(1) 考察高速缓存替换模拟，属简单题，预测正确率 90%。应当注意 FIFO 和 LRU 替换机制的区别。
(2) 考察缓存不命中的类型，属简单题，预测正确率 90%。
(3) 考察读图分析数据的能力，属简单题，预测正确率 90%。因为 L3 高速缓存访存速度最慢，因此平均访问时间最长。
(4) 考察读图分析数据的能力，属简单题，预测正确率 80%。由于 $N=2N_0$ 时 S_1 发生了 L2 到 L3 高速缓存的访存切换，S_2 发生了 L1 到 L2 高速缓存的访存切换，故有 $S_1 > S_2$。
(5) 考察对多路组相联高

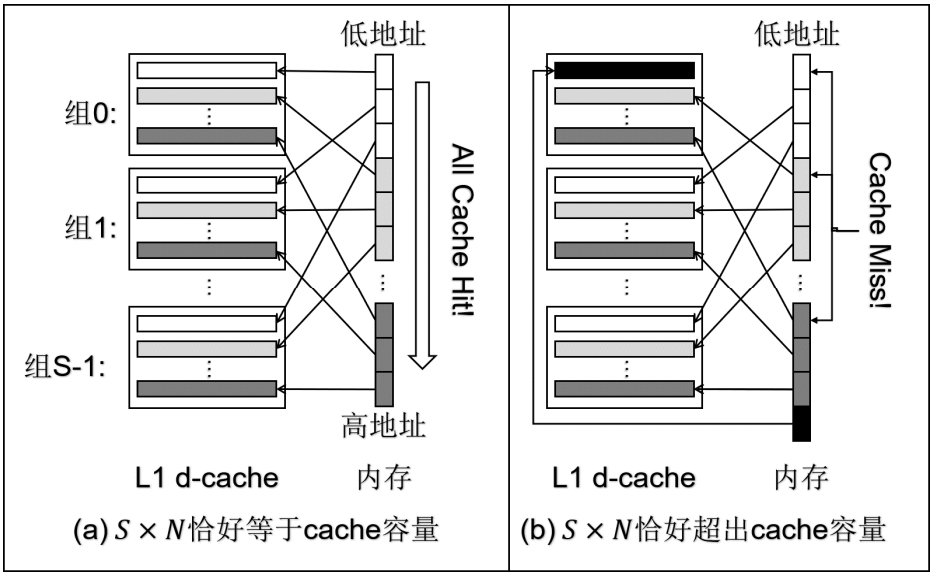


图 2-2

速缓存替换原理的理解，属难题，预测正确率 25%.

(i) 虽然步长 S 是从 1KiB 开始测量的，但我们不妨先考虑步长更小，即 S 恰好与 L1 d-cache 缓存块大小相等的情形，如图 2-2(a)所示. 不失一般性，我们假设所开辟内存空间的起始地址对应到 L1 d-cache 的组 0. 当$S \times N$恰好为 L1 d-cache 的容量时，L1 d-cache 恰好缓存了所有需要访问的值. 此时，所有的访存都是在 L1 d-cache 中进行的. 现在保持 S 不变，把 N 调整为 N+1. 如图 2-2(b)所示. 此时 L1 d-cache 不得不在组 0 中替换掉一行. 根据先进先出的替换原则，被换出的行恰好是第一个被加载进 L1 d-cache 的行. 所以，在第二遍访问所开辟内存空间的起始地址时，会遇到一次缓存不命中. 而加载该地址的过程又会替换掉第一遍访问时第二个被加载进组 0 的行，使第二遍访问它时也是不命中的. 以此类推，你发现访问所有对应到组 0 的地址都是不命中的. 频繁的 L2 cache 访存导致平均访存时间的剧烈上升.

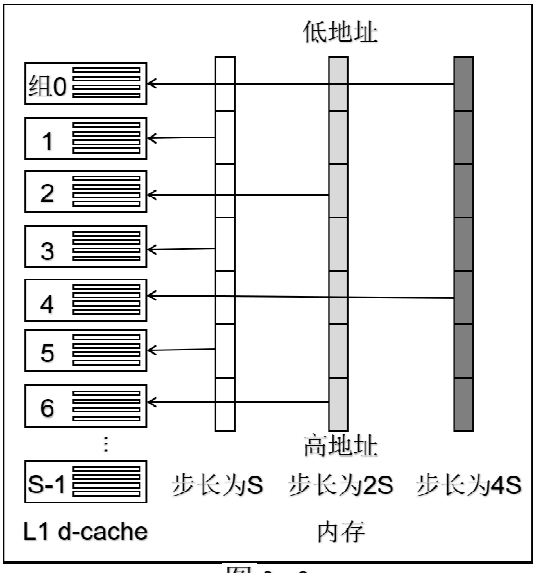


图 2-3

(ii) 再考虑$S \times N$保持不变，步长从 S 变成 2S，4S……时的情形. 如图 2-3 所示，你发现当步长翻倍时，虽然访存次数减少到原来的一半，但是 L1 d-cache 缓存的有效的值的数量可能也会缩小到原来的一半. 这使得平均访存时间可能是不变的.

假想 S 从 B 逐步增长到了 S_2，而$N=2N_0$处平均访存时间发生剧烈的上升，这说明 L1 d-cache 的容量为 $2 \cdot N_0 \cdot S_2$.

(6) 考察多路组相联高速缓存相联度的性质，属难题，预测正确率 25%. 如果高速缓存的相联度不为 1，那么转折是直接映射的，那么步长每增加一倍，则访存时间 T 的转折点横坐标应该变为原来的 1/2，但如果高速缓存的相联度不为 1，那么转折点横坐标在到达相联度的时候就不会再减小. 从图中可以看出，步长 S_1 至少是 S_2 的 4 倍（因为 $S=S_2, N=4N_0$ 时，平均访存时间 T 没有发生剧烈变化）. 但 S_1 的从 T_1 到 T_2 的转折点横坐标只是 S_2 转折点横坐标的 1/2. 这说明 L1 d-cache 的相联度为 N_0.

命题人：李浩雨，陈奕奇

2021 年 12 月 4 日

ICS 期末出题-第七章

（Linux 工具链）简单，基础 1-2 min

1. 以下关于 Linux 系统上处理可执行文件的工具的说法<u>不正确</u>的是
   A. 使用 objdump 反汇编 .text 节的机器码
   B. 使用 readelf 读取文件的节头部头(section header)和程序头部表(program header)
   C. 使用 ls 查询文件是文本文件还是二进制文件
   D. 使用 gdb 加载可执行文件、设断点，然后单步调试运行

C 错误

解析：

A 正确。objdump -dj .text [file]

B 正确。节头部表：readelf -S [file] 程序头部表：readelf -l [file]

C 错误。ls 只是取得文件的元数据，这与文件内容无关，而 linux 文件元数据中也不包含对 binary 或者 text 编码属性的描述。其他诸如 grep 和 file 的工具使用 heuristics 确定文件的类型。

D 正确。gdb 可以支持单步调试。

（静态链接）中等，需要 2-4min

2. 对如下两个 C 程序，用 gcc 生成对应的.o 模块，再链接在一起得到 a.out。则下列说法正确的是：

```
// main.c
#include <stdio.h>
static int a;
int main() {
  int *func();
  printf("%ld\n", func() - &a);
  return 0;
}
```

```
// util.c
int a = 0;
int *func() {
  return &a;
}
```

A. 在 main.o 中，符号 a 位于 COM 伪节

B. 在 util. o 中，符号 a 位于 COM 伪节

C. 无论怎样链接和运行 a.out， a.out 输出的结果都一样，但必不为 0

D. 以上说法都不正确

D 正确

A 错。它位于 .bss 节。

B 错。它位于 .bss 节。

C 错。注意 ld 时文件顺序的交换会改变 a.out 中两个符号的相对偏移。

于是 D 正确。

（动态链接）困难，需要理解并分析 需要 2-4min

3. 以下程序可以使用 gcc dl.c -ldl 编译并正常运行。如果缺少了 -ldl 标志，链接时会报错 undefined reference to `dlopen'。基于你对于动态链接的理解，请分析出以下说法中**不正确**的一项。

```
// dl.c
#include <dlfcn.h>

const char *path = "/lib/libc.so";
int (*printf)(const char *x);

int main() {
  // 加载共享库
  void *handle = dlopen(path, RTLD_NOW);
  // 解析符号 "printf" 并返回地址
  printf = dlsym(handle, "printf");
  // 调用
  printf("2022 is coming!\n");
  // 关闭共享库
  dlclose(handle);
}
```

A. 该机器上 libdl.so 模块中包含符号名为 dlopen 的动态链接符号表条目
B. 在 a.out 文件中包含 printf 的 PLT 条目和相应的 GOT 条目
C. 在 a.out 文件中包含 dlopen 的 PLT 条目和相应的 GOT 条目
D. 如果使用 gcc -ldl dl.c 编译程序，则会在链接时发生同样错误

B 错误。

解析：

省略 -ldl 标志报错，表明 ld 默认不会包含 libdl.so（与之对比，libc.so 默认包含），并且 dlopen 的定义来自于该共享库。

A 正确。.dynsym 含有一个符号表条目。格式形如

0000000000001390 g DF .text 0000000000000085 GLIBC_2.2.5 dlopen

“动态链接符号表”的描述是准确的，也不影响理解。

B 错误，printf 只是一个未初始化的全局变量。它不是内置的 printf 函数。

C 正确。默认程序动态绑定 dlopen 到共享库，它需要自己的 PLT 表和 GOT 表。

D 正确。gcc 按照命令行顺序解析。不管是动态库还是静态库只解析当前已经被引用的符号（这一点容易推断，否则没有必要建立专门的库文件格式了。因此没有补充在题目中交代动态链接符号解析的规则。），所以 -ldl 放在第一个位置没有任何效果。最后会在链接阶段产生 dlopen、dlsym 或者 dlclose 未能解析的错误。

额外说明，libc.so 和 libdl.so 在实际系统上可能会带上版本号，路径名一般也更复杂。这里为了出题，做了合适的简化。

4. **（本大题共三问，共 10 分）**有以下三个 c 文件 hd.h f1.c f2.c。使用

```
gcc -c f1.c f2.c; gcc f1.o f2.o
```

编译后得到可执行文件 a.out。回答以下问题。Part A 中涉及的符号所对应的变量已在代码中加粗。**本大题无需理解代码的含义。**

f1.c

```
#include "hd.h"
#include <stdio.h>

const int total = 1 << 30;
static int count = 0;
static Point pnt;
int iter;
int main() {
  for (iter = 0; iter < total; ++iter) {
    rand_point(&pnt);
    count += if_inside(&pnt);
  }
  printf("Integral on [0,1] is %lf.\n",
         1.0 * count / total);
}
```

hd.h

```
typedef struct {
  double x;
  double y;
} Point;

void rand_point(Point *);
int if_inside(Point *);
```

f2.c

```
#include "hd.h"
#include <stdlib.h>
#include <time.h>

void rand_point(Point *ptr) {
  static int seed = 0;
  if (!seed) {
    srand((unsigned)time(NULL));
    seed = 1;
  }
  ptr->x = 1.0 * rand() / RAND_MAX;
  ptr->y = 1.0 * rand() / RAND_MAX;
}

int if_inside(Point *p) {
  return 1 / (1 + p->x) >= p->y;
}
```

Part A. **（每个符号 1 分，共 5 分）**请说明以下符号是否在 a.out 的符号表中。如果是，请

进一步指出符号定义所在的节，可能的选择有.text、.data、.bss、.rodata、COM、UNDEF、ABS。

符号名	iter	pnt	Point	total	seed
在符号表中？**（填是/否）**					
定义所在节					

Part B. **（每空 1 分，共 3 分）**使用 objdump -dx f1.o f2.o 看到如下几条代码。**这里你可以将重定位类型 R_X86_64_PLT32 和 R_X86_64_PC32 同等看待。**

```
# objdump 重定位条目格式：
#            OFFSET: TYPE              VALUE
# e.g.           18: R_X86_64_PLT32    rand_point-0x4
# 所有数值均以十六进制表示


# f1.o
0000000000000000 <main>:
... # 省略无关代码
  17: e8 00 00 00 00.         callq  1c <main+0x1c>
          18: R_X86_64_PLT32      rand_point-0x4
  1c: 48 8d 3d 00 00 00 00    lea 0x0(%rip),%rdi
          1f: R_X86_64_PC32       .bss+0xc
  23: e8 00 00 00 00          callq  28 <main+0x28>
          24: R_X86_64_PLT32      if_inside-0x4
  28: 89 c2                   mov %eax,%edx
... # 省略无关代码


# f2.o
0000000000000000 <rand_point>:
... # 省略无关代码

0000000000000074 <if_inside>:
... # 省略无关代码
```

据此你可以确定 <main+0x1f> 处的重定位条目是针对符号_____（**填写符号名，不要填写 .bss 这个节名**）的重定位，同时该符号定义的位置在 f1.o 中相对于 .bss 节的偏移量是 0x_____。

现已知 a.out 文件中 <main+0x17> 行变成

```
11a1: e8 69 00 00 00          callq  <rand_point>
```

那么 a.out 中 <main+0x23> 行将变成

```
11ad: e8 ____________          callq  <if_inside>
```

Part C.（每空 1 分，共 2 分）使用 execve 加载 a.out 并执行时，其中第一个被执行的语句默认是_____(单选) 函数的开头。已知 gcc -e 可以修改该默认行为到一个程序指定的函数，据此你推断该函数执行在_____态下(填 用户/内核)。
A. _init B.main C.__libc_start_main D._start

解析：[Part A 和 Part C 是基础题，Part B 难度适中。**答案均唯一**]

Part A

符号名	iter	pnt	Point	total	seed
是否在符号表中	是	是	否	是	否
定义所在节	.bss	.bss		.rodata	

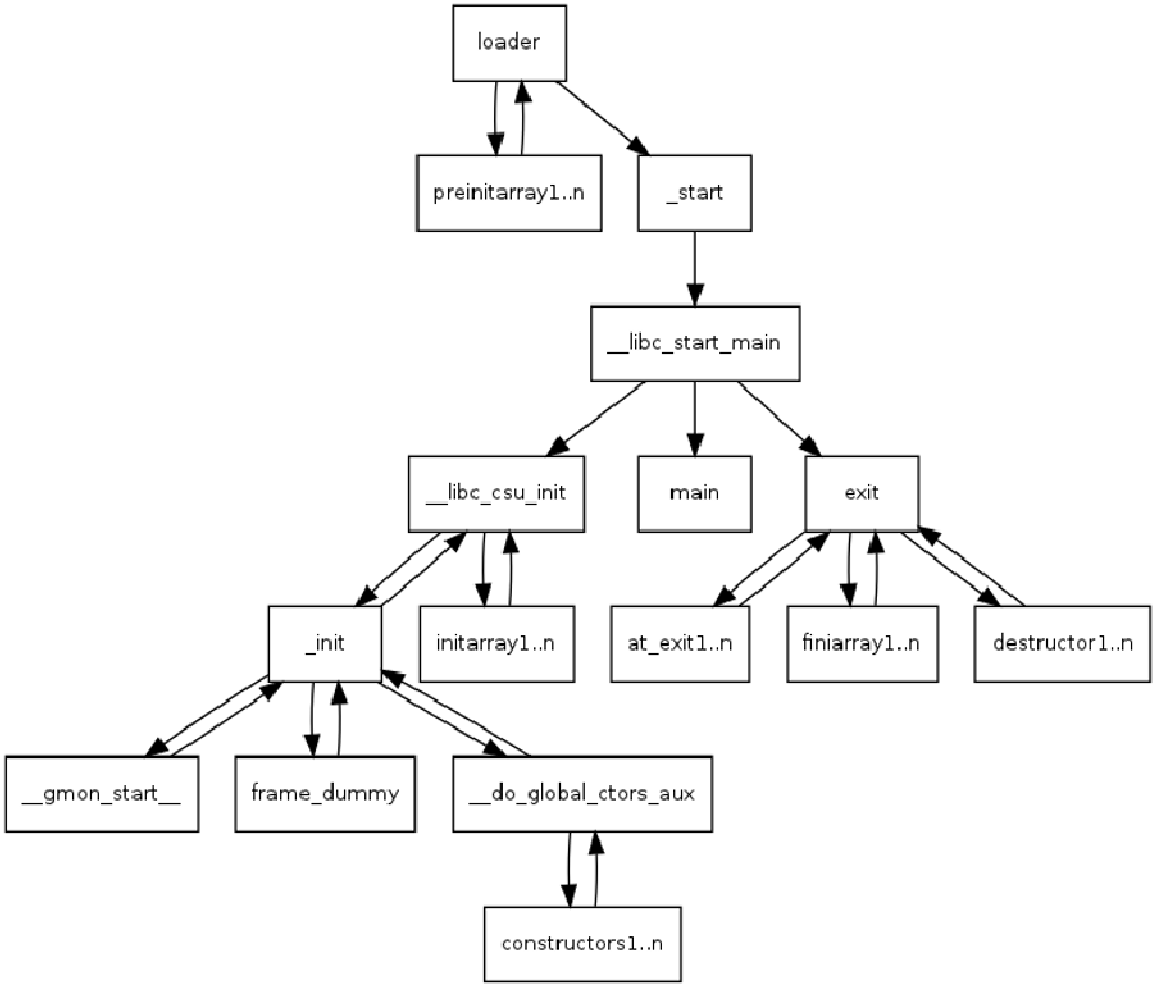
Point 和 total 的部分容易出错。Point 作为类型定义，在 C 中其结构信息已经作为偏移量被汇编代码包含，不需要再显式地输出到.o 文件中。total 已经被初始化成一个非零值，由于 const 修饰 [read-only]，将放入.rodata 中。[改卷时 .data 和 .rodata 均给分]
评分标准：在符号表中的符号，必须正确写出其定义所在节才能得分。

Part B
pnt; 10; d1 00 00 00
由汇编可知 <main+0x1c> 处是准备 if_inside 函数的参数。于是符号是 pnt。假设其相对.bss 偏移为 x，refaddr 表示条目的地址，则根据重定位类型都是相对偏移，有
.bss + 0xc - refaddr = .bss + x - %rip
于是
x = 0xc + %rip - refaddr = 0xc + 0x4 = 0x10
因为 if_inside - rand_point = 0x74 不变，故第四问的结果是
0x69 - (0x11ad - 0x11a1) + 0x74 = 0xd1
评分标准：第二空允许有若干前导 0，其余每空必须完全一致才得分

Part C D; 用户 _start 是 OS 执行这段程序的第一条语句，即入口点，它来自于 crt0.o(或者 crt1.o，代表 c run-time)模块[因为这段程序没有自定义入口点函数]。第二问的信息已经强烈暗示了该函数只能运行在用户态下[否则直接破坏操作系统对机器的保护和用户间隔离的作用]。实际上，_start 函数只需准备好 argc、argv 和 envp，然后准备好 __libc_start_main 的参数。
具体的启动过程如下图所示[图源：

http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html]

评分标准：必须完全一致才得分



本题代码是一个简单的 Monte Carlo 法求积分的算法。

1、 关于进程和异常控制流，以下说法正确的是：

A、 调用 waitpid (-1, NULL, WNOHANG & WUNTRACED) 会立即返回：如果调用进程的所有子进程都没有被停止或终止，则返回 0；如果有停止或终止的子进程，则返回其中一个的 ID。

B、 进程可以通过使用 signal 函数修改和信号相关联的默认行为，唯一的例外是 SIGKILL，它的默认行为是不能修改的。

C、 从内核态转换到用户态有多种方法，例如设置程序状态字；从用户态转换到内核态的唯一途径是通过中断/异常/陷入机制。

D、 中断一定是异步发生的，陷阱可能是同步发生的，也可能是异步发生的。

答案：C
简单
A 中 option 参数应该使用 | 运算结合。B 中 SIGKILL 不是唯一的例外，例外共有两个：SIGKILL、SIGSTOP。D 陷阱一定是同步发生的

2、异常可以分为四类：中断、陷阱、故障、终止。以下都属于中断的是：
A、I/O 请求完成、系统定时器的信号
B、除零、缺页
C、系统调用、非法指令
D、键盘 Ctrl+C、机器检查

答案：A
中等
A、 中断、中断
B、 故障、故障
C、 陷阱、终止
D、 中断、终止
P506

一. 异常控制流

PART A. Alice 想用两个进程来顺序输出奇数和偶数，请你帮她补全代码。她的做法是：父进程首先 fork 出两个子进程，之后子进程之间互相通信，顺序输出 1,2,…,N。请严格按照注释描述的功能填写代码，**假设兄弟进程的 pid 是相邻的**。

```
int N;
int nxt; //表示该子进程下一个需要输出的数
int pid_1 = 0; //第一次 fork 出的子进程 pid
int pid_2 = 0; //第二次 fork 出的子进程 pid
// All child process should do their work in handler 1
void handler1(int sig) {
    printf("%d\n", nxt);
    nxt += 2;
    if (nxt & 1) kill(getpid()+1, SIGUSR1);
    else {
        assert(pid_1 != 0);
        ________A________ // 通知兄弟进程
    }
    if (nxt > N) exit(0);   // 子进程完成输出后退出
}

int main(int argc, char* argv[]) {
    ________B________ //将 handler1 绑定到 SIGUSR1 上
    N = atoi(argv[1]);
    nxt = 1;
    if ((pid_1 = fork()) != 0) {
        ________C_______;   // 设置 nxt 的初值
        if ((pid_2 = fork()) != 0) {   // 该进程是父进程
            kill(pid_1, SIGUSR1);
            goto wait_til_end;   //跳转并等待子进程结束
        }
    }
    while (1) { sleep(1); }   // 子进程会在此循环直到输出完成

    wait_til_end:
    int status;
    while (_______D_______)   // 用 waitpid 等待子进程结束，注意 status
        assert(WIFEXITED(status));
    return 0;
```

（4 分，每空 1 分）

（难度：简单）A. _____

（难度：简单）B. _____

（难度：简单）C. _____

（难度：简单）D. _____

PART B. 阅读如下 C 代码，回答问题

```
int counter = 0;
int pid = 0;
int N = 2;
void handler1(int sig) {
    counter++;
    printf("%d", counter); fflush(stdout);
    // Kill(pid, SIGUSR2);
}
void handler2(int sig) {
    printf("R"); fflush(stdout);
}
int main() {
    Signal(SIGUSR1, handler1);
    Signal(SIGUSR2, handler2);
    if ((pid = Fork()) == 0) { // child
        for (int i = 0; i < N; ++i) {
            printf("C"); fflush(stdout);
            Kill(Getppid(), SIGUSR1);
        }
    } else { // parent
        Wait(NULL);
    }
    return 0;
}
```

1.（难度：简单）进程在何时检查待处理信号，并调用相应的 signal handler 处理信号？_____（1 分）

A.随时　　B.用户态切换到内核态　　C.内核态切换到用户态

2.（难度：简单）在两次检查并处理信号量的时间间隙中，如果进程接收到 n 个**相同**信号，那么进程实际会处理几个信号？_____（1 分）

A.1　　B.1 到 n 之间的随机数值　　C.n

3.（难度：中等）如果 N=2，所有可能的输出为：_____

（全部选对得 2 分，部分选对得 1 分，选错不得分）

A.CC　　B.CC1　　C. CC12　　D. C1C2

4.（难度：难）如果 N=2，并且取消 hanlder1 中的注释（第 7 行），所有**不可能**的输出为：_____

（全部选对得 2 分，部分选对得 1 分，选错不得分）

A. CC B. CC1 C. CC1R2 D. C1RC2 E. C1CR2

答：

PART A.

A: kill(pid_1, SIGUSR1);或 kill(getpid()−1, SIGUSR1);

B: signal(SIGUSR1, handler1);

C: nxt = 2;

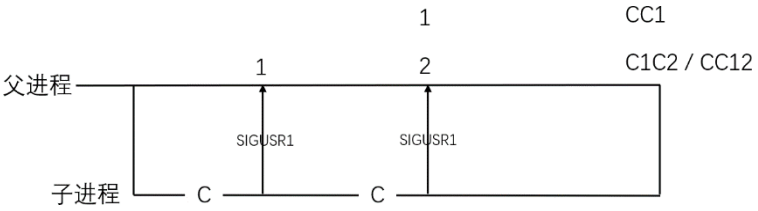
D: waitpid(0, &status, 0)> 0 或 waitpid(−1, &status, 0)>0

PART B.

1.C

2.A

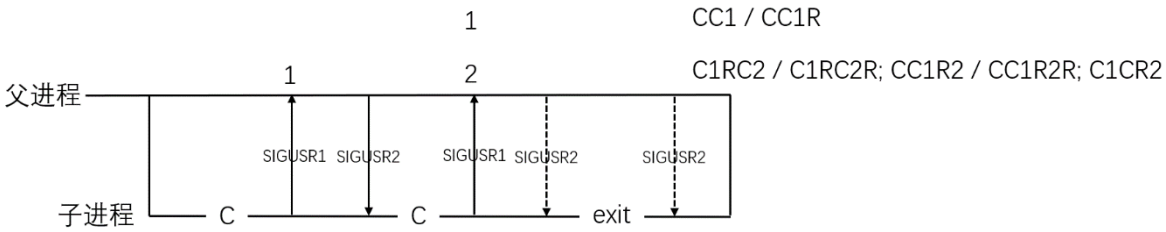
3.BCD: CC1, C1C2, CC12



4.A: CC

所有可能的输出为 CC1, CC1R, CC1R2, CC1R2R, C1RC2, C1RC2R, C1CR2。

比如，CC1 是可能的，因为子进程可以在接收到父进程的 SIGUSR2 之前退出。



解题要点：

1. 进程只在内核态切换到用户态时检测并处理信号
2. 在处理信号之前相同信号接收到多次，只按照一次计算
3. 注意子进程可以比父进程先结束，父进程将无法发送信号给子进程

选择

1. 下列关于虚存和缓存的说法中，**正确**的是：

A. TLB 是基于物理地址索引的高速缓存

B. 多数系统中，SRAM 高速缓存基于虚拟地址索引

C. 在进行**线程**切换后，TLB 条目绝大部分会失效

D. 多数系统中，在进行进程切换后，SRAM 高速缓存中的内容不会失效

答案：D

解析：属于中等题。考察虚拟内存和高速缓存的关系，并且和进程/线程有一定联系。

A 课本原话：A TLB is a small, virtually addressed cache where each line holds a block consisting of a single PTE.

B 课本原话：Although a detailed discussion of the trade-offs is beyond our scope here, most systems opt for physical addressing.

C 线程共享虚拟地址空间，所以 TLB 中条目不会失效

D 课本原话：With physical addressing, it is straightforward for multiple processes to have blocks in the cache at the same time and to share blocks from the same virtual pages.

2. 阅读下列代码并回答选项。（已知文件“input.txt”中的内容为“12”，头文件没有列出）

```
void *Mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);

int main(){
    int status;
    int fd = Open("./input.txt", O_RDWR);
    char* bufp = Mmap(NULL, 2, PROT_READ | PROT_WRITE ,
                      MAP_PRIVATE, fd, 0);
    if (Fork()>0){
       while(waitpid(-1,&status,0)>0);
       *(bufp+1) = '1';
       Write(1, bufp, 2); // 1: stdout
       bufp = Mmap(NULL, 2, PROT_READ, MAP_PRIVATE, fd, 0);
       Write(1, bufp, 2);
    }
    else{
        *bufp = '2';
        Write(1, bufp, 2);
    }
```

在 shell 中运行该程序，正常运行时的终端输出应为

A. 221112 B. 222121 C. 222112 D. 221111

答案：A

解析：属于中等题。主要考察对私有对象 COW 机制及 mmap 函数的理解。

程序打开文件描述符 fd 后，使用 mmap 函数创建了一块映射到文本“./input.txt”所在区域的私有虚拟内存区域，其地址存储在指针 bufp 中。父进程创建完子进程后，首先等待子进程结束。子进程在试图对*bufp 写入‘2’时，触发了一次 COW。它在物理内存中创建一个“input.txt”文本所在页面的副本，并在新的页面里完成写入操作，写入完成后子进程 bufp 所在地址的第一个字节值为新写入的‘2’，第二个字节值为原始值‘2’。子进程结束后，其缓冲区内”21”的值正常输出。父进程在试图对*(bufp+1)写入‘1’，同样触发了一次 COW，bufp 所在地址的第一个字节为原始值‘1’，第二个字节为新写入的‘1’，故父进程输出“11”。最后父进程重新创建一块映射到“input.txt”所在页面的虚拟内存并输出。由于上述写入都是在新的物理页面下完成的，“input.txt”所在页面没有发生修改，故最后一步父进程的输出为“12”。综上，最终在终端的输出为“221112”。

大题（15 分）

IA32 体系采用**小端法**、32 位虚拟地址和两级页表。两级页表大小相同，页大小都是 4 KB = 2^{12} Byte，结构也相同。TLB 采用**直接映射**，4 位组索引。TLB 和页表每一项格式如图所示：

31 12	11 9	8	7	6	5	4	3	2	1	0
Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	P

部分位的含义如下：

0 (P): 1 表示存在，0 表示不存在

1 (R/W)：1 表示可写，0 表示只读

2 (U/S)：1 表示内核模式，0 表示用户模式

当系统运行到某一时刻时，TLB **有效位为 1** 的条目如下（未列出部分都是无效的）：

索引（十进制）	TLB 标记	内容
0	0x0400	0x0ec91313
3	0x02ff	0x5d2bac01
5	0xd551	0x019fa42d
11	0x55a6	0xfdd3c66b
13	0x5515	0xb591926b

一级页表的基地址为 0x00e66000，物理内存中的部分内容如下（均为十六进制）：

地址	内容	地址	内容	地址	内容
00615000	21	00615001	2d	00615002	ee
00615003	c0	006154d0	ff	006154d1	a0
00e66001	a1	00e66002	a4	00e66003	67
00e66004	21	00e66005	57	00e66006	61
00e66007	00	2167e000	42	2167e001	67
2167e002	9a	2167e003	7c	c0ee2000	6f
c0ee2001	d5	c0ee2002	7e	c0ee24d0	48
c0ee24d1	83	c0ee24d2	ec	c0ee2d21	11
c0ee2d22	6b	c0ee2d23	82	c0ee2d24	8a

1. 将 cache 清空。访问一个在主存中的虚拟地址，TLB 命中，**没有**触发缺页异常，这一过程中，需要访问物理内存（主存）_____次（3 分）。具体来说，如果该虚拟地址为 y = 0xd5515213，y 地址所具有的**实质权限**是_____（多选题，选对才得分 2 分）。

①可写 ②只读 ③用户模式权限 ④内核模式权限

2. 不考虑第一小问，将 cache 清空。访问一个在主存中的虚拟地址，TLB 不命中，**没有**触发缺页异常，这一过程中，需要访问物理内存（主存）_____次（3 分）。具体来说，如果该虚拟地址为 x = 0x004004d0，那么 x 对应的二级页表起始地址是_____（填写 16 进制，例如 0x00123000，2 分），x 地址上单字节的**内容**是_____（填写 16 进制，例如 0x00，1 分）。

3. 考虑下面计算矩阵和向量乘法代码：

```
1 int *mat_vec_mul(int **A, int *x, int n)
2 {
3     int i, j;
4     int *y = (int *)malloc(n * sizeof(int));
5     for (i = 0; i < n; i++)
6         for (j = 0; j < n; j++)
7             y[i] += A[i][j] * x[j];
8     return y;
9 }
```

⑴ 在 64 位 Linux 机器中运行该代码，输入**同一组合法的参数**后，每次运行返回的向量的值都**不一样**，修复这一错误有一种简单的方法，将第____行改为_____。（第二空填写 C 代码，每空 1 分，共 2 分）

可能用到的函数：

```
void *memcpy(void *dest, const void *src, size_t n);
void *memset(void *s, int c, size_t n);
void *calloc(size_t nelem, size_t elemsize);
void *realloc(void *ptr, size_t size);
```

⑵ 在进行前一问的测试**之前**，还在 64 位 Linux 机器上进行过如下用户代码测试（输入 mat_vec_mul 的参数都**非零**）：

```
int *y = mat_vec_mul(A, x, n);
int z = y[0];
```

结果发生了段错误。通过 gdb 调试发现 mat_vec_mul 函数内并没有发生段错误，但是在初始化变量 z 时发生了段错误，后来发现是参数的输入有问题。写出出现这种错误的**充分必要条件**_____（1 分）

4. Double fault：Intel 处理器中有一种特殊的异常，被称为 double fault。此异常发生表明调用某个**故障（fault）**A 的处理程序后又触发了另一个故障 B。正常情况下，故障 B 会有相应异常处理程序来处理，因此两个故障 B 和 A 可以被顺序解决。但是如果处理器无法正常处理故障 B，或是处理了之后依然无法处理故障 A，就会产生 double fault，并终止（abort）。假设除了缺页异常处理程序外，其他异常处理程序**都不会产生新的故障**。如果在某次**缺页故障**时产生了 double fault，其原因可能是_____（不定项选择，都选对才得分，1 分）

① 运行缺页故障处理程序时， CPU 上的权限位是内核态，但所执行代码段 U/S=0

② 运行缺页故障处理程序时，CPU 接收到了键盘发送的 Ctrl + C 信号

③ 缺页故障处理程序没有加载到主存中

答案：

1. 1 ②④
2. 3 0x00615000 0x48
3. ⑴ 4 `int *y = (int *)calloc(n, sizeof(int));` 或
   `int *y = (int *)Calloc(n, sizeof(int));`
   其中 `calloc/Calloc` 的两个参数只要乘起来等于 `n * sizeof(int)` 即算对
   ⑵ `n < 0`（或其他等价表达）
4. ③

解析：

1. 考察 TLB 的位划分与表项含义，属于简单题。
题目告知 TLB 命中且无缺页，因此只需要访问物理页，即访问一次物理内存
根据 y 的值可以得到 TLBT=0xd551，TLBI=5，查表知 y 在 TLB 条目的内容为 0x019fa42d，R/W=0，U/S=1，因此这一页的实质权限为只读、内核模式。
参见课本第 9.6 和 9.7 节（P567-582）

2. 考察虚拟内存地址翻译，属于简单+中等题。
题目告知 TLB 未命中且无缺页，因此需要访问页目录、二级页表和物理页，共访问三次物理内存。
计算得 x 的 VPN1=1， VPN2=0，PPO =0x4d0。页目录基地址在 0x00e66000，因此相应页目录中所在条目地址为 0x00e66000+1*4=0x00e66004，从表中按小端法读出 0x00615721，因此二级页表存在，首地址为 0x00615000，这恰好也是相应二级页表中所在条目的地址，从表中按小端法读出 0xc0ee2d21，因此物理页存在，首地址为 0xc0ee2000，因此 x 的物理地址为 0xc0ee2000+0x4d0=0xc0ee24d0。从表中读出一个字节，为 0x48。
参见课本第 9.6 和 9.7 节（P567-582）

3. 本题考查内存安全。
⑴ 考察 `malloc` 函数的性质，属于中等题。
`malloc` 函数并不会清零所分配内存区域，因此同样的输入可能会有不同的输出，但 `mat_vec_mul` 函数别的行都不可能产生这一效果，因此只需要将第 4 行换为

```
int *y = (int *)Calloc(n, sizeof(int));
```

`Calloc` 是一个带错误检查的、会将所分配区域置零的函数。
参见课本第 9.11.2 节正文例子（P610）。

⑵ 内存安全的综合考察，属于难题。
假设 `malloc` 函数分配成功，那么 `y` 不是空指针。`y[0]` 不可能在外部访问出现段错误，因此这种情况不成立，`malloc` 一定出现了问题，导致 `y` 是空指针。但是此时 `mat_vec_mul` 并没有触发段错误，说明函数内部一次对 `y` 的访问都没有，这只有可能循环都没有进入，因此必须有 `n<=0`，根据题设，参数是非零的，所以 `n<0`。以上推导出了 `n<0` 是必要条件。
下面推导 `n<0` 是充分条件，即任何 `n<0` 都会导致这样的错误。为此，只需要说明 `malloc` 函数一定分配失败。首先，这是 `Linux` 64 位系统，因此地址空间为 64 位，`size_t` 为 64 位，用户虚拟地址空间为 48 位。注意 `n*sizeof(int)` 会将 `n` 的类型转为 `long`，再转为 `unsigned long`，而 `n` 只有 32 位，所以符号扩展之后最高位至少有 33 个 1，乘 `sizeof(int)`，即 4 之后，最高位还有至少 29 个 1，因此 $n*sizeof(int)>2^{63}>2^{48}$，超过了用户虚拟地址空间的大小，因此无论如何，

`malloc` 函数都会分配失败。
参见课本第 9.8 节（P576）、9.9.1 节（P588）、2.2.8 节（P58-59）。

4．本题考查对缺页故障、异常处理的综合理解，属于难题。
① 如果 CPU 权限位是内核态，它自然可以运行用户级代码，不会触发异常。
② 如果缺页故障处理程序运行时，收到外部中断，这不属于故障，因此不会触发。
③ 如果缺页故障处理程序不在主存，那么调用它会触发缺页故障，于是这个故障又会导致调用缺页故障处理程序，但它仍然不在主存，故这一故障无法被解决，于是触发 double fault。
相关内容参见课本第 8.1 节（P502-507）、9.7.2 节（P581-582）和 Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3: System Programming Guide 的第 6.15 节。

备选题

1. 虚拟内存为内存的使用和管理提供了简化，这样的简化**没有**体现在

A. 编译器将 C 文件编译为目标文件的过程

B. 链接器生成完全链接的可执行文件的过程

C. 加载器向内存中加载可执行文件和共享对象文件的过程

D. 不同进程共享相同物理页面的过程

答案：A

解析：属于简单题。考察虚拟内存在管理内存方面起到的作用，主要对应中文版课本第 566 页。

A 程序的编译过程与使用虚拟内存还是使用物理内存的联系较小。

B 课本原话：…Such uniformity greatly simplifies the design and implementation of linkers, allowing them to produce fully linked executables that are independent of the ultimate location of the code and data in physical memory.

C. 课本原话：Virtual memory also makes it easy to load executable and shared object files into memory.

D. 课本原话：… the operating system can arrange for multiple processes to share a single copy of this code by mapping the appropriate virtual pages to in different processes to the same physical pages.

2. 下列选项中**错误**的是

A. 在使用虚拟地址空间的的系统中，程序引用的页面总数**必须不超过**物理内存总大小。

B. **主存中**的每个**有效**字节都有至少一个选自虚拟地址空间的**虚拟地址**和一个选自物理地址空间的**物理地址**。

C. 当在程序中正常调用 malloc 函数时，操作系统会分配出相应大小（例如 k 个）的**连续虚拟页面**，并且将它们映射到物理内存中**任意位置**的 k 个**物理页面**。

D. **不同**进程的多个**虚拟**页面可以映射到**同一个**共享**物理**页面上。

答案：A

解析：属于简单题。主要考察地址空间和虚拟内存相关的基本概念。

A. 如果运行的程序有良好的时间局部性，工作集大小（引用的页面总数）即使超过物理内存总的大小也能有较好的性能。

B. 课本原话：[p561] Each byte of main memory has a virtual address chosen from the virtual address space, and a physical address chosen from the physical address space.

C. 课本原话：[p567] When a program […] requests additional heap space […], the operating system allocates an appropriate number […] contiguous virtual memory pages, and maps them […] arbitrary physical pages located anywhere in physical memory.

D. 课本原话：[p566] Notice that multiple virtual pages can be mapped tothe same shared physical page.

3. 以下关于动态内存分配的说法中，**错误**的是

A. 可以通过调用 sbrk(0) 获取当前进程中堆的顶部地址

B. 如果向一个已经 free 的指针写入数据，**一定会**触发异常

C. 如果使用 malloc(0x10) 获取一个指针，然后写入 0x200 字节大小的数据，**不一定**会触发异常

D. 使用 mmap 也是动态分配内存的方法之

答案：B

解析：属于中等题。考察对动态内存分配的理解，并与内存保护机制，异常有一定联系。

A 课本原话：If incr is zero, then sbrk returns the current value of brk.

B 已经分配的内存一般不会被 OS 回收，相当于用户空间正常的内存访问

C 只要不超出堆的范围，就不会触发异常。如果超出堆的范围，则触发异常

D 课本原话：While it is certainly possible to use the low-level mmap and munmap functions to create and delete areas of virtual memory, C programmers typically find it more convenient and more portable to use a dynamic memory allocator when they need to acquire additional virtual memory at run time.

4. 以下关于虚拟内存的说法**错误**的是

A. 虚拟内存一般不需要来自应用程序开发者的干涉

B. 虚拟地址空间可以比物理内存更小

C. 连续的虚拟内存**总是**映射到连续的物理内存

D. 目标文件中的.bss 段映射到全是二进制零的匿名文件

答案：C

解析：属于中等题。主要考察虚拟内存相关的理解问题。

A. 课本原话：… [virtual memory] works silently and automatically, without any intervention from the application programmer.

B. 课本原话：Interestingly, some early systems […] supports a virtual address space that was smaller than the available physical memory.

C. 课本原话：… the operating system allocates […] contiguous virtual memory pages […] The pages can be scattered randomly in physical memory.

D. 课本原话：The bss area is demand-zero, mapped to an anonymous file […]

第十章 I/O 部分选择（4 分）

1. 下列关于系统 I/O 的说法中，正确的是（）：

A. Linux shell 创建的每个进程开始时都有三个打开的文件：标准输入（描述符为 0），标准输出（描述符为 1），标准错误（描述符为 2），这使得程序始终不能使用保留的描述符 0,1,2 读写其他文件。

B. Unix I/O 的 read/write 函数是异步信号安全的，故可以在信号处理函数中使用。

C. RIO 函数包的健壮性保证了对于同一个文件描述符，任意顺序调用 RIO 包中的任意函数不会造成问题。

D. 使用 int fd1 = open(“ICS.txt”, O_RDWR); 打开 ICS.txt 文件后，再用 int fd2 = open(“ICS.txt”, O_RDWR); 再次打开文件，会使得 fd1 对应的打开文件表中的引用计数 refcnt 加一。

答案：B（简单）

A. 描述符 0，1，2 可以重定向到其他文件。

B. 正确，教材 P534。

C. 有缓冲区和无缓冲区的不可以交叉使用，教材 P629。

D. 多次打开文件应该对应着多个打开文件表，教材 P635。

2. 考虑以下代码，假设 ICS.txt 中的初始内容为"ICS!!!ics!!!"：

```
int fd = open("ICS.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
for (int i = 0; i < 2; ++i){
    int fd1 = open("ICS.txt", O_RDWR | O_APPEND);
    int fd2 = open("ICS.txt", O_RDWR);
    write(fd2, "!!!!!!", 6);
    write(fd1, "ICS", 3);
    write(fd, "ics", 3);
}
```

假设所有系统调用均成功，则这段代码执行结束后，ICS.txt 的内容为（）：

A. ICSics

B. !!!icsICS

C. !!!icsics!!!ICSICS

D. !!!icsICSICS

答案：D（中等）

选择题

11 章

1. 以下符合对 TCP 协议描述的是
①可靠传输 ②在主机间传输 ③可以基于 IP 协议
④全双工 ⑤可以基于 DNS

A. ①②④ B. ③④⑤ C. ①③④ D. ②④

答案：C
难度：简单
预计时间：小于 1min
①TCP 提供可靠传输
②TCP 可以在两个进程间传输，包括同一个主机上的
③TCP/IP 本身是一个协议族，TCP 可以基于 IP 来实现，也存在不使用 IP 的实现
④TCP 是全双工的，双方同时都可以进行读写
⑤TCP 并不依赖 DNS 服务

2. 对于建立客户端与服务器的 TCP 连接，以下关于套接字接口，正确的是：
A. 对于 TCP 连接而言，其可以由连接双方的套接字对四元组（双方的 IP 地址和端口号）唯一确定。
B. connect 函数用于客户端建立与服务器的连接，其参数中的套接字地址结构需要指明本客户端使用的端口号
C. socket 函数返回的套接字描述符，可以立即使用标准 Unix I/O 函数进行读写
D. bind 函数用于将用于客户端的主动套接字转化为用于服务器的监听套接字

答案：A
难度：简单
预计时间：小于 1min
A. TCP 连接由连接双方进程的套接字唯一确定
B. connect 应该指明想要连接的服务器的套接字地址结构
C. socket 只是获得套接字，还不可以进行读写，需要完成连接以后才可以进行读写
D. bind 函数只是用于绑定套接字地址的，而 listen 函数用于将用于客户端的主动套接字转化为用于服务器的监听套接字

3. 下列有关 Web 服务错误的是
A. Web 服务使用客户端-服务器模型，使用了 HTTP 协议，可以传输文本，HTML 页面，二进制文件等多种内容
B. HTTP 响应会返回状态码，它指示了对响应的处理状态
C. Web 服务使用 URL 来标明资源，这提供了一层抽象，使得客户端仿佛在访问远端的文件目录，而服务器处理了 URL 资源和具体文件/动态内容的映射关系
D. 多个域名可以映射到同一个 IP 地址，但一个域名不可以映射到多个 IP 地址

答案：D

难度：简单
预计时间：1min
A. 考察 Web 服务的基本概念
B. 考察 HTTP 状态码
C. 考察 Web 的 URL 机制内在的思想
D. 考察因特网域名的多对多关系

备选：

4. 下列关于全球 IP 因特网的说法中，正确的是
A. TCP/IP 协议规定网络字节顺序取决于不同主机的字节顺序，从而方便主机和网络之间收发数据
B. 域名集合和 IP 地址集合的映射关系目前由 http 来维护
C. 多个域名可以映射到同一个 IP 地址，但一个域名不可以映射到多个 IP 地址
D. 从内核的角度来看，套接字是通信的一个端点；从用户程序的角度来看，套接字是一个有相应描述符的打开文件

答案：D
解析：
A：网络字节顺序是大端法
B：DNS 负责维护域名集合和 IP 地址集合的映射关系
C：一个域名也可以被映射到多个 IP 地址

5. 下列关于网络的说法中，错误的是
A. 在 TCP 连接中，客户端和服务器都是指主机
B. 网络按照覆盖范围的大小可以分成局域网和广域网，其中一种流行的局域网技术是以太网
C. 网络可以被主机认为是一种 I/O 设备，是数据源和数据接收方
D. 路由器可以把多个不兼容的网络连接起来组成一个互联网络

答案：A
解析：简单题，考察 11.1 部分内容
A：在该模型中，客户端和服务器指进程
BCD 均正确

12 章

1.下列关于 C 语言中进程模型和线程模型的说法中，错误的是：
A.每个线程都有它自己独立的线程上下文，包括线程 ID、程序计数器、条件码、通用目的寄存器值等
B.每个线程都有自己独立的线程栈，任何线程都不能访问其他对等线程的栈空间
C.不同进程之间的虚拟地址空间是独立的，但同一个进程的不同线程共享同一个虚拟地址空间
D.一个线程的上下文比一个进程的上下文小得多，因此线程上下文切换要比进程上下文切换快得多

解析：B
考察 12.3 12.4 线程模型，属于简单题。B 选项，不同的线程栈是不对其他线程设防的。所以，如果

一个线程以某种方式得到一个指向其他线程栈的指针，那么它就可以读写这个栈的任何部分。

大题

并发编程（15 分）

“生产者-消费者”问题是并发编程中的经典问题。本题中，考虑如下场景：

a. 所有生产者和所有消费者**共享同一个 buffer**
b. 生产者、消费者各有 NUM_WORKERS 个（**大于一个**）
c. buffer 的容量为 BUF_SIZE，**初始情况下 buffer 为空**
d. 每个生产者向 buffer 中添加一个 item；若 buffer 满，则生产者等待 buffer 中有空槽时才能添加元素
e. 每个消费者从 buffer 中取走一个 item；若 buffer 空，则消费者等待 buffer 中有 item 时才能取走元素

1. 阅读以下代码并回答问题（**代码阅读提示：主要关注 producer 和 consumer 两个函数**）

```c
/* Producer-Consumer Problem (Solution 1) */

#include "csapp.h"

#define BUF_SIZE 3
#define NUM_WORKERS 50
#define MAX_SLEEP_SEC 10

volatile
static int items = 0; /* How many items are there in the buffer */

static sem_t mutex; /* Mutual Exclusion */
static sem_t empty; /* How many empty slots are there in the buffer */
static sem_t full;  /* How many items are there in the buffer */

static void sync_var_init() {
    Sem_init(&mutex, 0, 1);

    /* Initially, there is no item in the buffer */
    Sem_init(&empty, 0, BUF_SIZE);
    Sem_init(&full, 0, 0);
}

static void *producer(void *num) {
    ①;
    ②;

```

```
27.     /* Critical section begins */
28.     Sleep(rand() % MAX_SLEEP_SEC);
29.     items++;
30.     /* Critical section ends */
31.
32.     V(&mutex);
33.     V(&full);
34.
35.     return NULL;
36. }
37.
38. static void *consumer(void *num) {
39.     ③;
40.     ④;
41.
42.     /* Critical section begins */
43.     Sleep(rand() % MAX_SLEEP_SEC);
44.     items--;
45.     /* Critical section ends */
46.
47.     V(&mutex);
48.     V(&empty);
49.
50.     return NULL;
51. }
52.
53. int main() {
54.     sync_var_init();
55.
56.     pthread_t pid_producer[NUM_WORKERS];
57.     pthread_t pid_consumer[NUM_WORKERS];
58.
59.     for (int i = 0; i < NUM_WORKERS; i++) {
60.         Pthread_create(&pid_producer[i], NULL, producer, (void *)i)
    ;
61.         Pthread_create(&pid_consumer[i], NULL, consumer, (void *)i)
    ;
62.     }
63.
64.     for (int i = 0; i < NUM_WORKERS; i++) {
65.         Pthread_join(pid_producer[i], NULL);
66.         Pthread_join(pid_consumer[i], NULL);
67.     }
68. }
```

a) 补全代码（请从以下选项中选择，可重复选择，每个 1 分，共 4 分）

①_____（24 行）
②_____（25 行）
③_____（39 行）
④_____（40 行）
选项：
A. P(&mutex)
B. P(&empty)
C. P(&full)

b) 如果交换 24 行与 25 行（两个 P 操作），_____（单选，2 分）
   A. 有可能死锁
   B. 有可能饥饿
   C. 既不会死锁，也不会饥饿

c) 交换 32 行与 33 行（两个 V 操作）是否可能造成同步错误？_____（2 分）
   A. 可能
   B. 不可能

d) rand 函数是不是线程安全的？_____（1 分）
   A. 是
   B. 不是

28 行与 43 行对 rand 函数的使用是否会导致竞争？_____（1 分）
   A. 会
   B. 不会

已知 rand 函数的实现如下（来源：https://github.com/begriffs/libc/blob/master/stdlib.h 和 https://github.com/begriffs/libc/blob/master/stdlib.c）：

```
1. #define RAND_MAX 32767
2.
3. unsigned long _Randomseed = 1;
4.
5. int rand() {
6.     _Randomseed = _Randomseed * 1103515425 + 12345;
7.     return (unsigned int)(_Randomseed >> 16) & RAND_MAX;
8. }
9.
10.void srand(unsigned int seed) {
11.    _Randomseed = seed;
12.}
```

解析：a) BACA; b) A; c) B; d) BB
本小题是“生产者-消费者”问题用信号量的经典解法。
a、b 考察的是资源申请和互斥的顺序。
c 考察的是对死锁的分析
d 考察是是线程安全，以及线程不安全的函数在给定场景下是否会出错

评论：这是一道基础题，考察对基础问题的掌握程度，同学们应当能快速得到这些分数。尽管代码较长，但是代码可读性好、结构清晰，同时也是同学们熟悉的代码，阅读应当没有障碍。

2. 考虑“生产者-消费者”问题的另一种解法（**代码阅读提示：12-69 行之外均与上一种解法相同**）

```
1. /* Producer-Consumer Problem (Solution 2) */
2.
3. #include "csapp.h"
4.
5. #define BUF_SIZE 3
6. #define NUM_WORKERS 50
7. #define MAX_SLEEP_SEC 10
8.
9. volatile
   static int items = 0; /* How many items are there in the buffer */
10.
11.static sem_t mutex;                /* Mutual Exclusion */
12.static sem_t sem_waiting_producer; /* Wait for empty slots */
13.static sem_t sem_waiting_consumer; /* Wait for available items */
14.
15.volatile static int num_waiting_producer = 0;
16.volatile static int num_waiting_consumer = 0;
17.
18.static void sync_var_init() {
19.     Sem_init(&mutex, 0, 1);
20.
21.     Sem_init(&sem_waiting_producer, 0, ①);
22.     Sem_init(&sem_waiting_consumer, 0, ①);
23.}
24.
25.static void *producer(void *num) {
26.     P(&mutex);
27.     while (items == BUF_SIZE) {
28.         num_waiting_producer++;
29.         ②;
30.         ③;
31.         P(&mutex);
32.     }
33.
34.     /* Critical section begins */
35.     Sleep(rand() % MAX_SLEEP_SEC);
36.     items++;
37.     /* Critical section ends */
38.
39.     if (num_waiting_consumer > 0) {
40.         num_waiting_consumer--;
41.         V(&sem_waiting_consumer);
```

```
42.     }
43.     V(&mutex);
44.
45.     return NULL;
46. }
47.
48. static void *consumer(void *num) {
49.     P(&mutex);
50.     while (items == 0) {
51.         num_waiting_consumer++;
52.         ④;
53.         ⑤;
54.         P(&mutex);
55.     }
56.
57.     /* Critical section begins */
58.     Sleep(rand() % MAX_SLEEP_SEC);
59.     items--;
60.     /* Critical section ends */
61.
62.     if (num_waiting_producer > 0) {
63.         num_waiting_producer--;
64.         V(&sem_waiting_producer);
65.     }
66.     V(&mutex);
67.
68.     return NULL;
69. }
70.
71. int main() {
72.     sync_var_init();
73.
74.     pthread_t pid_producer[NUM_WORKERS];
75.     pthread_t pid_consumer[NUM_WORKERS];
76.
77.     for (int i = 0; i < NUM_WORKERS; i++) {
78.         Pthread_create(&pid_producer[i], NULL, producer, (void *)i)
    ;
79.         Pthread_create(&pid_consumer[i], NULL, consumer, (void *)i)
    ;
80.     }
81.
82.     for (int i = 0; i < NUM_WORKERS; i++) {
83.         Pthread_join(pid_producer[i], NULL);
84.         Pthread_join(pid_consumer[i], NULL);
```

```
85.     }
86.}
```

a) 补全补全代码（请从以下选项中选择，④⑤无需填写，每个 1 分，共 3 分）

①_____（21、22 行）

②_____（29 行）

③_____（30 行）

选项：

A. 0
B. 1
C. P(&sem_waiting_producer)
D. V(&mutex)

b) 如果 27 行和 50 行的 while 换成 if，是否可能造成同步错误？_____（2 分）

A. 可能
B. 不可能

解析：a) ADC; b) A

a 考察对代码理解，①涉及的两个信号量用于线程的休眠，因此在任意时刻，这两个信号量的值都是 0；②③应当先释放 mutex，然后再等待，注意信号量的 V 操作并不会丢失（信号量的值必然会加一），因此即便这两个 P、V 操作之间可能被打断，这个程序仍然是正确的。

b 考察对控制流的分析，在如下场景下这一修改会造成同步错误：一个线程从 27 行的 P 操作中被唤醒以后，被中断，另一线程比前述线程优先获取 mutex（26 行），且在 27 行测试得到 false，因此这个 slot 被这个新来的线程使用。

评论：本题较难，尤其是 b。

本题的背景，是用信号量模拟条件变量，所以 b 的解法实际上是用条件变量解决“生产者-消费者”问题。

删除部分代码，还可以得到用自旋锁的解法。

b 中的 while 是条件变量、管程中的经典问题。

本题中的代码均经过理论与实践双重验证。