1. 线程 API：给出三种以下程序可能的输出，假设系统调用都成功。

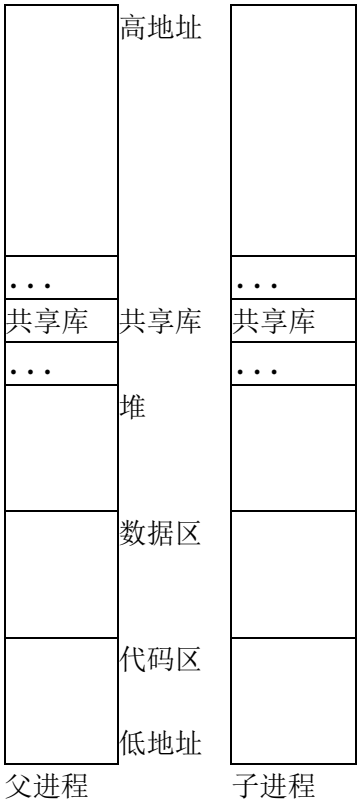
```
void subtask(void* args_) {
    int idx = (int) args_;
    printf("%d ", idx);
    return;
}

int main(void) {
    pthread_t threads[6];
    for (int i = 1; i <= 6; i++) {
        pthread_create(threads[i - 1], NULL, subtask, (void*) i);
        if (i % 3 == 0) {
            for (int j = i - 2; j <= i; j++) {
                pthread_join(threads[j - 1], NULL);
            }
        }
    }
    return 0;
}
```

2. **volatile** 保证定义的变量存放在内存中，而不总是在寄存器里。右侧为两个进程的地址空间。请在合适的位置标出变量 gCount、vCount 与 lCount 的位置。如果一个量出现多次，那么就标多次。

```
long gCount = 0;
void *thread(void *vargp) {
    volatile long vCount = *(long *)vargp;
    static long lCount = 0;
    gCount++; vCount++; lCount++;
    printf("%ld\n", gCount+vCount+lCount);
    return NULL;
}

int main() {
    long var; pthread_t tid1, tid2;
    scanf("%ld", &var);
    fork();
    pthread_create(&tid1,NULL,thread,&var);
    pthread_create(&tid2,NULL,thread,&var);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```



3. 下面的程序会引发竞争。一个可能的输出结果为 2 1 2 2。解释输出这一结果的原因。

```
long foo = 0, bar = 0;

void *thread(void *vargp) {
    foo++; bar++;
    printf("%ld %ld ", foo, bar); fflush(stdout);
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

4. 判断以下说法的正确性

( ) 在一个多线程程序中，其中一个线程主动调用 exit(0);只会导致该线程退出。
( ) 在同一进程中的两个线程 A 和 B，线程 A 不可以访问存储在线程 B 栈上的变量。
( ) 在同一进程中的两个线程 A 和 B 共享相同的堆，所以他们可以通过堆上的缓冲区完成线程间的通信。
( ) 在同一进程中的两个线程 A 和 B 共享相同的栈，所以他们可以通过栈上的缓冲区完成线程间的通信。
( ) 在进行线程切换后，TLB 条目绝大部分会失效
( ) 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
( ) 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
( ) printf()是线程安全函数

5. 考虑以下程序

```
#define WORMS 8

typedef struct {
    pthread_t tid;
    char *msg;
} pthread_args;

static void *spawn_worm(void *arg) {
    pthread_args *args = (pthread_args *)arg;
    char msg[100];
    // copies formatted string to MSG
    sprintf(msg, "Worm #%ld", args->tid);
```

```
    args->msg = msg;
}
int main() {
    pthread_args args;
    int s;
    for (int i = 0; i < WORMS; i++) {
        s = pthread_create(&args.tid, NULL, &spawn_worm, &args);
        if (s != 0) {
            return 1;
        }
    }
    for (int i = 0; i < WORMS; i++) {
        s = pthread_join(args.tid, NULL);
        if (s == 0) {
            printf("%s\n", args.msg);
        }
    }
    return 0;
}
```

你预期这个程序将给出如下输出：

```
ics@pku ~$ gcc silkworm.c - o silkworm - lpthread
ics@pku ~$./silkworm
Worm 1
Worm 2
Worm 3
Worm 4
Worm 5
Worm 6
Worm 7
Worm 8
ics@pku ~$
```

但实际上，运行时程序的输出是这样的：

```
ics@pku ~$./silkworm


ics@pku ~$
```

修改以上的程序使之能够确定地产生预期的输出。你可以修改最多 5 行代码。

6. 某次考试有 30 名学生与 1 名监考老师，该教室的门很狭窄，每次只能通过一人。考试开始前，老师和学生进入考场（有的学生来得比老师早），当人来齐以后，老师开始发放试卷。拿到试卷后，学生就可以开始答卷。学生可以随时交卷，交卷后就可以离开考场。当所有的学生都上交试卷以后，老师才能离开考场。

请用信号量与 PV 操作，解决这个过程中的同步问题。所有空缺语句均为 PV 操作。

<table>
<tr><td colspan="2">全局变量：
stu_count: int 类型，表示考场中的学生数量，初值为 0
信号量：
mutex_stu_count: 保护全局变量，初值为 1
mutex_door: 保证门每次通过一人，初值为_____
mutex_all_present: 保证学生都到了，初值为_____
mutex_all_handin: 保证学生都交了，初值为_____
mutex_test[30]: 表示学生拿到了试卷，初值均为_____</td></tr>
<tr><td>Teacher: // 老师
_____
从门进入考场
_____

_____ // 等待同学来齐
for (i = 1; i <= 30; i++)
_____ // 给 i 号学生发放试卷

_____ // 等待同学将试卷交齐
_____
从门离开考场
_____</td><td>Student(x): // x 号学生
_____
从门进入考场
_____

P(mutex_stu_count);
stu_count++;
if (stu_count == 30)
_____
V(mutex_stu_count);

_____ // 等待拿自己的卷子
学生答卷

P(mutex_stu_count);
stu_count--;
if (stu_count == 0)
_____
V(mutex_stu_count);

_____
从门离开考场
_____</td></tr>
</table>

7. 信号量 w,x,y,z 均被初始化为 1。下面的两个线程运行时可能会发生死锁。给出发生死锁的执行顺序。

线程 1	①P(w) ②P(x) ③P(y) ④P(z) ⑤V(w) ⑥V(x) ⑦V(y) ⑧V(z)
线程 2	ⅠP(x) ⅡP(z) ⅢP(y) ⅣP(w) ⅤV(x) ⅥV(y) ⅦV(w) ⅧV(z)

8. 竞争

以下几段代码创建两个对等线程，并希望第一个线程输出 0，第二个输出 1；但有些代码会因为变量 myid 的竞争问题导致错误，请你判断哪些代码会在 myid 上存在竞争。如果不存在竞争，请你判断这段代码是否一定先输出 0 再输出 1？

A.

```
void *foo(void *vargp) {
    int myid;
    myid = *(int *)vargp;
    free(vargp);
    printf("Thread %d\n", myid);
}
```

```
int main() {
    pthread_t tid[2];
    int i, *ptr;
    for (i = 0; i < 2; ++i) {
        ptr = malloc(sizeof(int));
        *ptr = i;
        pthread_create(&tid[i], 0,
foo, ptr);
    }
    pthread_join(tid[0], 0);
    pthread_join(tid[1], 0);
}
```

B.

```
void *foo(void *vargp) {
    int myid;
    myid = *(int *)vargp;
    printf("Thread %d\n", myid);
}
```

```
int main() {
    pthread_t tid[2];
    int i;
    for (i = 0; i < 2; ++i) {
        pthread_create(&tid[i], 0,
foo, &i);
    }
    pthread_join(tid[0], 0);
    pthread_join(tid[1], 0);
}
```

C.

```
void *foo(void *vargp) {
    int myid;
    myid = (int)vargp;
    printf("Thread %d\n", myid);
}
```

```
int main() {
    pthread_t tid[2];
    int i;
    for (i = 0; i < 2; ++i) {
        pthread_create(&tid[i], 0,
foo, (void *)i);
    }
    pthread_join(tid[0], 0);
    pthread_join(tid[1], 0);
}
```

D.

```
sem_t s;
```

```
int main() {
    pthread_t tid[2];
```

```
void *foo(void *vargp) {
    int myid;
    P(&s);
    myid = *(int *)vargp;
    V(&s);
    printf("Thread %d\n", myid);
}
```

```
    int i;
    sem_init(&s, 0, 1);
    for (i = 0; i < 2; ++i) {
        pthread_create(&tid[i], 0,
foo, &i);
    }
    pthread_join(tid[0], 0);
    pthread_join(tid[1], 0);
}
```

E.

```
sem_t s;

void *foo(void *vargp) {
    int myid;
    myid = *(int *)vargp;
    V(&s);
    printf("Thread %d\n", myid);
}
```

```
int main() {
    pthread_t tid[2];
    int i;
    sem_init(&s, 0, 0);
    for (i = 0; i < 2; ++i) {
        pthread_create(&tid[i], 0,
foo, &i);
        P(&s);
    }
    pthread_join(tid[0], 0);
    pthread_join(tid[1], 0);
}
```

9. 读者写者问题

一组并发的线程想要访问一个共享对象，有无数的读者和写者想要访问共享对象，读者可以和其它读者同时访问，而写者必须独占对象。以下是第一类读者写者问题的代码。

```
void reader() {
    P(&mutex);
    readcnt++;
    if (readcnt == 1)
        P(&w); /* line a */
    V(&mutex);
    /* reading... line b */
    P(&mutex);
    readcnt--;
    if (readcnt == 0)
        V(&w);
    V(&mutex);
}
```

```
void writer() {
    P(&w); /* line c */
    /* writing... line d */
    V(&w);
}
```

(1) 假设在时刻 0~4 分别有五个读、写者到来；它们的顺序为 R1, R2, W1, R3, W2；已知读操作需要等待 3 个周期，写操作需要等待 5 个周期；假设忽略其他语句的执行时间、线程的切换/调度的时间开销，因此在任意时刻，每个读者、写者只能处在上面标注好的 abcd 四处语句，请你分析这五个读者/写者线程终止的顺序？

(2) 基于(1)的发现，这段代码容易导致饥饿，于是一位同学规定：当有写者在等待时，后来的读者不能进行读操作，写出了第二类读者写者问题的代码如下（所有信号量初始化为 1）：

```
void reader() {
    P(&r); /* a */
    P(&mutex);
    readcnt++;
    if (readcnt == 1)
        P(&w); /* b */
    V(&mutex);
    V(&r);
    /* reading...  c */
    P(&mutex);
    readcnt--;
    if (readcnt == 0)
        V(&w);
    V(&mutex);
}
```

```
void writer() {
    P(&mutex);
    writecnt++;
    if (writecnt == 1)
        P(&r); /* d */
    V(&mutex);
    P(&w); /* e */
    /* writing... f */
    V(&w);
    P(&mutex);
    writecnt--;
    if (writecnt == 0)
        V(&r);
    V(&mutex);
}
```

这段代码会导致死锁，请你列举一种可能导致死锁的线程控制流，并提出一种改进的方案。

(3) 在修改了(2)中的问题后，请你基于第二类读者写者问题的代码再回答(1)中的题目。

10. 线程安全函数

吴用功同学找了一个找素数的函数 next_prime，ta 在实现这个函数的线程安全版本 ts_next_prime 的时候出现了问题，请你帮助 ta。

```
struct big_number *next_prime(struct big_number current_prime) {
    static struct big_number next;
    next = current_prime;
    addOne(next);
    while(!isNotPrime(next))
        addOne(next);
    return &next;
}

struct big_number *ts_next_prime(struct big_number current_prime) {
    return next_prime(current_prime);
}
```

A. 现在的 ts_next_prime 为什么线程不安全？

B. 下面的代码是否线程安全?

```
struct big_number *ts_next_prime(struct big_number current_prime)
{
    struct big_number *value_ptr;

    P(&mutex); /* mutex is initialized to 1*/
    value_ptr = next_prime(current_prime);
    V(&mutex);

    return value_ptr;
}
```

C. 请使用 lock&copy 技术实现线程安全的 ts_next_prime

```
sem_t mutex;
struct big_number *ts_next_prime(struct big_number current_prime)
{
    struct big_number *value_ptr;
    struct big_number *ret_ptr = ______________________;

    P(&mutex); /* mutex is initiallized to 1*/
    value_ptr = next_prime(current_prime);
    ____________________________________;
    V(&mutex);

    return ret_ptr;
}
```

得分

第六题．请结合教材第十二章**“并发编程”**的有关知识回答问题**(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**）

```
1./* Producer-Consumer Problem (Solution 1) */
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 t
        he buffer */
10.
11.     static sem_t mutex; /* Mutual Exclusion */
12.     static sem_t empty; /* How many empty slots are there
        in the buffer */
13.     static sem_t full;  /* How many items are there in the
         buffer */
14.
15.     static void sync_var_init() {
16.         Sem_init(&mutex, 0, 1);
17.
18.         /* Initially, there is no item in the buffer */
19.         Sem_init(&empty, 0, BUF_SIZE);
20.         Sem_init(&full, 0, 0);
21.     }
22.
23.     static void *producer(void *num) {
24.         ①;
25.         ②;
26.
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, produce
        r, (void *)i);
61.             Pthread_create(&pid_consumer[i], NULL, consume
        r, (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、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
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.}
```

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 t
        he 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 ite
        ms */
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, (vo
          id *)i);
79.         Pthread_create(&pid_consumer[i], NULL, consumer, (vo
          id *)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