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;  } |

**1 2 3 4 5 6**

**1 3 2 4 5 6**

**2 1 3 4 5 6**

**(只需要满足前三个数是1,2,3的一个排列，后三个数是4,5,6的一个排列即可)**

1. **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;  } |  | **vCount**  **vCount** | 高地址 | **vCount**  **vCount** |
| ... |  | ... |
| 共享库 | 共享库 | 共享库 |
| ... |  | ... |
|  | 堆 |  |
| **gCount**  **lCount** | 数据区 | **gCount**  **lCount** |
|  | 代码区  低地址 |  |
| 父进程 |  | 子进程 |

1. 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;  } |

**【答】线程1将foo、bar改为1以后被线程2打断，线程2将foo改为2以后被线程1打断，线程1输出了2 1，线程2将bar改为2，并输出了2 2。**

1. 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 |

1. 考虑以下程序

|  |  |
| --- | --- |
| 5  10  15  20  25  30 | #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;  } |

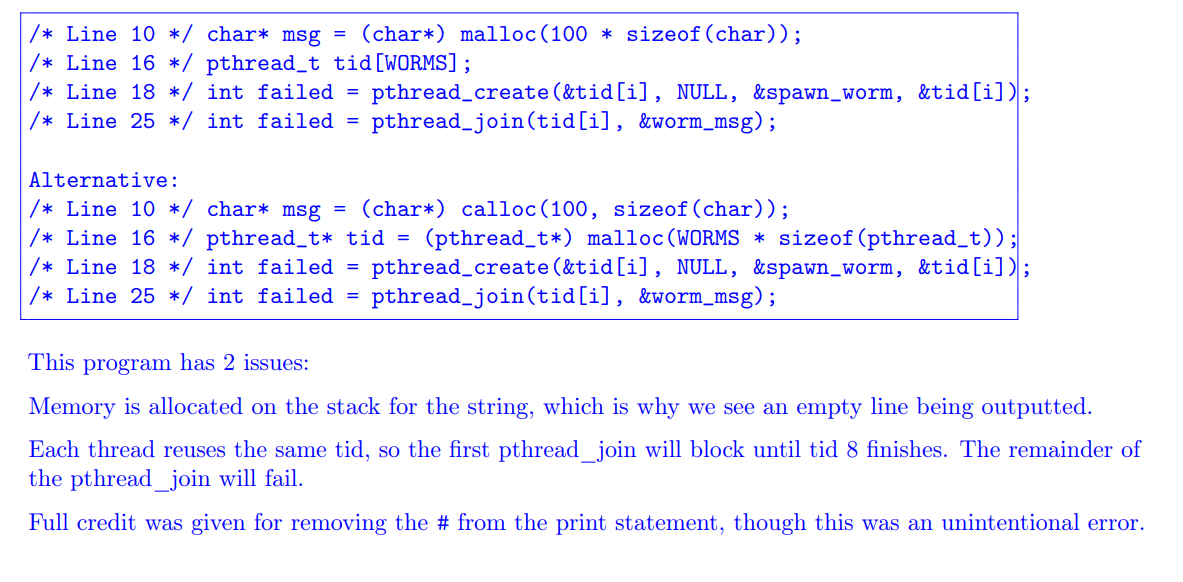
你预期这个程序将给出如下输出：

|  |
| --- |
| 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。



1. 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。

|  |  |
| --- | --- |
| 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**1**  mutex\_all\_present: 保证学生都到了，初值为**0**  mutex\_all\_handin: 保证学生都交了，初值为**0**  mutex\_test[30]: 表示学生拿到了试卷，初值均为**0** | |
| Teacher: // 老师  **P(mutex\_door)**  从门进入考场  **V(mutex\_door)**  **P(mutex\_all\_present)** // 等待同学来齐  for (i = 1; i <= 30; i++)  **V(mutex\_test[i])** // 给i号学生发放试卷    **P(mutex\_all\_handin)** // 等待同学将试卷交齐  **P(mutex\_door)**  从门离开考场  **V(mutex\_door)** | Student(x): // x号学生  **P(mutex\_door)**  从门进入考场  **V(mutex\_door)**  P(mutex\_stu\_count);  stu\_count++;  if (stu\_count == 30)  **V(mutex\_all\_present)**  V(mutex\_stu\_count);  **P(mutex\_test[i])** // 等待拿自己的卷子  学生答卷  P(mutex\_stu\_count);  stu\_count--;  if (stu\_count == 0)  **V(mutex\_all\_handin)**  V(mutex\_stu\_count);  **P(mutex\_door)**  从门离开考场  **V(mutex\_door)** |

1. 死锁

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) |

**【答】①→Ⅰ→Ⅱ→Ⅲ，此时线程1占用了w而在等待x，线程2占用了x而在等待w。**

**这两道题对于信号量的操作都没有带“&”，期末考试的时候看上下文，可能对于信号量的操作需要带“&”。**

1. 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？

1. 不会，因为两个线程myid对应heap中不同位置的变量

|  |  |
| --- | --- |
| **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);  } |

1. 存在竞争，两个myid都是对main函数堆栈中i的引用

|  |  |
| --- | --- |
| **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);  } |

1. 不存在竞争，因为创建线程传递的是值而非指针。

|  |  |
| --- | --- |
| **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);  } |

1. 存在竞争，两个对等线程和主线程都会访问i，即使在对等线程中加了互斥锁进行保护，但是主线程仍然可以修改vargp对应的内存（也就是i）的值

|  |  |
| --- | --- |
| sem\_t s;  **void** \*foo(**void** \*vargp) {  **int** myid;  P(&s);  myid = \*(int \*)vargp;  V(&s);  printf("Thread %d\n", myid);  } | **int** main() {  pthread\_t tid[2];  **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);  } |

1. 不存在竞争，因为这段代码实现了同步，三个线程对i的访问实现了完全互斥，同时这段代码也保证先输出0再输出1

|  |  |
| --- | --- |
| 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);  } |

1. 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？

R1R2R3W1W2 or R1R2R3W2W1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 时刻 | R1 | R2 | W1 | R3 | W2 |
| 0 | b |  |  |  |  |
| 1 | b | b |  |  |  |
| 2 | b | b | c |  |  |
| 3 | - | b | c | b |  |
| 4 | - | - | c | b | c |
| 5 | - | - | c | b | c |
| 6 | - | - | c/d | - | d/c |

根据上表分析可知两个写者将等待最后一个读者退出执行V(&w)语句才得以继续执行；由于V操作会随机唤醒一个睡眠在P操作的线程，所以W1和W2的执行顺序不可知。

(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。

初始时刻某个读者和写者同时到来，读者执行P(&r)，写者执行P(&mutex)，然后两个线程都无法继续执行，后来的线程也会阻塞在第一个P操作。

解决方案：1.使用不同的信号量实现对readcnt和writecnt的互斥保护；2.将读者线程V(&r)移到P(&r)的后面；3. 调换a处和下一行P(&mutex)的顺序。

(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。

R1R2W1W2R3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 时刻 | R1 | R2 | W1 | R3 | W2 |
| 0 | c |  |  |  |  |
| 1 | c | c |  |  |  |
| 2 | c | c | e |  |  |
| 3 | - | c | e | a |  |
| 4 | - | - | f | a | e |
| 5 | - | - | f | a | e |
| 6 | - | - | f | a | e |
| 7 | - | - | f | a | e |
| 8 | - | - | f | a | e |
| 9 | - | - | - | a | f |

1. 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);  } |

1. 现在的ts\_next\_prime为什么线程不安全？

返回了一个指向静态变量的指针

1. 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;  } |

并不安全，在V语句后，其他线程也可以调用next\_prime，进而导致value\_ptr静指向的静态变量next被修改。

1. 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;  } |

(struct big\_number \*)malloc(sizeof(struct big\_number))

memcpy(ret\_ptr, value\_ptr, sizeof(struct big\_number))

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：

1. 所有生产者和所有消费者**共享同一个buffer**
2. 生产者、消费者各有NUM\_WORKERS个（**大于一个**）
3. buffer的容量为BUF\_SIZE，**初始情况下buffer为空**
4. 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
5. 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
6. 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**）
7. */\* Producer–Consumer Problem (Solution 1) \*/*
8. #include "csapp.h"
9. #define BUF\_SIZE 3
10. #define NUM\_WORKERS 50
11. #define MAX\_SLEEP\_SEC 10
12. volatile static int items = 0; */\* How many items are there in the buffer \*/*
13. static sem\_t mutex; */\* Mutual Exclusion \*/*
14. static sem\_t empty; */\* How many empty slots are there in the buffer \*/*
15. static sem\_t full;  */\* How many items are there in the buffer \*/*
16. static void sync\_var\_init() {
17. Sem\_init(&mutex, 0, 1);
18. */\* Initially, there is no item in the buffer \*/*
19. Sem\_init(&empty, 0, BUF\_SIZE);
20. Sem\_init(&full, 0, 0);
21. }
22. static void \*producer(void \*num) {
23. ①;
24. ②;
25. */\* Critical section begins \*/*
26. Sleep(rand() % MAX\_SLEEP\_SEC);
27. items++;
28. */\* Critical section ends \*/*
29. V(&mutex);
30. V(&full);
31. return NULL;
32. }
33. static void \*consumer(void \*num) {
34. ③;
35. ④;
36. */\* Critical section begins \*/*
37. Sleep(rand() % MAX\_SLEEP\_SEC);
38. items--;
39. */\* Critical section ends \*/*
40. V(&mutex);
41. V(&empty);
42. return NULL;
43. }
44. int main() {
45. sync\_var\_init();
46. pthread\_t pid\_producer[NUM\_WORKERS];
47. pthread\_t pid\_consumer[NUM\_WORKERS];
48. for (int i = 0; i < NUM\_WORKERS; i++) {
49. Pthread\_create(&pid\_producer[i], NULL, producer, (void \*)i);
50. Pthread\_create(&pid\_consumer[i], NULL, consumer, (void \*)i);
51. }
52. for (int i = 0; i < NUM\_WORKERS; i++) {
53. Pthread\_join(pid\_producer[i], NULL);
54. Pthread\_join(pid\_consumer[i], NULL);
55. }
56. }
    1. 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：

1. P(&mutex)
2. P(&empty)
3. P(&full)
   1. 如果交换24行与25行（两个P操作）， （单选，2分）
      * 1. 有可能死锁
        2. 有可能饥饿
        3. 既不会死锁，也不会饥饿
   2. 交换32行与33行（两个V操作）是否可能造成同步错误？ （2分）
      * 1. 可能
        2. 不可能
   3. rand函数是不是线程安全的？ （1分）
      * 1. 是
        2. 不是

28行与43行对rand函数的使用是否会导致竞争？ （1分）

1. 会
2. 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. unsigned long \_Randomseed = 1;
3. int rand() {
4. \_Randomseed = \_Randomseed \* 1103515425 + 12345;
5. return (unsigned int)(\_Randomseed >> 16) & RAND\_MAX;
6. }
7. void srand(unsigned int seed) {
8. \_Randomseed = seed;
9. }

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。

1. 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**）
2. */\* Producer–Consumer Problem (Solution 2) \*/*
3. #include "csapp.h"
4. #define BUF\_SIZE 3
5. #define NUM\_WORKERS 50
6. #define MAX\_SLEEP\_SEC 10
7. volatile static int items = 0; */\* How many items are there in the buffer \*/*
8. static sem\_t mutex;                */\* Mutual Exclusion \*/*
9. static sem\_t sem\_waiting\_producer; */\* Wait for empty slots \*/*
10. static sem\_t sem\_waiting\_consumer; */\* Wait for available items \*/*
11. volatile static int num\_waiting\_producer = 0;
12. volatile static int num\_waiting\_consumer = 0;
13. static void sync\_var\_init() {
14. Sem\_init(&mutex, 0, 1);
15. Sem\_init(&sem\_waiting\_producer, 0, ①);
16. Sem\_init(&sem\_waiting\_consumer, 0, ①);
17. }
18. static void \*producer(void \*num) {
19. P(&mutex);
20. while (items == BUF\_SIZE) {
21. num\_waiting\_producer++;
22. ②;
23. ③;
24. P(&mutex);
25. }
26. */\* Critical section begins \*/*
27. Sleep(rand() % MAX\_SLEEP\_SEC);
28. items++;
29. */\* Critical section ends \*/*
30. if (num\_waiting\_consumer > 0) {
31. num\_waiting\_consumer--;
32. V(&sem\_waiting\_consumer);
33. }
34. V(&mutex);
35. return NULL;
36. }
37. static void \*consumer(void \*num) {
38. P(&mutex);
39. while (items == 0) {
40. num\_waiting\_consumer++;
41. ④;
42. ⑤;
43. P(&mutex);
44. }
45. */\* Critical section begins \*/*
46. Sleep(rand() % MAX\_SLEEP\_SEC);
47. items--;
48. */\* Critical section ends \*/*
49. if (num\_waiting\_producer > 0) {
50. num\_waiting\_producer--;
51. V(&sem\_waiting\_producer);
52. }
53. V(&mutex);
54. return NULL;
55. }
56. int main() {
57. sync\_var\_init();
58. pthread\_t pid\_producer[NUM\_WORKERS];
59. pthread\_t pid\_consumer[NUM\_WORKERS];
60. for (int i = 0; i < NUM\_WORKERS; i++) {
61. Pthread\_create(&pid\_producer[i], NULL, producer, (void \*)i);
62. Pthread\_create(&pid\_consumer[i], NULL, consumer, (void \*)i);
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. }
    1. 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：

* + - 1. 0
      2. 1
      3. P(&sem\_waiting\_producer)
      4. V(&mutex)
  1. 如果27行和50行的while换成if，是否可能造成同步错误？ （2分）
     + 1. 可能
       2. 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。