BUG Patterns

This exam is about **B. U. G. Patterns**, a famously bad programmer. "Bug", as this programmer is affectionately called, usually introduces problems into the code, which Bug's coworkers end up fixing. Unfortunately, not all the fixes got fixed, so you have to step in and do some work. Even worse: it's all concurrent code with threads!

For all questions, assume:

- **thread_create()** creates a thread and takes three arguments (a pointer to a thread_t type, a function pointer to the child thread's code, and a single argument)
- the child thread does not return any value (for simplicity)
- **thread_join()** waits for a thread to complete and takes one argument (the thread_t type previously filled in by thread_create)
- **mutex_t** is a lock
- **mutex_acquire()** and **mutex_release()** acquire and release the lock, respectively
- **cond_t** is a condition variable
- **cond_wait()** and **cond_signal()** wait and signal on the condition, respectively
- **sem_t** is a semaphore
- **sem_init()** initializes the semaphore; **sem_wait()** and **sem_post()** do what you'd expect
- Assume calls (like **thread_create()**, **malloc()**, or any other code that is called but not specified) **never fail**, unless otherwise specified.
- Assume that if a mutex or condition variable or semaphore is used, it is properly initialized, unless otherwise specified.
- A **null dereference** reliably crashes code

Can you fix Bug's many problems? Let's hope so!

Each of 32 questions has **exactly one answer**: **a, b, c, d** or **e**. Make sure to fill them all out!

Good luck!

Question 1.

Bug wrote the following code:

```
int i = 0; // global variable
```

Two threads run through this code:

```
if (i == 0)
    i++;
```

What are the possible value of `i` after both threads run:

a) 0

[✓ b) 1]

c) 2

[✓ d) 1 or 2]

e) None of the above

If you assume an assembly sequence like this:

```
load i -> reg
check if reg is 0
if so {
  increment reg;
  store reg -> i
}
```

This will always get 1, because even if both threads enter into the if statement, they will both store 1 into i at the end.

If you assume an assembly sequence like this:

```
load i->reg
check if reg is 0
if so {
  load i->reg
  inc reg
  store reg->i
}
```

This can get 1 or 2. It will get 1 if (say) one thread does the sequence first; later threads will see i isn't 0 and skip the increment. It can get 2 if they both check and see i is 0 and enter the i++ part of the code. At that point, one could do the increment, then the other, resulting in 2.

Thus, we accept b or d.

Question 2.

Bug then tried to fix the code with a mutex.

```
// global variables
mutex_t m;
int i = 0;
```

Two threads run through this code:

```
if (i == 0) {
    mutex_acquire(&m);
    i++;
    mutex_release(&m);
}
```

The lock doesn't help here, so same as above.

Possible value of `i` after both threads run:

a) 0

[✓ b) 1]

c) 2

[✓ d) 1 or 2]

e) None of the above

Question 3.

Bug was playing around with the following code, and decided to change how locking was done. This is what Bug's code ended up looking like:

```
mutex_t m;
int slot = -1;  // Note: Changed from 0 to -1 on board during exam
int array[2] = { 0, 0 }; // initialize to 0, 0 contents

function1() {
    mutex_acquire(&m);
    slot++;
    mutex_release(&m);
    int tid = get_counter();
    mutex_acquire(&m);
    array[slot] = tid;
    mutex_release(&m);
}
```

Assume that `get_counter()` has its own internal locking (not shown), and will return 1 when first called, and 2 when called next, etc.

Assume that `function1()` is called by two threads at roughly the same time. What are the final contents of the array?

a) [0, 0]
b) [0, 1]
c) [1, 2]
d) [0, 2]
[✓ e)] Indeterminate; more than one answer is possible

Case 1: One thread runs to completion, then the next. The first would get slot=0 and put 1 in it; the second would get slot = 1 and put 2 in it. Thus [1, 2] is possible.

But, now imagine thread 0 going through the first acquire/release; it gets slot=0. Then, thread 1 goes through and gets slot=1. But now, thread 1 continues first, and calls get_counter() first. Thus, it gets 1 for its tid value, and thread 0 gets 2. Thus, [2, 1] is possible.

As such, answer is e - indeterminate.

Question 4.

Bug decided to fix the code above as follows:

```
function1() {
    mutex_acquire(&m);
    slot++;
    int tid = get_counter();
    array[slot] = tid;
    mutex_release(&m);
}
```

In this case, the first thread through always populates the 0th entry of the array with 1; the second thread through always puts 2 into the 1st entry. Thus [1, 2] is the final content of the array.

Now what are the final contents of the array?
a) [0, 0]
b) [0, 1]
[✓ c) [1, 2]]
d) [0, 2]
e) Indeterminate; more than one answer is possible

Question 5.

Bug wrote the following code to initialize a list.

```
// global
List_t *L = NULL;
```

Assume two threads run through the following code sequence:

```
if (L == NULL) {
    L = malloc(sizeof(List_t));
    List_Init(&L); // this just sets L->head = NULL;
}
```

What are the possible outcomes:
a) The code crashes
b) Memory may be leaked
c) The list is properly initialized (once) and the code works as expected
[✓ d) Both b and c are possible]
e) Each of a, b, and c are possible

This code could just work; imagine one thread runs first, checks for L==NULL, finds that it is true, allocates space for the List_t, and then initializes the list. The second thread then runs, finds L is no longer NULL, and continues. Thus, c is possible.

However, if both threads check for L==NULL at the same time, and find that it is, they could both allocate memory for the List_t, and each call List_Init(). The last writer of L will be used from then on; the first allocation of L will be lost. Thus, b is possible.

As stated in the assumption on the first page, malloc() will not fail, so the code should be fine otherwise. Thus, d is the answer.

Question 6.

Bug is trying to remember how to use a semaphore to wait for a child thread to complete. This is the code Bug writes:

```
sem_t s;

void child(void *arg) {
    // do some stuff, then signal completion
    sem_post(&s)
}

int main(int argc, char *argv[]) {
    thread_t p1;
    sem_init(XXX);
    thread_create(&p1, child);
    sem_wait(&s);
    return 0;
}
```

```
// Definition (from class)
// note: functionally equivalent to textbook:
sem_wait() {
   while (value <= 0)
      put_self_to_sleep();
   value—;

// Definition (changed for some during exam, alas):
sem_wait() {
   while (value < 0)
      put_self_to_sleep();
   value—;
```

But Bug can't remember: What should the semaphore **s** be initialized to?

[✓ a) 0]
b) 1
c) 2
d) 3
[✓ e) None of the above]

Given the textbook or class definition of the semaphore, we should init the semaphore to 0. Thus ensures that the parent will wait until the child is done. Thus, a.

Given the definition given to some people in one exam room, the semaphore should be initialized to -1. Thus, e is possible.

Question 7.

Assume the same code as Q6 above, except this time, the parent creates three children.

```
int main(int argc, char *argv[]) {
    thread_t p1, p2, p3;
    sem_init(XXX);
    thread_create(&p1, child);
    thread_create(&p2, child);
    thread_create(&p3, child);
    sem_wait(&s);
}
```

What should the semaphore **s** be initialized to?

a) 0
b) 1
c) 2
d) 3
[✓ e) None of the above]

The semaphore should be initialized to -2. In this case, after the first post (in the child), it goes to -1 (still waits). After the second, it goes to 0 (still waits); finally, after the third post, it goes to 1, and the parent will wake up and return from wait. Thus, e is the answer.

With alternate definition, would be -3. Inc'd to -2, -1, then 0. Answer still e in this case.

Question 8.

Bug knows you can use semaphores as locks, and locks as locks. So why not use both?

```
int count = 0;
mutex_t m;
sem_t s; // the semaphore s is initialized to 1 (not shown)
```

There are two threads. One thread uses a lock, the other a semaphore, and they run concurrently.

```
thread 1:
mutex_acquire(&m);
count++;
mutex_release(&m);

thread 2:
sem_wait(&s);
count++;
sem_post(&s);
```

We could have one thread run to completion, then the other. In this case, we get 2. But, we could have each thread enter the critical section, load 0->reg, increment it, and store 1 back to count (twice). Thus, 1 is possible. Thus, 1 or 2 is possible. Using a lock and a semaphore to protect the same critical section doesn't prevent this data race.

The final value of count is:
a) 0
b) 1
c) 2
[✓ d) 1 or 2]
e) None of the above

Question 9.

Bug wants to do some multiplying. A global variable `count` is initialized to **100**. Two threads each run the following:

```
count = count * 2;
```

What could the final value of count be? Assume the multiply instruction works on registers, i.e., the value must be loaded from memory into a register, then multiplied, then stored back to memory.

a) 100
b) 200
c) 400
[✓ d) 200 or 400]
e) None of the above

We could have one thread run first in entirety, thus changing count from 100 to 200. The next thread would then run, changing it from 200 to 400. Thus, 400 is possible. However, if both load the value 100 and the multiply it "at the same time", each could have 200 as a result, and overwrite count with 200 (twice, total). Thus, 200 is possible. Thus, d.

Question 10.

This time, Bug remembers to use a mutex. The `count` again starts at **100**, and two threads each run the following code:

```
mutex_acquire(&m);
count = count * 2;
mutex_release(&m);
```

What could the final value of count be?

a) 100
b) 200
[✓ c) 400]
d) 200 or 400
e) None of the above

This code ensures that one multiply happens in entirety before the other. Thus, count goes from 100->200, and then 200->400. Thus, 400.

Question 11.

Now Bug is back to using semaphores. Maybe Bug is figuring things out for a change?

```
int count = 0;
sem_t s; // semaphore s is initialized to 1 (not shown)
```

Two threads run the following code:

```
sem_wait(&s);
count--;
sem_post(&s);
```

What is the final value of **`count`**?

a) 0
b) -1
[✓ c) -2]
[✓ d) -1 or -2]
e) None of the above

Correct answer means this semaphore is being used as a lock. Thus, count properly gets decremented twice, and is thus -2 (answer c).

However, with the "alternate" definition, both threads could enter the critical section at the same time. In this case, you could get the right answer (-2), but you could also get each setting the value of count to -1. Thus, d is possible.

Question 12.

Bug then starts playing around with the initial value of the semaphore. That seems like fun! At least, that is what Bug thinks. The `count` is 0 to begin, and the semaphore `s` initialized to 0.

Two threads run the following code:

```
sem_wait(&s);
count--;
sem_post(&s);
```

If the semaphore starts at 0, a thread calling wait will get stuck waiting. In this case, both threads will get stuck, and thus count remains 0.

With the alternate definition, this works as a lock. This means count will reliably be -2 after the threads run. Thus, c.

What is the final value of **`count`**?

[✓ a) 0]
b) -1
[✓ c) -2]
d) -1 or -2
e) None of the above

Question 13.

One last variation. Here, Bug initializes `count` to 0 again, and the semaphore `s` to 2.

Two threads run the following code:

```
sem_wait(&s);
count--;
sem_post(&s);
```

With either definition, this initialization allows two threads into the critical section at the same time. Thus, it might work (count set to -2), or you might get each loading the value (0) into a register, decrementing it (-1), and then each storing -1 into count. Thus, -1 or -2 (d).

What is the final value of **`count`**?

a) 0
b) -1
c) -2
[✓ d) -1 or -2]
e) None of the above

Question 14.

In this code snippet, Bug can't remember when to set global variable `p` to the address of `x`. What happens?

```
int *p = NULL; // global

void child(void *arg) {
    printf("%d\n", *p);
}

int main(int argc, char *argv[]) {
    thread_t p1;
    int x = 3;
    thread_create(&p1, child, NULL);
    p = &x;
    return 0;
}
```

The code could definitely crash; p is NULL to begin, and the child thread may get created, run immediately, dereference p, and crash.

The code could also print 3. How? The parent creates the child, then sets p to point to the address of x, then the child runs and prints 3.

Finally, the parent could run, create the child, set p to the address of x, and then exit, printing nothing.

Thus, d is the answer.

What will this code print?

a) 3
b) It won't print anything and exit
c) It will crash (not print)
[✓ d)] 3, or nothing, or it will crash
e) None of the above

Question 15.

Now Bug remembers sometimes `thread_join()` helps. Bug also mucks around with the code a bit, so read carefully. That Bug, always changing code!

```
void child(void *arg) {
    int *p = (int *) arg;
    printf("%d\n", *p);
}

int main(int argc, char *argv[]) {
    int x = 3;
    thread_t p1;
    thread_create(&p1, child, &x);
    thread_join(p1);
    return 0;
}
```

The child is given the address to the stack variable x in the parent; this is guaranteed to be live when the child runs because the parent properly waits for the child to complete. Thus, it will print the value 3.

What will this code print?

[✓ a) 3]
b) It won't print anything and exit
c) It will crash (not print)
d) 3, or nothing, or it will crash
e) None of the above

REPEATED HERE for simplicity:

The child is given the address to the stack variable x in the parent; this is guaranteed to be live when the child runs because the parent properly waits for the child to complete. Thus, it will print the value 3.

Question 16.

Bug tries to solve the fork/join problem. But something isn't quite right, is it?

```
cond_t c;

void child(void *arg) {
    printf("child\n");
    cond_signal(&c);
}

int main() {
    thread_t p1;
    thread_create(&p1, child, NULL);
    cond_wait(&c);
    printf("done\n");
    return 0;
}
```

What will this code print?

a) Just `child`
b) `child` then `done`
[✓ c) Just `child` OR `child` then `done`]
d) Just `done`
[✓ e) None of the above]

Assume the parent runs, then waits. Then the child runs, prints child, wakes the parent; the parent then returns from sleeping, prints done. Thus (child then done) is possible.

Assume the parent creates the child which runs immediately, prints child, and then signals. The parent then waits and is now stuck forever. Thus, (just child) is possible.

Thus, c.

However, you could argue that cond_wait() should take a mutex as a parameter. Thus, the code wouldn't compile. Thus, e.

Question 17.

Bug tries again, but adds an old friend) the call to `sleep()`. The `sleep()` routine just puts the calling thread to sleep for the specified amount of seconds.

```
cond_t c;
void child(void *arg) {
    printf("child\n");
}

int main(int argc, char *argv[]) {
    thread_t p1;
    thread_create(&p1, child, NULL);
    sleep(1); // sleeps for one second
    printf("done\n");
    return 0;
}
```

What will this code print?

a) Just `child`
b) `child` then `done`
c) `child` then `done` OR `done` then `child`
d) Just `done`
[✓ e) None of the above]

Likely case: parent creates child, which prints child. Parent sleeps. Parent wakes and prints done. Thus (child then done).

However, the parent could run, sleep, then print done (for some reason the child is taking a long time), and then the child could run. Thus, (done then child).

Parent, however, could print done and then exit, causing the entire process to exit. Thus, just (done) is possible. As such, e.

Question 18.

Bug wants to write a simple spin lock.

```
int m = 0; // global

void lock_acquire(int m) {
    while (xchg(&m, 1) == 0)
        ;
}

void lock_release(int m) {
    xchg(&m, 0);
}
```

What is wrong with Bug's code?

a) Nothing; it works!
b) It will spin forever
c) The release does not free the lock
d) The code will crash
[✓ e) None of the above]

m is passed in a parameter. Thus, each thread that calls will atomically set its own parameter to 1, and thus eventually pop out of the while loop, and thus the lock doesn't work (it allows both threads into the critical section). Thus, e is one possible answer.

If however you were told to remove the parameter (and thus m is only a global variable), you still have problems. The first thread to call lock_acquire will test it (and find that it is zero), but set it to 1 (that is what xchg does). Then, it will test it again and find that the value is 1. Thus, this thread can enter the critical section. The next thread will test it (finding it is 1), set it to 1, and also enter the critical section. Thus, this code does not provide mutual exclusion, either. Thus, e.

Question 19.

Bug tries for a spin lock again. Can Bug make it work?

```
int m = 0; // global

void lock_acquire(int m) {
    while (xchg(&m, 1) == 1)
        ;
}

void lock_release(int m) {
    xchg(&m, 0);
}
```

m is passed in a parameter. Thus, each thread that calls will atomically set the parameter to 1, and thus the lock doesn't work (each thread can enter the critical section). Thus, e is one possible answer.

If however you were told to remove the parameter (and thus m is only a global variable), this is a working lock. Thus, a is possible.

What is wrong with this code?

[✓ a) Nothing; it works!]
b) It will spin forever
c) The release does not free the lock
d) The code will crash
[✓ e) None of the above]

Question 20.

Now Bug is so confused people are making Bug review the basics.

```
void mythread(void *arg) {
    return;
}
thread_t p1, p2;
thread_create(&p1, mythread, NULL);
thread_create(&p2, mythread, NULL);
thread_join(p1);
thread_join(p2);
```

When this code runs, how many total threads can there be (maximum) at a given moment in time?

a) 1
b) 2
[✓ c) 3]
d) 4
e) None of the above

The parent creates two children. At the moment where the children exist, and the parent exists, there are three threads. Thus, c.

Question 21.

Bug tries to add some fun to this code by doing some math. But math is tricky, Bug.

```
void mythread(void *arg) {
    int result = 0;
    result = result + 200;
    printf("result %d\n", result);
}

thread_t p1, p2;
thread_create(&p1, mythread, NULL);
thread_create(&p2, mythread, NULL);
thread_join(p1);
thread_join(p2);
```

When this code runs, and `result` is printed, what value will be printed?

a) 0
[✓ b) 200]
c) 400
d) indeterminate; there is a race condition
e) Some other constant value

result is local to each thread. Thus, when it gets printed, it will print out 200. (yes, 200 will be printed twice, but still it's the best answer).

Question 22.

Bug realizes it might be more fun to update a global variable, `balance`.

```
int balance = 0; // global

void mythread(void *arg) {
    balance = balance + 200;
}

thread_t p1, p2;
thread_create(&p1, mythread, NULL);
thread_join(p1);
thread_create(&p2, mythread, NULL);
thread_join(p2);
```

There is no race, because first the child p1 runs to completion (and updates balance to 200), and then the next child (p2) runs to completion (which increments it to 400. Thus, c.

When this code runs, what is the final value of `balance`?

a) 0
b) 200
[✓ c) 400]
d) indeterminate; there is a race condition
e) Some other constant value

Question 23.

Now Bug flips around some joins. That won't matter much, Bug thinks. Or will it?

```
void mythread(void *arg) {
    balance = balance + 200;
}
thread_t p1, p2;
thread_create(&p1, mythread, NULL);
thread_create(&p2, mythread, NULL);
thread_join(p1);
thread_join(p2);
```

When this code runs, what is the final value of `balance`?

a) 0
b) 200
c) 400
[✓ d)] indeterminate; there is a race condition
e) Some other constant value

Here there is a race to update balance, because both children are active and trying to update balance at the same time.

Question 24.

Bug was told to speed up `function2()`. There were two locks (`outer` and `inner`), so Bug unlocked the `outer` lock so as to allow more concurrency. Could this lead to a problem?

```
function1() {
    mutex_lock(&outer);
    mutex_lock(&inner);
    function2();
    mutex_unlock(&inner);
    mutex_unlock(&outer);
}

function2() {
    mutex_unlock(&outer);
    // do some stuff
    mutex_lock(&outer);
}
```

If one thread runs first and runs to completion before the other thread really does anything, the code "works" (both threads run to completion). But, imagine this scenario: thread1 runs, grabs the outer lock, then the inner, then calls function2, and releases the outer lock. Then, thread2 runs, grabs the outer lock. At this point, the code could deadlock (t1 has the inner, will soon try to reacquire the outer; t2 has the outer, will soon try to grab the inner). Thus, b.

Assuming two (or more) threads call into `function1()`, this code.... :

a) will always run to completion
[✓ b)] can deadlock
c) will deadlock
d) will livelock
e) None of the above

Question 25.

This lock is implemented in a funny way. Thanks, Bug! Read the code carefully.

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    lock->flag = FREE;
}

void acquire(lock_t *lock) {
     while(xchg(&lock->flag, HELD) == HELD)
         ; // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = 1;
}
```

What value should **FREE** be in `init()`, for this code to work as a mutual exclusion primitive (i.e., a lock)?

a) 0
[✓ b) 1]
c) 2
d) 0 or 2
e) 0 or 1

Because release sets the flag to 1, 1 naturally means the lock is free. Thus, FREE should be set to 1, and is the best answer.

Question 26.

Assume the same code as in question 25. What should **HELD** be set to in `acquire()`, for the code to work as a mutual exclusion primitive (i.e., a lock)?

a) 0
b) 1
c) 2
[✓ d) 0 or 2]
e) 0 or 1

Basically, HELD needs to be something that is not equal to FREE. As such, 0 or 2 both work. Thus, d.

Question 27.

Poor Bug has to understand this reader/writer lock (it's the same one as in class). But Bug has to initialize those semaphores…

```
typedef struct _rwlock_t {
    sem_t lock;
    sem_t writelock; int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, LOCK_INIT);
    sem_init(&rw->writelock, WRITELOCK_INIT);
}
void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(&rw->writelock);
    sem_post(&rw->lock);
}
void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers--;
    if (rw->readers == 0)
        sem_post(&rw->writelock);
    sem_post(&rw->lock);
}
rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->writelock);
}
```

As per class/book, the lock protects the critical section inside the rwlock read acquire and release. Thus, to use it as a lock, the answer is 1.

With the alternate definition of the semaphore, it should be 0.

What should the value of **`LOCK_INIT`** be?

[✓ a) 0]
[✓ b) 1]
c) 2
d) N (where N is the number of readers you wish to allow)
e) None of the above

Question 28.

Same as Question 28, except what should the value of **`WRITELOCK_INIT`** be?

[✓ a) 0]
[✓ b) 1]
c) 2
d) N (where N is the number of readers you wish to allow)
e) None of the above

As per class/book, the lock ensures only one writer. Thus, the answer is 1.

With the alternate definition of the semaphore, it should be 0.

Question 29.

Now Bug has to make a List implementation work with threads. Here is the original code:

```
typedef struct node {
    int key;
    node_t *next;
} node_t;

typedef struct {
    node_t *head;
} list_t;

void List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t)); // L1
    new->key = key;                       // L2
    new->next = L->head;                  // L3
    L->head = new;                        // L4
}
```

Assume malloc() is thread safe. Assume that someone told Bug to put an `mutex_unlock()` call after Line L4. Thus, the question for Bug: where is the **best** place to put the call to `mutex_lock()`?

a) right before L1
b) right before L2
[✓ c) right before L3]
d) right before L4
e) None of the above

The key part of the code is assigning next to head, and head to the new node. Those two steps must happen atomically. As such, right before L3 is best. Earlier just reduces concurrency.

Question 30.

Bug is getting tired. But it's never too late for some more buggy code. Here, Bug tries to write an atomic increment routine, using the `compare_and_swap` instruction.

```
void increment(int *p) {
    do {
        int v = *p;                              // L1
        Int n = v + 1;                           // L2
    } while (compare_and_swap(p, v, n) == 0);    // L3
}
```

If **three threads** call increment on the same variable at roughly the same time, how many times might line L2 be executed?

a) 3
b) 4
c) 5
[✓ d) 6]
e) None of the above

This question is asking for a max - how many times might L2 be executed in the worst case? If three threads enter at the same time, each could execute L2, which means three times, each trying to swap in (v+1). Two will fail, and retry from the top. In the right situation, each will try to swap in (v+2), thus two more times. One will fail, and retry, and finally succeed. Thus, 6 total max.

Question 31.

Bug now has to confront the dreaded producer/consumer problem. The producer code:

```
mutex_lock(&m);                    // P1
while (numfull == max)             // P2
    cond_wait(&cond, &m);          // P3
do_fill(i);                        // P4
cond_signal(&cond);                // P5
mutex_unlock(&m);                  // P6
```

The consumer code looks like this:

```
mutex_lock(&m);                    // C1
while (numfull == 0)               // C2
    cond_wait(&cond, &m);          // C3
int tmp = do_get();                // C4
cond_signal(&cond);                // C5
mutex_unlock(&m);                  // C6
```

This code only has a single condition. As such, a producer could wrongly wake a producer; similarly, a consumer could wrongly wake a consumer. Thus, b and c are not right (they talk about the case of one type of thread waking the OTHER type, which is usually fine). Using while is correct (always good to recheck the condition), thus a is not right. Finally, producers should wait when numfull==max, and similarly consumers when numfull==0. Thus, b is not right. As such, e is the best answer.

Now Bug thinks… hmm… there is something wrong here. What is it?

a) Code uses while loop instead of if statement
b) Lines C2 and P2 are switched
c) Producer can incorrectly wake a consumer
d) Consumer can incorrectly wake a producer
[✓ e) None of the above]

Question 32.

Now Bug decides to “fix” the code.

The producer code looks like this:

```
mutex_lock(&m);                 // P1
while (numfull == max)          // P2
    cond_wait(&bug, &m);        // P3
do_fill(i);                     // P4
cond_signal(&patterns);         // P5
mutex_unlock(&m);               // P6
```

The consumer code looks like this:

```
mutex_lock(&m);                 // C1
while (numfull == 0)            // C2
    cond_wait(&patterns, &m);   // C3
int tmp = do_get();             // C4
cond_signal(&bug);              // C5
mutex_unlock(&m);               // C6
```

What is wrong with the code now? (from a correctness standpoint)

a) Too much signaling and waiting
b) Signal signals the wrong CV; wait waits on the wrong one
c) Producer can incorrectly wake a producer
d) Consumer can incorrectly wake a consumer
[✓ e)] None of the above

This uses two conditions (the consumer waits on patterns, and signals on bug; the producer does the opposite). So, it works! Well done, Bug! As such, e again.