COSC 301: Operating Systems
Lab 7: Semaphores

1. Dining savages problem (from Downey, 2008)

    > A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary1. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.

Any number of savage threads run the following code:

```
# Unsynchronized (wrong!) code
while True:
    getServingFromPot()
    eat()
```

And one cook thread runs this code:

```
# Unsynchronized (wrong!) code
while True:
    putServingsInPot(M)
```

The synchronization constraints are:

- Savages cannot invoke getServingFromPot if the pot is empty.
- The cook can invoke putServingsInPot only if the pot is empty.

Add (pseudo-)code using semaphores for the savages and the cook that satisfies the synchronization constraints:

```
# Pseudocode here










#
```

2. Consider the following type declaration:

```
    typedef struct __zem_t {
        int value;
        pthread_cond_t cond;
        pthread_mutex_t lock;
    } zem_t;
```

Complete the following code to implement a semaphore using a condition variable and mutex:

```
void zem_init(zem_t *z, int value) {









}

void zem_wait(zem_t *z) {









}

void zem_post(zem_t *z) {
}
```

3. Consider the following `main` function (pseudocode) and functions to be used in a producer-consumer solution with 1 producer thread and any number of consumer threads:

```
// horrible, horrible global variables
int max, loops, consumers;
int *buffer;

int main(int argc, char **argv) {
    max = atoi(argv[1]);
    loops = atoi(argv[2]);
    consumers = atoi(argv[3]);
    buffer = (int*) malloc(max * sizeof(int));
    pthread_t pid, cid[CMAX];

    pthread_create(&pid, NULL, producer, NULL);
    for (int i = 0; i < consumers; i++) {
        pthread_create(&cid[i], NULL, consume, NULL);
    }

    pthread_join(pid, NULL);
    for (int i = 0; i < consumers; i++) {
        pthread_join(cid[i], NULL);
    }
    return 0;
}

void do_fill(int value) {
    buffer[fillptr] = value;
    fillptr = (fillptr + 1) % max;
    numfull++;
}

int do_get(void) {
    int tmp = buffer[useptr];
    useptr = (useptr + 1) % max;
    numfull--;
    return tmp;
}
```

Here are four possible solutions for `produce` and `consume` functions; some are incorrect. For each solution, describe how it works and what (if anything) is broken. You can assume that all mutexes and condition variables are initialized correctly.

Solution version 1:

```
# One condition variable, one mutex

void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
        while (numfull == max) {
            cond_wait(&cond, &m);
        }
        do_fill(i);
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}

void *consumer(void *arg) {
    while (1) {
        mutex_lock(&m);
        while (numfull == 0) {
            cond_wait(&cond, &m);
        }
        int tmp = do_get();
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

Solution version 2:

```
# Two condition variables, one mutex

void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
        if (numfull == max) {
            cond_wait(&empty, &m);
        }
        do_fill(i);
        cond_signal(&fill);
        mutex_unlock(&m);
    }
}

void *consumer(void *arg) {
    while (1) {
        mutex_lock(&m);
        if (numfull == 0) {
            cond_wait(&fill, &m);
        }
        int tmp = do_get();
        cond_signal(&empty);
        mutex_unlock(&m);
    }
}
```

Solution version 3:

```
# two condition variables, one mutex

void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
        while (numfull == max) {
            cond_wait(&empty, &m);
        }
        do_fill(i);
        cond_signal(&fill);
        mutex_unlock(&m);
    }
}

void *consumer(void *arg) {
    while (1) {
        mutex_lock(&m);
        while (numfull == 0) {
            cond_wait(&fill, &m);
        }
        int tmp = do_get();
        cond_signal(&empty);
        mutex_unlock(&m);
    }
}
```

Solution version 4:

```
# two condition variables, one mutex

void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
        while (numfull == max) {
            cond_wait(&empty, &m);
        }
        mutex_unlock(&m);
        do_fill(i);
        mutex_lock(&m);
        cond_signal(&fill);
        mutex_unlock(&m);
    }
}

void *consumer(void *arg) {
    while (1) {
        mutex_lock(&m);
        while (numfull == 0) {
            cond_wait(&fill, &m);
        }
        mutex_unlock(&m);
        int tmp = do_get();
        mutex_lock(&m);
        cond_signal(&empty);
        mutex_unlock(&m);
    }
}
```