ECE254 Midterm S2017

1.

A) Assuming to use 3 processes. I'm printing in the processes themselves instead of returning the result index since Linux return codes have a range of [0, 255], so if the array size is larger than that the return value would overflow (I don't think we need to know this though. I didn't, this was courtesy of Evan).

```
int main(int argc, char** argv) {
	int childResult;
	int child2Result;
	int parentResult;

	pid_t pid = fork();

	if (pid < 0) return 1;

	if (pid == 0) {
		// Child process
		for (int j = 0; j < array_length/3; ++j) {
			if (array[i] == search_value) {
				printf("Found at %d\n", i);
				return 1;
			}
		}
		return -1;
	} else {
		// Parent process
		pid_t pid2 = fork();
		if (pid2 == -1) return 1;

		if (pid2 == 0) {
			// Child 2
			for (int j = array_length/3; j < 2*array_length/3; ++j) {
				if (array[i] == search_value) {
					printf("Found at %d\n", i);
					return 1;
				}
			}
			return -1;

		} else {
			// Parent
			parentResult = -1;
			for (int j = 2*array_length/3; j < array_length; ++j) {
				if (array[i] == search_value) {
					printf("Found at %d\n", i);
					break;
				}
			}
			wait(&childResult);
			wait(&child2Result);
		}
	}
	if (parentResult == -1 && childResult == -1 && child2Result == -1) {
		// Not found, but they didn't say to print anything for not found :P
	}
```

```
        return 0;
}
```

B) Allowing some signals to be caught allows the process to clean up before termination or whatever action the signal signals. Some signals, such as SIGKILL and SIGSTOP, cannot be caught. These immediately terminate or stop the process, so require no action from the process itself. This allows us to terminate unresponsive or badly behaved processes.

2.

A)

```
// No additional initializations or cleanups
void* oxygen(void* ignore) {
        pthread_mutex_lock(&bond_mutex);
        oxygen++
        if (hydrogen >= 2)
                sem_post(&hydrogen_queue);
                sem_post(&hydrogen_queue);
                hydrogen -= 2
                sem_post(&oxygen_queue);
                oxygen--
        else
                pthread_mutex_unlock(&bond_mutex);

        sem_wait(&oxygen_queue);
        bond();

        barrier_enter();
        pthread_mutex_unlock(&bond_mutex);
        barrier_exit();
}

void* hydrogen(void* ignore) {
pthread_mutex_lock(&bond_mutex);
hydrogen++
if (hydrogen >= 2 && oxygen >= 1)
        sem_post(&hydrogen_queue);
        sem_post(&hydrogen_queue);
        hydrogen -= 2
        sem_post(&oxygen_queue);
        oxygen--
else
        pthread_mutex_unlock(&bond_mutex);

        sem_wait(&hydrogen_queue);
        bond();

        barrier_enter();
        barrier_exit();
}
```

<u>Oxygen</u>

```
wait(bond_mutex)
oxygen++
if (hydrogen >= 2)
        post(hydrogen_queue)
        post(hydrogen_queue)
        hydrogen -= 2
        post(oxygen_queue)
        oxygen--
else
        post(bond_mutex)

wait(oxygen_queue)
bond()
barrier_enter()
post(bond_mutex)
barrier_exit()
```

<u>Hydrogen</u>

```
wait(bond_mutex)
hydrogen++
if (hydrogen >= 2 && oxygen >= 1)
        post(hydrogen_queue)
        post(hydrogen_queue)
        hydrogen -= 2
        post(oxygen_queue)
        oxygen--
else
        post(bond_mutex)

wait(hydrogen_queue)
bond()
barrier_enter()
barrier_exit()
```

B) This technique does not work, since the lock++ statement is not atomic. An example is as follows.

Thread A gets to the lock++ statement.
Thread A reads 0.
Thread A increments lock (value now 1).

Thread switch: Thread B is chosen to run.
Thread B gets to the lock++ statement.
Thread B reads 0.
Thread B increments lock (value now 1).
Thread B writes 1 to lock.
Thread B evaluates the if statement and enters the critical section.
Thread switch: Thread A is chosen to run.
Thread A writes 1 to lock.
Thread A evaluates the if statement and enters the critical section.
Both threads A and B are now in the critical section :(

C) [Not covered on midterm]
This does not solve the problem of deadlock. The same problem can occur. Suppose each philosopher attempts to pick up the chopstick adjacent to their seat (which is free). Then we are back at the same deadlock situation, where each philosopher has one chopstick.

3. **A)** [Not covered on midterm]

```
void foo() {
        /* Get Ready */
        int locked_both = 0;
        while( locked_both == 0 ) {
                int locked1 = pthread_mutex_trylock( &m1 );
                int locked2 = pthread_mutex_trylock( &m2 );
                if (locked1 != 0 && locked2 == 0 {
                        pthread_mutex_unlock( &m2 );
                } else if (locked1 == 0 && locked2 != 0 ) {
                        pthread_mutex_unlock( &m1 );
                } else if (locked1 != 0 && locked2 != 0 ) {
                        /* Do nothing */
                } else {
                        locked_both = 1;
                }
        }
        /* Critical section */

        pthread_mutex_unlock( &m1 );
        pthread_mutex_unlock( &m2 );

        /* Clean Up */
}
```

bar() is exactly the same.

B) [Not covered on midterm]