Homework Assignment 3

William Jagels

December 1, 2016

I. Q&A

I.1.

1. `abort()` is supposed to halt the program in the event of an unrecoverable error in the program. By default, `SIGABRT` will halt and core dump.

2. We want to make sure that the signal handler is run for `SIGABRT`. The first one lets the user's handler go to work, and hopefully that should call exit on its own. If that returns, then we have to force the default signal handler, and then use that to fully kill the program.

3. The self-kill should always execute the default `SIGABRT` handler which is defined to exit the program. If line 27 is executed, something is terribly wrong.

4. We don't want any other code being executed because it could rely on the integrity of the thing that just caused `abort()` to be called. Allowing other signals to be caught may be dangerous, so we shouldn't allow that to happen.

I.2.

1. This code spawns one child that spawns another child, and then kills iteslf, leaving the grandchild as an orphan. The orphan will continue with init as its parent as if it was started by init. The parent process can also do whatever it wants, maybe even communicate with the orphan.

2. This code is useful if the programmer wants to start a daemon process. When the shell used to call the initial process exits, the grandchild process won't exit. The grandchild can run some sort of a server or service for the rest of the machine without the risk of being killed by a shell exiting.

I.3.

1. Code without condition variable.

```
1  struct msg *workq;
2  pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;
3  pthread_mutex_t olock = PTHREAD_MUTEX_INITIALIZER;
4
5  void process_message() {
6    struct msg *mp;
7
8    for (;;) {
9      pthread_mutex_lock(&qlock);
10     while (workq == NULL) {
11       pthread_mutex_unlock(&qlock);
12       sleep(1);
13       pthread_mutex_lock(&qlock);
14     }
15     mp = workq;
```

```
16    }
17  }
18
19  void enqueue_msg(struct msg *mp) {
20    pthread_mutex_lock(&qlock);
21    mp->m_next = workq;
22    workq = mp;
23    pthread_mutex_unlock(&qlock);
24  }
```

2. The condition variable is better in this situation because it prevents needless unlocking and locking of the mutex. When there's nothing in workq, there will be a lot of useless work done on the cpu.

3. When `pthread_cond_wait()` is called, the mutex is unlocked and then the consumer thread blocks on the condition variable, allowing the producer thread to do work.

4. Sending the signal before the unlock is fine, the wait also waits for the lock to be unlocked by the other producer thread.

I.4.

Using `pthread_mutex_trylock`, in an `if` block will make this work. Since trylock will only succeed for one thread, there will only ever be one thread that enters that branch, unless the mutex is unlocked.