I.1.

1. Abort() function unblocks and raises SIGABRT signal for the calling process and corresponding handler will be called. If the signal is ignored or caught by handler, the abort() function will still terminate the process as it restores the default disposition of SIGABRT and raises the signal for second time.

   The default disposition of SIGABRT is
   1. Terminate the process abnormally,
   2. Return the error exit code to the parent process,
   3. All open streams are flushed.
   4. Core dump.
2. The first implementation of kill() will raise the SIGABRT and it can be caught by any other handler other than default disposition, registered by the process. To make sure that the program terminates even after the signal caught by handlers, the disposition of SIGABRT is restored to default (line 22 and line 23) , the kill() is called again to terminate the process.

3. Abort() function is intended to terminate the process abnormally without any storage cleanup, and dumping the core. Exit() function does the same termination as abort() but exit() cleans up the storage and does not dump core. So if exit(1) (line 27) is called, it will change the actual implementation of abort(). So it should not be called.

4. No, we don't need to unblock other signals. Unblocking after first kill() do not guarantee termination of the process as intended. Unblocking after second kill() does not make any sense, as it won't be executed.

I.2.

1.
   a. The grandparent process forks a child process and continues running.
   b. The child process forks a grandchild process and exits.
   c. The grandchild process keeps running as orphan process and is adopted by init process.
2. This kind of orphan process adopted by init process are technically called as daemons in UNIX-like systems. These kind of implementation will be useful in running long background jobs with no interference from the actual parent process. Even if the actual parent process is terminated, the orphan process will be running.

I.3.

1.

```
struct msg {
        struct msg *m_next;
        /* ... more stuff here ... */
```

```
}

struct msg *workq;
//pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qclock = PTHREAD_MUTEX_INITIALIZER;

volatile int available = 0 // adding global variable

void process_msg(void)
{
        struct msg *mp;
        for(;;){
                pthread_mutex_lock(&qlock);
                while( available > 0) // Checking the global var if message is enqueued.
                        available --;
                        //pthread_cond_wait(&qready, &qclock);
                mp = workq;
                workq = mp->m_next;
                pthread_mutext_unlock(&qlock);
                … … /*process the message up */
        }
}

void enqueue_msg(struct msg *mp)
{
        pthread_mutex_lock(&qlock);
        mp->m_next = workq;
        workq = mp;
        available ++: // Incrementing global variable when it is ready
        pthread_mutex_unlock(&qlock);
        //pthread_cond_signal(&qready);
}
```

2. Without conditional variable, this can be implemented. But this is very costly and resource killing one. The consumer will be trying to lock the resource repeatedly until the workq is available. So calling mutex_lock repeatedly is resource-intensive one. So in this implementation, Conditional variable is favourable solution.
3. Whenever a thread calls pthread_cond_wait(),
    a. The function assumes that consumer mutex is locked, and it releases the lock.
    b. Then it blocks the consumer thread until waken up by the signal.

   If pthread_cond_signal() is called by another thread,
    a. The consumer thread acquires the mutex lock access and it is locked to the consumer thread before returning from pthread_cond_wait()

4. Since we are checking the workq msg in while loop, the order of holding the lock and signaling does not matter, as the consumer thread wakes, checks for workq msg and sleeps. But if the code does not support these kind of race, it is advisable to signal while holding the lock. So it is better to switch the order of line 30 and line 31.

I.4.

While initializing the program, we can store the thread_id of the main function using pthread_self() and store it as global constant variable. Whenever required function is called, we can check if the caller thread id with the global constant variable using pthread_equal, and make it execute only if it matches else skip it.