Systems Programming HW-3

Saiprakash Nalubolu

B01037579

I.1

(1) The abort() function is a standard library function in C, defined in <stdlib.h>. Its purpose is to abort the current process immediately. It raises the SIGABRT signal, which by default causes the process to terminate abnormally. The default disposition of SIGABRT is to cause the program to terminate and produce a core dump.

(2) The implementation calls kill(getpid(), SIGABRT) twice to ensure the process terminates even if the SIGABRT signal handling is somehow modified or ignored after the first call. The process flow ensures:

a) Any ignoring of SIGABRT is reset to default handling. SIGABRT is explicitly unblocked and then sent.
b) After flushing the I/O buffers and trying the signal again, it aims to leave no chance for the process to continue if the first signal was caught and modified by a custom handler that might prevent termination.
c) Replacing the calls from line 21 to 27 with a simple exit() would change the function's behavior. The exit() function terminates the process normally, performing cleanup operations such as flushing buffers and running functions registered with at exit(). In contrast, abort() is designed to terminate the process abruptly, by passing cleanup operations (except for flushing stdio buffers) and emphasizing immediate termination for error conditions. Using exit() would not fulfill the intended abrupt termination characteristic of abort().

(3) The line 27 “exit(1)” is considered unreachable under normal circumstances because the process should have already been terminated by the SIGABRT signal sent twice before reaching this line. It acts as a safety net, ensuring the process exits with a non-zero status (indicating error) if, for some unlikely reason, both attempts to send SIGABRT are ineffectual.

(4) The implementation unblocks SIGABRT to ensure it can be received, but it blocks all other signals. This is to prevent any other signal from interrupting the sequence of operations leading to the termination of the process. Typically, there would be no need to unblock other signals within this code block because abort() is meant to terminate the process immediately without further interruption. Unblocking other signals either inside or outside this code is unnecessary unless there's a specific requirement to handle other signals differently before the process terminates. However, such a scenario would not be a normal/general usage of abort().

I.2

(1) The code snippet outlines a process of forking processes and handling the execution flow depending on whether the operation is executed in the parent, child, or grandchild process:

1. **First Fork (cpid_1 = fork()):** This creates a child process. If the fork fails, it prints an error message and exits with a status of 1.
2. **Child Process (when cpid_1 == 0):**

    - **Second Fork (cpid_2 = fork()):** The child process itself forks another process, creating a grandchild.
    - If the second fork fails, it prints an error and exits with a status of 1.
    - **Grandchild Process (when cpid_2 == 0):** This is where the "real work" is done. After completing the work, it exits with a status of 0.
    - **Child Process After Forking the Grandchild:** The child process immediately exits with a status of 0 after spawning the grandchild. This makes the grandchild an orphan, meaning its parent process is no longer running and it is now adopted by the init process.

3. **Parent Process (original process):** The parent process enters a loop where it checks if the child has terminated using waitpid() with the WNOHANG option. If waitpid() returns -1, an error occurred, which is reported and the parent process exits. If the child has terminated, it can potentially break out of the loop (though not shown in the snippet) and continue doing other tasks as suggested by the placeholder comment.

    Outcome of the code:

- The grandchild becomes an orphan and can execute independently of its parent.
- The child exits immediately after creating the grandchild.
- The parent continues to run other tasks while periodically checking on the status of its direct child.

(2) This code pattern is particularly useful in scenarios where:

1. **Background Jobs in Unix/Linux Systems:** When a service or application needs to launch tasks that should run independently of the main application process, allowing the main application to terminate without killing these tasks.
2. **Daemon Processes:** Creating daemons in Unix-like systems often involves forking twice. The first child process exits to ensure that the actual working process (the grandchild) is not a session leader, thereby preventing it from accidentally acquiring a controlling terminal.
3. **Resource-Intensive Tasks:** When an application needs to offload heavy computations or I/O operations that should not interfere with the main application process, ensuring smooth operation without waiting for the task completion.

4. **Handling Multiple Tasks Simultaneously:** The parent can execute other operations or spawn more child processes for other tasks, managing multiple operations in a non-blocking fashion.

This method leverages the Unix process management effectively to isolate tasks and manage process hierarchies, ensuring that critical processes have the minimal risk of being inadvertently terminated.

I.3

(1) If we were restricted to only use mutexes for thread synchronization and could not use conditional variables, the code would need to handle polling or busy waiting for messages. Here's how you might rewrite process_msg() and enqueue_msg() using only a mutex:

```
void process_msg(void) {
   struct msg *mp;

   for (;;) {
      pthread_mutex_lock(&qlock);
      while (workq == NULL) {
         pthread_mutex_unlock(&qlock);  // Unlock to allow producer to add messages.
         sched_yield();                 // Yield processor to reduce busy waiting.
         pthread_mutex_lock(&qlock);    // Lock again to check condition.
      }
      mp = workq;
      workq = mp->m_next;
      pthread_mutex_unlock(&qlock);
      // process the message mp
   }
}

void enqueue_msg(struct msg *mp) {
   pthread_mutex_lock(&qlock);
```

```
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
}
```

(2) Using a Conditional variable has the following advantages,

- **Efficiency:** Conditional variables avoid busy waiting. Busy waiting consumes CPU time needlessly while waiting for a condition to change. In contrast, with conditional variables, threads are put to sleep and do not consume CPU resources while they wait for the condition to become true.
- **Synchronization:** Conditional variables provide a means to allow threads to wait for specific conditions to occur, waking up only when the condition is met or a signal is received. This is more synchronized as opposed to constantly acquiring and releasing a lock to check a condition.
- **Resource Utilization:** Conditional variables can improve overall resource utilization by ensuring that threads only execute when necessary conditions are met, thus avoiding unnecessary lock contention.

(3) Consumer Thread and Mutex qlock When pthread_cond_wait() is Called:

Consumer Thread: The consumer thread, which is executing process_msg(), blocks and waits on the condition variable qready. This means the thread stops executing and will not proceed until it is signaled.

Mutex qlock: Simultaneously, pthread_cond_wait() automatically releases the mutex qlock that the thread had locked. This is crucial as it allows other threads (producers) to acquire the mutex to modify the shared resource (workq). Once the condition is signaled and the thread is awakened, it automatically re-acquires the mutex qlock before it can proceed with execution.

Behavior When pthread_cond_signal() is Called;

Consumer Thread: If the consumer thread is waiting on the condition variable, pthread_cond_signal() will wake up at least one waiting thread. The awakened thread will then compete to acquire the mutex associated with the condition variable in order to continue execution.

Mutex qlock: The mutex qlock remains held by the signaling thread (producer) during the call to pthread_cond_signal(). The mutex is not affected by the signaling operation and must be manually released by the producer when ready.

(4) Yes, we can switch the order of signaling and unlocking in the enqueue_msg() function. However, the conventional practice is to unlock the mutex after signaling for several reasons:

If the mutex is unlocked before signaling, the waiting thread that is awakened by pthread_cond_signal() may immediately proceed to try to lock the mutex. If the mutex is not yet unlocked because the signal was sent first, the awakened thread must block again waiting for the mutex, leading to an unnecessary context switch.

Unlocking first ensures that when the waiting thread wakes up, it can immediately acquire the mutex and proceed, potentially reducing latency and improving performance. Switching the order might not always cause issues, but it can lead to less efficient handling of thread wake-up and continuation, especially under high contention scenarios.