1. [10 points] Consider the following code:

```c
#include <signal.h> // for kill()
#include <stdlib.h> // for exit() and atoi()
int main(int argc, char *argv[])
{
    int aig;  
    pid_t pid;

    aig = atoi(argv[1]);
    pid = atoi(argv[2]);
    if ((aig < 1) || (aig > 31))
        exit(1);
    kill(pid, aig);
}
```

The `kill()` is a system call. The `atoi()` call converts a string to an integer. Suppose this code is compiled and run, and that the executing process ID is 1000. It is run with the values 9 and 2000 for `argv[1]` and `argv[2]`, respectively. Answer the following questions with convincing arguments:

- [3 points] From what processes (identified by `pid`) is the system call for `kill()` raised?
- [3 points] To what process is the signal sent?
- [4 points] The system call interrupt is handled by the operating system. What data is received by the interrupt handler?

**Answer:** Because program is run with values 9 and 2000, `argv[1]` and `argv[2]` are 9 and 2000, respectively. Therefore, `aig` and `pid` are 9 and 11. Moreover, we know the executing process ID is 1000.

- The system call `kill(pid, aig)` is called by the executing process, and the answer to this question is 1000.
- The system call `kill(pid, aig)` has its first argument a process ID `pid` to which the signal `aig` is sent. Hence, the receiver of this signal 9 (i.e., SIGKILL) is the process with ID 2000.
- A system call requires at least the call number and a list of arguments. In this case, the interrupt handler will receive the system call number that corresponds to `kill`, and the arguments are at least the receiver’s ID (i.e., 2000) and the signal to be sent (i.e., 9). The process ID of the caller may be retrieved from the PCB when the caller is suspended due to this interrupt (actually a trap).

2. [15 points] Consider the following code:
1. int intA;
2. int main(void)
3. {
4. int intB;
5. pid_t cpid;

6. intA = 0; intB = 2;
7. cpid = fork();
8. intA++; intB++;
9. if (cpid == 0) {
10. intA++; intB++;
11. printf("intA <%d>, intB <%d>\n", intA, intB);
12. }
13. else {
14. wait();
15. intA++; intB++;
16. printf("intA <%d>, intB <%d>\n", intA, intB);
17. }
18. }

Suppose that all the calls execute normally.

- [7 points] Is the output from this program deterministic? More precisely, does it print out the same thing every time?
- [8 points] If the output is deterministic, what is printed out? (Why?) If not, explain as clear as possible what are the possible output?

**Answer:** Let us look at intA first. It is a global variable and initialized to 0 in the main() (line 6) before a child process is created. After a child process is created (line 7), the parent and the child have their own identical but separate address spaces. This means there are two copies of intAs, one in the parent’s address space and the other in the child’s. Then, both the parent and child see intA to be 0 in their own address spaces and add 1 to it. At this moment, intA is 1. The child adds 1 to intA making its value 2 (line 10) and prints out the value of intA. Meanwhile, the parent may or may not be waiting for the child’s completion. Once the child terminates, the parent adds 1 to its own copy of intA, making its value 2 (line 15). Therefore, for intA its value is always 2.

Let us turn to intB which is set to 2 (line 6). A child process is created, and the parent and the child both have intB in their corresponding address spaces, and the value of intB is 2. Finally, the parent (line 15) and the child (line 10) both add 1 to intB, and both copies of intB are the same (i.e., 4).

Therefore, the output of this program is **deterministic** and the out is

```
intA <2>, intB <4>
intA <2>, intB <4>
```

Note that there are two output lines, one from the parent and one from the child.

3. [15 points] As discussed in class that a thread should not delete its own states upon exit. More precisely, when a thread calls THREAD_EXIT() to terminate itself, THREAD_EXIT() should not delete this thread’s states. Answer the following questions:

- [10 points] Why a thread should not delete its own states. Answer this question as accurate as possible and use examples to illustrate your point.
- [5 points] Then, how would you solve this problem?

**Answer:**
A thread should not remove its own states upon exit for at least two major reasons.

- If the thread frees its state, it does not have a stack to finish its code in `thread_exit()`.
- When an interrupt occurs just after the running thread’s stack has been deallocated, if the context switch code tries to save the current’s state, it will be written to de-allocated memory, which may have been allocated to other thread for some other data structure.

We can use another thread to free a thread’s state. In other word, this thread when it is created is wrapped by a very simple thread (i.e., wrapper or stub).

Refer to slide numbers 22-24 in 04-Concurrency.pdf for the details.

4. [15 points] Suppose we have the following problem using `setjmp()` and `longjmp()`.

```c
jmp_buf Buf1, Buf2;

int main(void) { 
    // other code
    if (setjmp(Buf1)==0) C();
    else
        longjmp(Buf1,1);

    printf("I am back\n");
}
```

Study the above code and answer the following questions:

- [5 points] There are two calls to `longjmp()`. Are they correct? Identify the incorrect one(s).
- [10 points] If you identify at one incorrect `longjmp()`, then use diagrams with elaboration to explain why a particular call to `longjmp()` is incorrect.

**Answer:** The following diagram shows the situation of the calls from `main()` to function `C()`. Note that the data section has two global variables `Buf1` and `Buf2`.

(a) `main()` calls `A()` in which the `setjmp()` sets `Buf1` to point the return point at the if statement.
(b) `A()` calls `B()`, which, in turn, calls `C()`.
(c) Now in `C()`, the `setjmp()` call sets `Buf2` to point to the return point at the if statement. Note the location of the stack top pointer.

The `longjmp()` in `C()` brings the control back to the location recorded in `Buf1`, which is in `A()`. This long jump is a valid one because once the control is brought back to `A()` the stack of `A()` is still intact. **Therefore, the first `longjmp()` is a correct one.**

Now let us turn to the long jump in `A()`. Because the `longjmp()` in `C()` returns 1, the `else` part of the if in `A()` is executed. Also note that from `A()`’s point of view, the stack top pointer points to a location next to `A()`’s stack frame. Certainly, `A()` does not know where `C()`’s stack frame is. As a result, the configuration at this point is shown below.
Now, A()’s longjmp() sends the control back to C() is problematic. Therefore, when the control goes back to C(), C() does not have its original stack to run its code. Well, if C() does not use any local variables, this may be fine. But, in general, the second longjmp() is problematic.

5. [15 points] Suppose a system in which there are two types of processes, type A processes and type B processes. All processes of type A execute the same code, and all processes of type B execute the same code. The code for each process type is shown below.

<table>
<thead>
<tr>
<th>A Processes</th>
<th>B Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(X)</td>
<td>P(Y)</td>
</tr>
<tr>
<td>V(Y)</td>
<td>P(Y)</td>
</tr>
<tr>
<td>V(X)</td>
<td>V(Y)</td>
</tr>
</tbody>
</table>

Here, X and Y are general semaphores. X is initialized to 2, and Y is initialized to 0. Suppose three processes of type A and two processes of type B are brought into execution simultaneously. Answer the following two questions:

- [8 points] Is it possible for processes to finish in the order of AABAB? If so, show an execution sequence that results in this order. If not, explain why as accurate as possible.
- [7 points] Is it possible for processes to finish in the order AABBA. If so, show an execution sequence that results in this order. If not, explain why as accurate as possible.

**Answer:** The first execution order AABAB is possible, but the second one AABBA is impossible.

- The following is an execution sequence showing that AABAB is possible:

<table>
<thead>
<tr>
<th>A₁</th>
<th>A₂</th>
<th>B₁</th>
<th>A₃</th>
<th>B₂</th>
<th>Semaphore X</th>
<th>Semaphore Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P(X)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V(Y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P(X)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>V(Y)</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P(Y)</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P(Y)</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V(X)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>V(Y)</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>P(X)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>V(Y)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P(Y)</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P(Y)</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>V(X)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>V(Y)</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
This execution sequence shows clearly that processes $A_1$, $A_2$ and $A_3$ in group $A$, and processes
$B_1$ and $B_2$ in group $B$ can indeed produce the order $AABAB$.

- The sequence $AABBA$ is impossible. From the above execution sequence, after $AA$ semaphores $X$ and $Y$ have counters 0 and 2. If $B$ follows, $X$ and $Y$ will be 1 and 1. Because $Y$'s counter is 1, the next $B$ can only pass its first $P(Y)$ and blocks by the second $P(Y)$. Then, the second $B$ will be blocked by its first $P(Y)$, and, as a result, the order of $AABBA$ is impossible.

6. [15 points] In the implementation of lock::acquire() and lock::release() on multiprocessor system, we have the following code:

```c++
1. Lock::acquire()                 Lock::release()
2. {
3.   spinLock.acquire();
4.   if (value == BUSY) {
5.     waiting.add(myTCB);
6.     scheduler.suspend(&spinLock);
7.   }
8. } else {
9.   value = BUSY;
10.  }
11. spinLock.release();
12. }
```

A spinlock is implemented as follows:

```c++
1. Spinlock::acquire()
2. {
3.   while (testAndSet(&lockValue) == BUSY) lockValue = FREE;
4.   memorybarrier();
5. }
```

As you can see there is no disableinterrupts() and no enableinterrupts(). However, functions scheduler.suspend() and scheduler.makeReady() do disable interrupts at the very beginning and enable interrupts upon exit. Explain why as accurate as possible.

**Answer:** We could think the other way around: what if we add disableinterrupts() and enableinterrupts() to the lock implementation? In the way, Lock::acquire() would become the following:

```c++
1. Lock::acquire()
2. {
3.   disableinterrupts();
4.   spinLock.acquire();
5.   if (value == BUSY) {
6.     waiting.add(myTCB);
7.     scheduler.suspend(&spinLock);
8.   }
9. } else {
10.  value = BUSY;
11.  }
12. enableinterrupts();
13. spinLock.release();
14. }```
Note that scheduling occurs only if a running process enters its terminated state, or waits for event (e.g., I/O completion), or voluntarily releases the CPU (e.g., executing the yield command), or an interrupt occurs (e.g., time quantum expires). Now if in Lock.acquire() interrupts are disabled and the spinLock starts to spin, then unless the value of the spinlock is changed to FREE the processor will not be able to do anything. This could become very inefficient. Of course, if this code runs on a uniprocessor system, it is a disaster. (Why?)

7. [15 points] The following is an implementation of condition variable wait() and signal(). This implementation uses a queue to chain waiting threads’ TCBs together. This queue has an Append() function to add a TCB to the queue and an Empty() function to test whether the queue is empty. There is a semaphore with initial value 0 to block those threads that have to wait.

```cpp
wait(lock) signal()
{ { queue.Append(myTCB); if (!queue.Empty()) {
    lock.release(); semaphore.V();
    semaphore.P();
    lock.acquire();
} }
```

This implementation is incorrect as discussed in class. Study this implementation and enumerate as many problems as possible using execution sequences as needed, and for each problem state its nature and possible violations with respect to condition variable requirements.

**Answer:** The problem of this “solution” is that a signal() to a condition variable in which no one is waiting could have some unexpected effect. In the following execution sequence Thread 1 calls wait() on a condition variable with its TCB queued and lock released. Because the lock is released, Thread 2 acquires it, finds out the queue being non-empty (i.e., Thread 1 waiting), and executes V() in order to release Thread 1. Meanwhile, Thread 1 is not waiting on the semaphore, and, as a result, the count of his semaphore is increased by 1. If a context switch occurs allowing Thread 3 to run, Thread 3 executes a wait() and queues its TCB, and calls P() to wait. Because of a previous V() on this semaphore (line 4), Thread 3 does not have to wait at all. Hence, Thread 2 released a “future” waiting thread. This violates the meaning of condition wait and condition signal.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 wait()</td>
<td></td>
<td></td>
<td>semaphore = 0</td>
</tr>
<tr>
<td>2 myTCB queued</td>
<td></td>
<td></td>
<td>lock is open</td>
</tr>
<tr>
<td>3 lock released</td>
<td></td>
<td></td>
<td>semaphore V()</td>
</tr>
<tr>
<td>4</td>
<td>semaphore</td>
<td></td>
<td>semaphore = 1</td>
</tr>
<tr>
<td>5</td>
<td>wait()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 myTCB queued</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>semaphore</td>
<td>P()</td>
<td>semaphore = 0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>Thread 3 released</td>
</tr>
<tr>
<td>9 semaphore</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

However, the big problem is: Because Thread 1 has not reached its waiting state, the V() should have no effect. Of course, this is an incorrect implementation. Refer to slide numbers 47–48 in 05-Synchronization.pdf.