Part 1. More fun with fork and exec*

What is the output generated by this program? Please assume that each executed print statement completes, e.g., assume that each print is followed by an "fflush(stdout)". Be sure to consider whether the output is uniquely defined. You should presume that no errors occur. [30 points]

```c
main(int argc, char * argv[])
{
    printf("0");
    fork();
    printf("1");
    execvp(argv[0], argv);
    printf("2");
}
```

Optional: explain. I will consider your explanation for partial credit if you need it.
Part 2. Ping Pong

These questions deal with the “ping pong” example discussed in class. PingPong is implemented in Java roughly like the code on the right:

```java
synchronized void PingPong()
{
    while(true) {
        computeSomething();
        notify();
        wait();
    }
}
```

(a) Suppose that multiple threads call this PingPong method on the same object concurrently. Describe the resulting behavior. I am looking for 1-3 sentences summarizing any constraints on the execution ordering. [15 points]

(b) Does it matter if we change the order of the statements in the loop? What if `computeSomething()` is between the `notify` and the `wait`, or after the `wait`? What if we change the order of the `notify` and `wait`? [15 points]
Part 3. Servers and threads and events

Answer each of these short questions using a word, or a phrase, or a sentence, or maybe two. [10 points each]

(a) It has been said that an event-driven design pattern is better than using multiple threads to structure a server. I have argued in class that it is best to use both models together. Why is it important for a server to have multiple threads, even in a system with full support for event-driven programming (e.g., with non-blocking system calls)?

(b) In a ThreadPool implementation, how does a sleeping worker thread "know" that it has been selected to handle the next request or event? What wakes it up?

(c) The main thread of an Android application runs an event-driven pattern. If the app has a task to perform that is long-running or that must block, the main thread can start an AsyncTask to do the work. In this case, how does the AsyncTask report its progress back to the main thread?

(d) An atomic instruction performs an indivisible read-modify-write on memory: in particular a thread can use an atomic instruction to set a lock variable safely, indicating that the lock is busy. In this case, how does a thread "know" if it lost the race and the lock is already busy? What should the thread do if the lock is busy?
Part 4. dsh

These questions pertain to your shell project code (dsh). If you don’t remember the details of what your group did then make something up that sounds good. Please confine your answers to the space provided. [12 points each]

(a) What would happen if you type the command “dsh” to your dsh, i.e., to “run dsh under itself”? Will it work? What could go wrong?

(b) Briefly summarize how your dsh implemented input/output redirection (<>). What system calls did you call from the parent for this purpose? What system calls did you call from the child for this purpose?

(c) For pipeline jobs like `cat | cat | cat`, what would happen if a child writes into a pipe before the next downstream child (the process that is supposed to read from that pipe) has started? Can this scenario ever occur?

(d) For pipeline jobs like `cat | cat | cat`, what would happen if a child reads from a pipe before the next upstream child (the process that is supposed to write to that pipe) has started? Can this scenario ever occur?

(e) For pipeline jobs like `cat | cat | cat`, how does a middle child’s setup differ from the processes at the ends of the pipeline? How does a process “know” that it is the middle child?
Part 5. Maps

These questions pertain to a “classic” 32-bit virtual memory system, with linear page tables and 4KB pages. Answer each question with an ordered sequence of events. Feel free to draw on the back page: otherwise, I am looking for as much significant detail as you can fit in the space provided. [10 points each]

(a) Suppose that a running thread issues a **load** instruction on the following 32-bit virtual address: 0x00002014. Suppose that the address misses in the TLB. How does the hardware locate the page table entry for the page? Please show your math.

(b) Suppose further that the page table entry is “empty”: it does not contain a valid translation. The hardware cannot complete the requested virtual memory reference. What does it do?

(c) How does the operating system determine if the reference is legal, i.e., how does it determine whether or not the reference results from an error in the program?

(d) If the reference is legal, how does the operating system find the data for the page? Is it possible that the page already resides in memory? (Why or why not?) You may presume that the requested virtual page has never been modified.