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");
}
```

011….
An unbounded sequence of 0s and 1s with twice as many 1s as 0s (in the limit). After the first 01 the exact order is undefined.

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

Parent prints 0.
Fork creates child.
Both parent and child print 1.
Both parent and child exec the same program again (argv[0]). [5 points]
Exec* does not return if there is no error, so 2 is never printed. [5 points]

Note: for the remaining answers that have key words or phrases in bold, those words were generally sufficient for full credit. (I also used bold for identifiers, but those answers required more explanation.)
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]

The threads execute `computeSomething` serially (one after the other) in some undefined order, forever.

Optional explanation: Since the method is `synchronized`, a monitor is held and so at most one thread can execute in the method at a time. Each thread enters the method and holds the monitor until it calls `wait`. Each `wait` blocks the caller and releases the monitor, allowing another thread to execute in the method. Before calling `wait`, a thread calls `notify` to wake up one waiting thread, if there is one. Upon resuming in `wait` after a wakeup (resulting from a `notify` by some other thread), a thread reacquires the monitor, returns from `wait`, and executes another iteration of the loop before waiting again.

(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]

It doesn’t matter where the `computeSomething` call is: it might affect the order in which the threads compute, but the order is undefined anyway. [10]

If the `wait` is before the `notify`, then all threads `wait` before any thread calls `notify`, and so all threads `wait` forever. [5]

A surprisingly large number of answers seemed to suggest that calling `notify` at the wrong time implicitly yields the monitor, or somehow relaxes the mutual exclusion of the monitor. It doesn’t! Only `wait` releases the monitor, but it reacquires it before returning, and so the monitor/lock is always held during `computeSomething()`.
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)?

**Multi-core.** Optional explanation: a fully event-driven server can execute multiple requests at the same time even without multiple threads, but each thread can use only a single core at a time. [See the blue slide on event-driven server to see how multiple requests can be in different stages of processing at the same time.] Modern machines have multiple cores and so they need multiple threads to use all of those cores. I gave a few points for “concurrently”, and I gave the benefit of the doubt to “parallelism”.

(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?

When an event/request/task arrives, it is placed on a queue and a worker thread is awakened, e.g., with a **notify**. When a thread holds the mutex/lock/monitor on the queue and extracts an event/request/task from the queue, it “knows” it was selected to handle it. I gave full credit for any answer that suggested an **event queue with a monitor** (mutex and condition variable).

(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?

By posting a message/event on the main thread’s **event queue**. The AsyncTask is an object with a thread. The thread may report progress by calling an AsyncTask method, which posts a progress update event on a designated queue. The main thread handles events one at a time from its queue: it receives the progress update at some time in the future, whenever it is ready.

(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?

The atomic instruction **returns the "old" value** of the lock variable, e.g., in a register. **Spin or block**.

Atomic instructions such as test-and-set or compare-and-swap return the value of the read before writing to the target location, e.g., by leaving the target’s "old" value in a register. If the old value shows the lock was already held, the thread has lost the race. It should spin or block and retry until it succeeds in setting the lock when the lock is free.
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?

Sure, that’ll work. What could go wrong? The shell runs commands, which are just programs. The shell itself is just a program like any other. Many students seemed convinced that this could not work, and came up with some creative and fun ways that it might fail (but they were generally nonsense). My favorite answer was “it works --- I did it by accident hundreds of times”.

(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?

open and dup* (e.g., dup2). In a proper shell these system calls are done in the child. The parent does no system calls for this purpose: the parent just forks the child, which sets up its own environment before exec*.

(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?

Yes, it can occur because the order in which the children run is undefined. The bytes are stored in the pipe buffer until the reader is ready to retrieve them. If the writer fills the buffer, then it blocks until the reader starts reading. If no process holds the read side of the pipe, then the writes fail. But there is always such a process: the parent holds the pipe open until it has forked both children.

(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?

Yes, it can occur because the order in which the children run is undefined. The reader blocks until the writer deposits some bytes in the pipe buffer. If no process holds the write side of the pipe, then the read returns an EOF. But there is always such a process: the parent forks the left child (the writer) before the reader, and in any case it holds the pipe open until it has forked both children.

(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?

It has to dup the left-side pipe onto its stdin and the right-side pipe onto its stdout. The child does it before exec*: the parent cannot do it because the parent must preserve its own stdin and stdout (e.g., the tty). The child “knows” to do it because the child gets a snapshot of the parent’s memory at the moment of the fork, including data structures that tell the child what to do (e.g., a list of process objects for the current job, and a pointer to a specific process in the list).
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.

Break the VA into a Virtual Page Number (VPN) and a byte offset in the page. Index the page table with the VPN and examine the PTE. Conceptually this is just like indexing into any array. Extra smiles if you told me about the hierarchical page table structure or that the page table to index is the one whose base is loaded into the core’s Page Table Base Register. Few did.

This feels like it should be nasty math, but it is very easy. There are 4K bytes on each page, 4K = 4*1024 = 2^2 * 2^10 = 2^{12}. So the offset is the low-order 12 bits, or three hex digits (16=2^4, so 4 bits per hex digit) = 0x014. The high-order 20 bits (32-12=20) are the VPN, so VPN=0x00002=2. If you are weak on power-of-two math (we both know who you are!) take the time to bone up and see how easy it is.

(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?

Fault and let the kernel handle it (parts c and d). The machine raises a fault to redirect control to the OS kernel. Many answers suggested this was an error, as in “segmentation fault”. But it is just a miss in the page table, which is itself a cache.

(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?

Short full-credit answer: vm_map. The OS determines if the faulting address lies within some valid segment of the current virtual address space, and if it is allowed for that segment (read, write, execute permission). For example, it might run down a linked list of segments (vm_map). If the access is not valid for some attached segment, THEN it is an error: e.g., a segmentation fault or protection fault.

(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.

If the segment is backed by a file, then find the block location on disk by indexing the inode block map, and read that block into a freshly allocated frame of machine memory. If the segment is anonymous (stack, heap) then zero-fill the frame. Install a page table entry to map the virtual page to the frame, and return from the fault.

It is possible that the page already resides in memory. One example we discussed is a write (store instruction) to a copy-on-write page: the machine raises a fault because the page is write-protected. The OS copies it to a new frame, and maps the virtual page to the new frame, and enables writes. There are other cases.
Exam scores (ranked)
Here we see that surprisingly few students really understand the basics of VM (question 5), and many did not take the time to study the material on threads that was presented in the last two weeks before the exam (questions 2 and 3). People did much better on the questions pertaining to the Unix system call interface (shell lab).