(a) What is the output generated by this program? In fact the output is not uniquely defined, i.e., it is not always the same. So please give three examples of possible outputs.

```c
int i = 0;
main()
{
    printf("%d\n", i); /* print i on a line */
    fork();
    i = i + 1;
    printf("%d\n", i);
    fork();
    i = i + 1;
    printf("%d\n", i);
}
```

0121222
0122122
0112222
(In columns, of course.)

(b) Briefly justify/explain your answer for (a). Try to characterize the set of all possible outputs.

The program executes in a process. The first fork creates a child process. After the first fork, both parent and child execute the remainder of the program. Both execute the second fork, creating two more processes. Refer to the figure on the following page.

The processes may execute in various orders/schedules, including the schedules that produce the outputs listed for part (a) above. In particular, each legal schedule corresponds to a traversal of the tree in which each node of the tree is visited before any of its children: a process can run only after the fork that creates it.

Part (a) is worth 30 points: 15 for one correct output, 25 for two demonstrating an awareness of concurrency, and 30 for three. Any clear explanation gets the rest of the points. Some answers gave alternative outputs presuming that one or both of the forks failed. These got some partial credit.
int i = 0;
main()
{
    printf("%d\n", i);
    fork();
    i = i + 1;
    printf("%d\n", i);
    fork();
    i = i + 1;
    printf("%d\n", i);
}
Part 2. Inside the shell

A typical Unix shell executes many of its commands by invoking (launching) a named program in a child process. In some respects that is similar to calling a local subprogram (e.g., a procedure in a library). In particular, the caller can pass arguments into the (sub)program, the program runs until it completes, and then the program returns a result to the caller.

(a) How does your Lab #2 shell pass arguments into a program running in a child process? Feel free to illustrate with drawings or pseudocode. How is this different from passing arguments into a local subprogram?

The shell uses exec* syscall (execvp, execve) to “launch a named program in a child process”. Exec* allows the caller to pass (argc, argv[]) as arguments. The kernel installs the arguments in the program’s virtual address space and passes them to the program’s “main”. Other useful detail: the arguments are limited to strings. The argv is a variable-sized array of variable-length strings. These are copied through the kernel, but the kernel does not ever interpret them. Also, an array of environment variables (strings of the form “name=value”) are preserved in the address space across exec*, and are also passed into the program’s main(). The kernel does not interpret environment variables either.

Arguments are passed to a local subprogram in registers or on the stack, with no kernel involvement. They may include data of any type, including pointers to data of any type anywhere in the address space.

2a is worth 10 points. I gave at least 7 for any answer that mentioned exec syscall and argv. Some answers focused on stdin/stdout and did not mention exec syscall or argv. I gave those 3/10 in general.

(b) How does your Lab #2 shell obtain a result from a program running in a child process? Feel free to illustrate with drawings or pseudocode. How is this different from obtaining a result from a local subprogram?

A child process may pass a result (status) to the exit() syscall when the program completes. The parent obtains the result via the wait*() syscall, e.g., waitpid. Other useful supporting detail: the exit status is limited to a single integer. It is copied through the kernel.

A local subprogram may return one or more results in registers or on the stack, or by writing any data anywhere in the shared address space, with no kernel involvement. The results may include data of any type.

2b is worth 10 points. I gave at least 7 for any answer that mentioned exit/wait status. Some answers focused on stdin/stdout. I gave those 3/10 in general.
(c) The shell interprets any file names (pathnames) relative to the shell’s current directory. The `cd` command changes the current directory. But `cd` is a builtin command: the shell does not fork a child process to execute a `cd`. Why? How does `cd` work?

The kernel tracks the current directory of each process. A process may change its current directory using the `chdir()` syscall. The shell implements `cd` by invoking `chdir` directly: if the shell forked a child, the child could use `chdir` to change its own current directory, but cannot affect the current directory of the parent. So `cd` would not work.

**Other useful detail:** Any syscall that takes a pathname argument (open, mkdir, rmdir, link, unlink, symlink…) interprets the pathname relative to the current directory of the requesting process, unless the pathname begins with a “/”, in which case the kernel interprets the pathname relative to the root directory. Shells and other user programs may also keep track of the pathname of the current directory, e.g., in an environment variable. However, this optional choice does not involve the kernel and does not affect the kernel’s handling of pathname arguments passed to system calls.

2c was worth 10 points. I gave 5 points for anything reasonable, but you had to mention `chdir` syscall to get all 10.

---

(d) Children of the shell also interpret file names relative to the shell’s current directory. How do they “know” what directory to use? What happens if the shell performs a `cd` while a child is running?

Like other kernel state for a process, the process current directory is inherited across `fork()`. As with other process state, the parent and child may diverge after the fork. In particular, any change to the current directory of one process (via `chdir`) does not affect the other.

2d was worth 10 points. I gave 5 points for anything reasonable.
Part 3. Fixing the plumbing

For some command lines the shell (dsh) spawns two child processes connected by a pipe.

(a) The following pseudocode proposes sequences of system calls for the parent shell and for each child to set up the pipe. Naturally, the sequences are incorrect. Point out three key errors in the sequences. For each error, write a sentence or two explaining why it is wrong and how to fix it.

Extra credit: estimate how many hours of sleep each error will cost you. Justify your answer without reference to “tcsetpgrp” or “setpgid”.

<table>
<thead>
<tr>
<th>parent</th>
<th>left child</th>
<th>right child</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork();</td>
<td>pipe();</td>
<td>close(…);</td>
</tr>
<tr>
<td>wait*(…);</td>
<td>close(…);</td>
<td>dup2(…);</td>
</tr>
<tr>
<td>fork();</td>
<td>dup2(…);</td>
<td>exec*(…);</td>
</tr>
<tr>
<td>wait*(…);</td>
<td>exec*(…);</td>
<td></td>
</tr>
</tbody>
</table>

(1) Do the pipe() syscall in the parent, and not in a child. The parent must do it so that each child may inherit the file descriptors for both ends of the pipe. A pipe created in a child cannot enable communication with a sibling.

(2) Fork both children before waiting for either to exit. As structured, the first child must complete before the second child can start. It is then impossible for them to run concurrently as intended. Worse, in a typical shell the left child leads the process group for the job, and the process group is destroyed when the leader exits, causing errors in any attempt to bind a new process to the same group after the second fork.

(3) The parent should close both ends of the pipe. The kernel reference-counts the pipe descriptors: if the parent holds the write end open, then the kernel presumes the parent may intend to write to the pipe, and it does not terminate the reader (right child) properly when the writer (left child) exits.

(4) Each child should close whichever end of the pipe it is not using (e.g., the close before the dup2), and then close the original descriptor for the other end after the dup2.

3a was worth 20 points. 7 points each for any clear identification of the first two major bugs, and six points for a third. Partial credit as warranted. Some students hypothesized errors not shown in the pseudocode, like passing the wrong arguments or not checking the error returns. But the question asked for bugs in the sequences. I didn’t give much if any credit for those answers.
Once the pipe is established, the left child issues a series of write system calls to write bytes into the pipe, while the right child issues a series of read system calls to read bytes from the pipe. Like all system calls, read and write trap to the kernel and execute in kernel mode in the kernel space. Write copies bytes from the writer process (user space) into a kernel memory buffer for the pipe. Read copies bytes from the kernel memory buffer into the reader process (user space). If the buffer is full, then write blocks, putting the writer process to sleep until there is space in the buffer again. If the buffer is empty, then read blocks, putting the reader process to sleep until there are bytes in the buffer again. If either process blocks, the kernel context switches to the other process. If the pipe buffer is full, then the writer blocks and the reader runs to drain it. If the buffer is empty, then the reader blocks and the writer runs to fill it. The kernel may context switch at other times as it chooses. But in general it will alternate between the reader and writer to pump the bytes through the pipe.

I gave 20 points for any reasonable answer, with partial credit as warranted. I wanted you to say that read/write are blocking system calls, but did not require it. I took points off for semantically garbled explanations, but allowed for vagueness. “Producer/consumer bounded buffer” is a full-credit answer.
Part 4. Piling on the heap

Misuse of heap-managed storage (malloc/free) can cause a C program to crash. In fact, some kinds of program errors can cause the process to crash in the heap manager, even if the heap manager code itself is correct. Give three examples with reference to your solution for Lab #1. Feel free to illustrate with pseudocode or drawings.

The key point here is that the program and the heap manager reside in the same address space and are not protected from one another, at least not for program written in a type-unsafe language like C. We discussed a number of common errors for C programs using heap memory: dangling references (using a block after a call to free), memory leaks (failing to call free), buffer overflows (writing memory past the end of an allocated block), double frees (freeing the same block twice), and pointer arithmetic errors that confuse the start of a block, e.g., to pass free a pointer that is not the start of a currently allocated block.

I gave all 40 points to any two good errors with a good explanation. To get the points you had to say something about how these errors could cause a failure in the heap manager, e.g., by corrupting data structures in the heap manager, such as the freelist or block headers. Some students sketched out scenarios involving failures outside the heap manager, or bugs or limitations in the heap manager, such fragmentation. Those got a little bit of partial credit because I am a softie, and not for merit.

Note also that heap manager operations do not affect the validity of any part of the address space (except for heap region allocations, e.g., using sbrk).
Part 5. Crossing the border

(a) How does the classical Unix kernel choose the user ID for a process launched with fork/exec*?

Inherited across fork. If the program is a setuid program, then exec changes it to the owner of the program.

(b) What causes the machine to enter privileged (kernel) mode? Upon entering kernel mode, what instruction does the machine execute next, i.e., how is the value in the Program Counter (or Instruction Pointer) register determined?

Trap, fault, interrupt. Begins executing in a handler routine chosen by the kernel.

(c) What causes the machine to enter unprivileged (user) mode? Upon entering user mode, what instruction does the machine execute next, i.e., how is the value in the Program Counter (or Instruction Pointer) register determined?

Kernel can do this any time, and can control how the PC is set, along with the rest of the register context. Common case is return from trap/fault/interrupt to the next instruction. However, after a fault the kernel might set the PC to reexecute the faulting instruction. The kernel can trigger a transition to user mode at any PC of its choice: e.g., to start in main on exec, or to upcall a signal handler.

(d) What is a kernel stack? Where do kernel stacks come from? Why do they exist?

A region of kernel-space memory used to store the runtime stack of a thread executing in the kernel, i.e., in a trap or fault. They are created on thread creation. They maintain a thread's call chain and local variables when the thread is not running, e.g., to allow a thread to block in the kernel and then wake up and resume whatever it was doing.

(e) What is a stub? Where do stubs come from? Why do they exist?

A short routine that is called to initiate a protected procedure call, such as a system call or an RPC call. Stubs for system calls are in the standard runtime library. Stubs for RPC interfaces are generated during the build process by a stub generator, based on an specification of the RPC interface in an Interface Description Language.