Problem 1 -- Short Answers

Examine the following program:

```
main()
{
  int ws= -1;
          f1();
          if (fork()==0)
                    f1();
          f1();
          wait(&ws);
          return (ws>>8)&255;
}

void f1()
{
          static int i=10;
          printf("%d\n",i);
          i++;
}
```

1A) In which memory region would we find the variable i ?

1B) You can certainly run this program and see what it prints out. Explain why are there 4 lines of output?

1C) I'll bet if you run the program a hundred times, the output will always be the same. But, is the program output deterministic? Explain.

1D) After you run this program, you `echo $?` What do you get and why?

Submission for problem 1: As usual, a PDF or TXT file.

Problem 2 -- File Descriptor Tables

```
/* Assume the program below was invoked with the following command line:
          ./program <in.txt >out.txt 2>&1
   further, assume that all systems calls in the code below succeed
*/

main()
{
 int pid,fd;
  write(1,"1234",4);
  switch(pid=fork())
  {
   case -1:perror("fork");return -1;
   case 0: dup2(2,3);break;
   default: fd=open("out.txt",O_WRONLY|O_APPEND);dup2(fd,1);
                  write(1,"ABC",3);break;
  }
  /* Sketch the file descriptor tables of both parent and child processes
     when they have both reached this point.  Also sketch the open files
     "table" (struct file entries) showing the connections between them
     and then per-process fd tables as well as to the in-core inodes.
     Show the values of the f_mode, f_flags, f_count and f_pos fields.
     It is sufficient to denote the inodes by the name of the file that
     they represent, similar to how these diagrams were presented in the
     lecture notes and in class.
  */
  for(;;)
          /* endless loop */;
}
```

Submission for problem 2: Computer-generated output. Suggest that you use a sketching/draw program for problem #2 to make neat and legible diagrams. Export this to a PDF sized for letter-sized paper.

Problem 3 -- Simple Shell Program

In this assignment, you will write an extremely simplistic UNIX shell which is capable of launching one program at a time, with arguments, waiting for and reporting the exit status and resource usage statistics.

Your shell shall accept lines of input from standard input until EOF. [But see below about being invoked as a shell script interpreter!] **Don't worry about issuing a prompt, command-line editing, etc.** Each line of input is a command to be executed. For each line:

1) if the line begins with a # (this is known as an octothorpe, or a pound sign, not a "hash tag"!), ignore it.

2) split up the line into tokens (see below) and separate the I/O redirection tokens from the command and its arguments.

3) fork to create a new process in which to run the command

4) establish the requested I/O redirection (if any) in the child. If any I/O redirection fails, report the error message and exit the child with an exit status of 1.

5) exec the command. If the exec system call fails, report the error message and exit the child an exit status of 127.

6) `wait` for the command to complete

7) report the following information about the command that was just spawned: exit status or signal which killed the process, real time elapsed, user and system time. All time values must be at least millisecond precision. (This information can be obtained by using the `wait3` system call, with `times` system call, or with `getrusage` system call. Be careful of issues of cummulative vs per-child statistics.)

7A) remember the exit status, which will be needed by the `exit` built-in command (see below). The exit status is either the `WEXITSTATUS` part (the most significant 8 bits) of the 16-bit exit status word if the command ended voluntarily, or the least significant 8 bits if the command was killed by a signal.

8) If any errors are encountered at any of these steps, report the error, but **do not quit**! Just go on to the next line of input.

Syntax and Tokenization

You may assume that each command is formatted as follows:

```
command {argument {argument...} } {redirection_operation {redirection_operation...}}
```

This is an extreme simplification of shell syntax, but this is after all a course in operating systems, not compilers. The above optional arguments and operators are whitespace-delimited, i.e. they are separated by one or more tabs or spaces. This will simplify parsing and you can use `strtok` to break up the input line into its components. The real shell accepts `command argument>output` as well, but you don't have to parse that if you don't want to. You can further assume that the redirection operators (if any) will only appear after the command and arguments (if any) as illustrated above. **Note: A line that begins with the # character is a comment and must be ignored.**

Built-in commands

Implement the following built-in commands. Built-in commands do not result in the forking and exec'ing, but are handled directly within your shell. You do not need to implement I/O redirection for built-in commands.

`cd {dir}`: Change the current working directory of the shell with the `chdir` system call. If the directory is not specified, use the value of the environment variable `HOME`.

`pwd`: Display the current working directory. The `getcwd` library function or similar can be used for this. Some interesting cases can arise if you cd to a directory via a symlink. You aren't required to deal with this...just report whatever `getcwd` returns.

`exit {status}`: Exit your shell immediately, with the status value specified (an integer) as the return value from your shell. If no status is given, use the exit status of the last command which you spawned (or attempted to spawn). When your shell ends because it has reached EOF on its input, it is as if there were an explicit `exit` command, with no specified exit status, i.e. use the exit status of the last spawned command.

I/O redirection

Support the following redirection operations (note that pipes are not required since we haven't talked about them yet):

```
<filename          Open filename and redirect stdin
>filename          Open/Create/Truncate filename and redirect stdout
2>filename         Open/Create/Truncate filename and redirect stderr
>>filename         Open/Create/Append filename and redirect stdout
2>>filename        Open/Create/Append filename and redirect stderr
```

Note that a given command launch can have 0, 1, 2 or 3 possible redirection operators. A failure to establish any of the requested I/O redirections should result in an error message and the command should not be launched. You may consider it an error or undefined behavior if a given file descriptor is redirected more than once in a command launch.

Clean File Descriptor Environment

Your shell should fork and exec the command with a standard, clean file descriptor environment. Only file descriptors 0, 1 and 2 should be open, possibly redirected as above. There should be no "dangling" file descriptors. You can assume that *your* shell was invoked the by "real" shell with such a clean environment.

Example

Note that in the example below, the informational messages such as the amount of time consumed and the exit status of the command are going to stderr, not stdout. That is the proper place for them, as well as for any "debugging" output that you leave in your code. Note that `ls.out` in the example is not "polluted" by these messages.

```
$ mysh
cd /some/directory
pwd
/some/directory
ls -l >ls.out
Child process 11628 exited normally
Real: 0.002s User: 0.001s  Sys: 0.001s
cat ls.out
mysh.c
mysh
Child process 11629 exited normally
Real: 0.010s User: 0.001s  Sys: 0.001s
ls asfljsakfjasklf
ls: cannot access asfljsakfjasklf: No such file or directory
Child process 11630 exited with return value 2
Real: 0.001s User: 0.000s  Sys: 0.000s
 ./dumpcore
Child process 11631 exited with signal 11 (Segmentation fault)
Real: 0.001s User: 0.000s  Sys: 0.000s
end of file read, exiting shell with exit code 139
```

Shell Scripts

Your shell must also support being invoked as a script interpreter: In order for this to work, your shell must, if provided with with an argument, open that file and execute each line as a command (ignoring comments). I will provide a few sample scripts that your simple shell should be able to handle, along with instructions on how to test.

Exit status

Re-read the description of the `exit` built-in command if you are unsure what exit status your shell should return.

Submission

Your submission must include: source code, "screenshot" or text session grab showing your shell running. In particular, show your runs against the supplied shell script test cases.