Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page fact sheet.
- No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If your answer will not fit in the space provided, you probably do not fully understand the question.
- There are 5 questions, some with multiple parts, priced as marked. The maximum score is 50.
- When you have completed the test, sign the pledge at the bottom of this page, sign your fact sheet, and turn in the test and fact sheet.
- Note that failing to return this test, and discussing its content with a student who has not taken it are violations of the Honor Code.

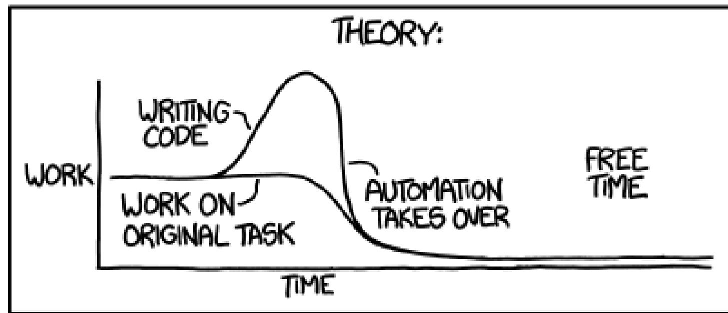
Do not start the test until instructed to do so!

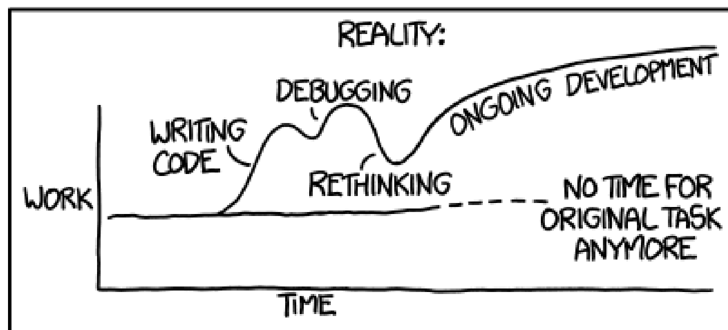
Name Solution ______________________________
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

____
signed

"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"





xkcd.com

1. [12 points] Evaluate each of the following statements. If the statement is true, say so. If the statement is false, explain why.

a) "A successful call to `fork()` creates a child process and triggers a context switch to that child process."

❑ is true.

❑ is false, because **a successful fork() creates a new process, but the scheduling of that process is entirely up to the kernel's internal logic**

b) "A context switch between processes in an OS using dual-mode operation always involves a mode switch."

❑ **is true; context switches are carried out by code that must run in kernel mode.**

❑ is false, because

c) "When user-implemented signal handlers are called, they execute in kernel mode."

❑ is true.

❑ is false, because **user code cannot safely be allowed to execute in kernel mode; a user-written signal handler may be called FROM the kernel, but it executes in user mode.**

d) "When a user process attempts to access invalid memory addresses, by default it is stopped until an administrator can examine its state and terminate the process."

❑ is true.

❑ is false, because **this leads to a SIGSEGV, and the default action is to terminate the process; no administrative action is needed.**

---

2. [5 points] Suppose that a process has `fork`'d one or more children, but all of the children have terminated before the parent process calls `wait()`. What will occur?

The call to wait() will return immediately, with the PID of a terminated child process, which will be reaped.

A common omission was to not explain that a child process would be reaped.

3. The following C program compiles and links without errors or warnings (with suitable headers). The checks to see if `fork()` and `execv()` succeed are omitted to save space; the executable `alpha` exists in the current working directory, and terminates correctly.

```
int main() {                                                          //  1

   char* cargs[] = {NULL};                                            //  2
   int local = 0;                                                     //  3
   pid_t cpid = fork();                                               //  4
   local = cpid;                                                      //  5

   if ( cpid == 0 ) {                                                 //  6
      printf("Child is %d and parent is %d\n", getpid(), getppid());  //  7
      local = local + 25;                                             //  8
      execv("./alpha", cargs);                                        //  9
   }
   else {                                                             // 10
      printf("Parent is %d and child is %d\n", getpid(), cpid);       // 11
      local = local - 25;                                             // 12
   }

   printf("%d sees local = %d\n", getpid(), local);                   // 13

   if ( cpid == 0 ) {                                                 // 14
      exit(42);                                                       // 15
   }

   int child_status;                                                  // 16
   pid_t wpid = wait(&child_status);                                  // 17

   // Irrelevant code responding to the return from wait() is omitted.
   . . .
   return 0;
}
```

When the program was executed, lines 7 and 11 produced the following output:

```
Parent is 1750 and child is 1751
Child is 1751 and parent is 1750
```

a) **[4 points]** What output would have been produced by line 13?

1750 sees local = 1726
The child process fork'd at line 4 will exec() at line 9, and leave this program; so, only the parent process executes line 13.

b) **[4 points]** State the PID of every process that would have been terminated by the execution of line 15.

None; the child has exec'd earlier, and the parent will not see cpid == 0.

c) **[4 points]** Suppose that the program `alpha` `fork`'d a child process. Assume, for this question only, that the program shown above does execute the call to `wait()` in line 17. Could that call to `wait()` return due to the termination of `alpha`'s child? Justify your answer, briefly.

No. wait() only returns *in the parent* (upon a state change of a child of that parent).

4. **[9 points]** The following C program compiles and links without errors or warnings (with suitable headers). Consider executing this program, under normal conditions, on our rlogin cluster:

```
int main() {

    int fd[2];
    assert(pipe(fd) == 0);

    char buf[5];
    printf("FORK\n");

    if ( fork() ) {
        int rc = read(fd[0], buf, 4);

        switch (rc) {
        case  4:  buf[4] = 0;
                  printf("%s\n", buf); break;
        case  0:  printf("READ-EOF\n"); break;
        case -1:  printf("READ-ERROR\n"); break;
        }
    }
    else {
        assert(dup2(fd[1], STDOUT_FILENO) == STDOUT_FILENO);

        char* args[] = { "echo", "ECHO", NULL };
        int rc = execv("/bin/echo", args);

        switch (rc) {
        case  0:  printf("EXEC-SUCCESS\n"); break;
        case -1:  printf("EXEC-ERROR\n"); break;
        }
    }

    printf("DONE\n");
    return 0;
}
```

Which of the following are possible outputs of this program? Check all that apply!

❑	FORK ECHO EXEC-SUCCESS DONE DONE	❑	FORK FORK ECHO DONE DONE
❑	FORK READ-EOF DONE	❑	FORK ECHO DONE
❑	FORK FORK ECHO EXEC-SUCCESS DONE DONE	❑	FORK READ-ERROR EXEC-ERROR DONE DONE

Comments:

The child process's stdout is redirected to a pipe from which the parent process reads. The child execs 'echo ECHO' which writes ECHO to stdout and thus to the pipe, which the parent reads from. Since execv() does not return, DONE is output only once.

Pipes keep their data even if the processes writing to them have already terminated. Thus, the above is the only possible output.

In more detail:

"FORK" will definitely be printed; ONCE. That rules out two answers.

There are no checks to see if the fork() or execv() calls succeed; we must consider several possibilities.

Suppose the fork() fails; then the child is not created, and the parent will block on read(); there will be no further output.

Assume fork() succeeds. If the assert() fails, the child will be terminated, and the parent will block on read().

Assume the assert() succeeds; then the child's output to stdout will go into the pipe.

Suppose the exec() fails (unlikely, since /bin/echo is correct); exec() will then return -1. So, the child will write "EXEC-ERROR" into the pipe; but the parent will only read "EXEC" from the pipe, which will return 4, and the parent will write "EXEC", and then write "DONE".

If the exec() succeeds, the child will write "ECHO" (and never execute the following code). The parent will then read "ECHO" from the pipe, which will return 4, and the parent will write "ECHO", and then write "DONE".

So, the middle answer on the right is possible.

5. You may recall the following comment in the course notes, regarding issues related to state maintenance in the `esh` project:

- How to maintain an accurate depiction of the state of external entities subject to change outside of the program's control, when…
- external entities can change state asynchronously, and …
- tools for monitoring state changes (e.g., UNIX signals) are imperfect

a) **[3 points]** What "external entities" that are "subject to change" does this comment refer to?

Child processes fork'd by the shell in response to user commands.

There were any number of vague answers, like "processes".

Answers like "signals" do not fit the question; the shell doesn't "monitor state changes" of signals, it uses signals to monitor state changes of…?

b) **[3 points]** Why are these changes of state "outside of the program's control?" (the "program" here refers to the shell.)

Child processes may change state for a number of reasons, including termination due to the internal action of the child process itself, user actions via the command line, kernel actions, etc.

The shell will have no control over such things.

c) **[6 points]** Give two reasons why UNIX signals are an "imperfect" tool.

Major limitations include:

- **signals are not queued, so signals may not be delivered to the targeted process**
- **handling may be due to user-implemented handlers**
- **signals cannot be cancelled**
- **a blocked signal may be delivered out-of-order, with respect to other (unblocked) signals that are sent while signal blocking is in effect**

There was a contention that signals do not carry sufficient information; the discussion of the sigaction() interface shows that considerable additional information is available to a signal handler.