**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.

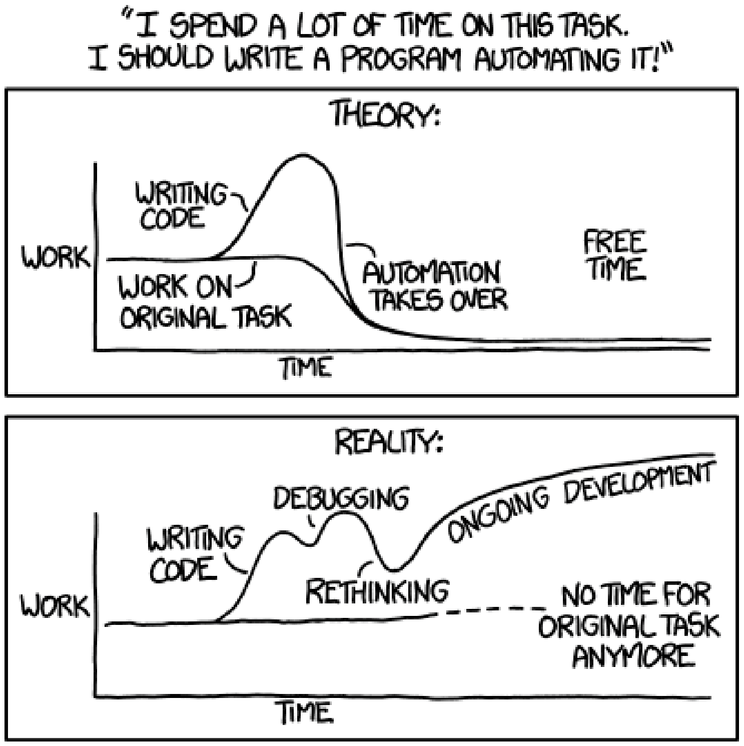
**Do not start the test until instructed to do so!**

**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!"
THEORY:
WRITING CODE
WORK
WORK ON ORIGINAL TASK
AUTOMATION TAKES OVER
FREE TIME
TIME
REALITY:
DEBUGGING
WRITING CODE
ONGOING DEVELOPMENT
RETHINKING
WORK
NO TIME FOR ORIGINAL TASK ANYMORE
TIME

**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)** "If a parent process calls `exit()` and terminates the program, any non-terminated child processes that have not `exec`'d will be terminated."

❑ is true.

❑ is false, because **a call to exit() terminates the calling process; a fork'd child is an independent process, whether it has exec'd or not.**

**b)** "A mode switch always leads to a context switch."

❑ is true.

❑ is false, because **mode switches can will occur whenever a process makes a system call, and most system calls will not lead to a context switch**

**c)** "When a process is executing on a CPU under a multitasking operating system, an interrupt handler may cause a context switch to another process."

❑ **is true; an interrupt can even lead to the termination of a process, which would, of course, lead to a context switch.**

❑ is false, because

**d)** "Processes executing systems calls always transition to the BLOCKED state where they remain until the system call completes."

❑ is true.

❑ is false, because **the BLOCKED state is entered only if the system call causes the process to wait for something that will complete in the future. For instance, if a process makes a system call to receive a packet from the network, it will be BLOCKED only if no packet has already been received and buffered. Similarly, a process trying to read from disk may block only if the data is not already cached. Some system calls (e.g., getpid()) will not block at all.**

---

**2. [5 points]** Suppose that a process that has not `fork`'d a child calls `wait()`. What will occur?

**The call to wait() will return immediately; wait() only blocks if there are *waitable* child processes (see man 2 wait).**

**A common misconception was that the call to wait() would block indefinitely; a common omission was to explain that a child process would be reaped.**

**3.** The following C program compiles and links without errors or warnings (with suitable headers). The check to see if the call to `fork()` succeeds is omitted to save space.

```
int main() {                                                          //  1

   int local = 0;                                                     //  2
   pid_t cpid = fork();                                               //  3
   local = cpid;                                                      //  4

   if ( cpid == 0 ) {                                                 //  5
      printf("Child is %d and parent is %d\n", getpid(), getppid());  //  6
      local = local + 25;                                             //  7
   }
   else {                                                             //  8
      printf("Parent is %d and child is %d\n", getpid(), cpid);       //  9
      local = local - 25;                                             // 10
   }

   printf("%d sees local = %d\n", getpid(), local);                   // 11

   if ( cpid == 0 ) {                                                 // 12
      exit(0);                                                        // 13
   }

   int child_status;                                                  // 14
   pid_t wpid = wait(&child_status);                                  // 15

   // 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 2010 and child is 2030
Child is 2030 and parent is 2010
```

**a) [4 points]** What output would have been produced by line `11`?

2010 sees local = 2005
2030 sees local = 25          (in either order)

The parent sets local to 2030 and then subtracts 25; the child sets local to 0 and then adds 25.

**b) [4 points]** State the PID of every process that would have been terminated by the execution of line `13`.

The child process, 2030.

**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 `15`. Could that call to `wait()` return due to the termination of `alpha`'s child? Justify your answer, briefly.

*A good answer depends on what program you decided alpha referred to.*

*Assuming alpha is the program shown above, yes . . .*
*If alpha is some other program, then the answer is no . . .*

**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<br>ECHO<br>EXEC-SUCCESS<br>DONE<br>DONE | ❑ | FORK<br>FORK<br>ECHO<br>DONE<br>DONE |
| ❑ | FORK<br>READ-EOF<br>DONE | ❑ | FORK<br>ECHO<br>DONE |
| ❑ | FORK<br>FORK<br>ECHO<br>EXEC-SUCCESS<br>DONE<br>DONE | ❑ | FORK<br>READ-ERROR<br>EXEC-ERROR<br>DONE<br>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.**

**"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 arising in currency control in the `esh` project:

- How to ensure the correctness of data when …
- the control flow is subject to asynchronous interruption, and…
- there are complex control flows in shell

**a) [3 points]** Give one example of data the shell maintains that is at risk in the face of asynchronous interruptions.

- list of state information related to current child processes

**b) [3 points]** Give one example of something that might cause the "asynchronous interruption" referred to here.

- termination of a child process (normal or otherwise)
- user-entered keyboard sequence triggering state change in child process

**c) [6 points]** Give one example of something the shell can do to ensure the "correctness of this data" in the face of asynchronous interruptions.

Block any signal that would trigger a handler, when operating on data that is accessed or modified by that handler.

Simply saying "block signals" is imprecise.

Simply saying "write signal handlers" is even less precise.