CSE190 — Midterm

Yee **Spring '00**

Name and Class Account Login: Answer Key

There are a total of 10 questions on 8 pages. There are 100 points possible. You may not finish the entire exam — don't be discouraged. Wait until the instructor has passed out exams to everybody before you start. Advice: skim through the entire test to determine which of the problems you can solve quickly and work on those first, rather than getting stuck on a hard problem early and wasting too much of your time on it.

When you can start, you should first make sure that you have all the pages, and write your name and your login name on the first page, and your login name on the top of *all subsequent pages*. Pages of this exam will be separated and graded separately — if you fail to write your name at the top of a page, you will not receive credit for answers on that page. **Write clearly**: if we cannot read your handwriting or your pencil smudges, you will not properly get credit for your answers.

This exam is closed book. You are allowed a single sheet of notes You may look at your *own* notes all you want. You may **not** look at anybody else's books, notes, exam, or otherwise obtain help from another human being, artificial intelligence, metaphysical entity, or space alien. If I see your eyeballs wandering, you will get a zero for the exam. If you must look away from your exam/notes to think, look up at the ceiling / into space or close your eyes.

No electronic computation aids are allowed.

Problem	1	2	3	4	5	6	7	8	9	10	**Total**
Score											
Possible	8	7	8	10	16	10	10	10	20	1	100

1 (System Calls) (A) What are system calls? (B) How are they directly invoked (when possible) from programs written in various programming languages?

(8pts)

1: System calls are the lowest-level services provided by the operating system kernel.

2: The binary interface to system calls typically use some kind of `trap` instruction. Language-specific bindings to system calls, if such direct bindings exist, are typically in the form of library routines (e.g., libc) that provide routines that implement the language's calling sequence and then trap to the kernel.

2 (Standard C Library) What does the standard C library routine `exit` do?

(7pts)

The `exit` routine performs `atexit` (and `on_exit` for some libc implementations) processing, flushes all I/O buffers, and then invokes the `exit(2)` system call to terminate the current process.

3 (Kernel Abstractions) If your process is reading from a descriptor that refers to a pipe, (1) in what manner does it get an EOF indication? (2) precisely when will this indication be generated?

(8pts)

An EOF on a pipe is like an EOF on any other I/O descriptor: it is indicated by a zero return value from a `read` system call.

An EOF indication is generated by the kernel as a result of a `read` when the pipe buffer is empty *and* there are no references to the write end of the pipe, i.e., there is no way that any new data can be made available for the `read`.

4 (Shell Syntax) Suppose program `p` outputs two data streams to standard output (descriptor 1) and to standard error (descriptor 2). You are using either the Bourne shell or bash.

(1) Run `p` so that the outputs that `p` writes to descriptor 1 will be sent to the standard error (descriptor 2) of the invoking environment, and vice versa, i.e., swap the descriptors.

(2) Write the shell command(s) that will capture the standard output data into a file `p.log` and pipe the standard error stream through a paginator program such as `more` or `less`.

(10pts)

```
p 3>&1 1>&2 2>&3 3>&-
p 2>&1 > p.log | more
```

5 (System Calls) Describe what do the following system calls do:

1: `fork`

2: `execve`

3: `dup` and `dup2`

4: `pipe`

(16 pts)

1: `fork` creates a new process, called a *child process*, that is identical to the calling process, except that the return value of the `fork` system call is 0 for the child, and the process ID of the child process for the parent.

2: `execve` replaces the current process with the process image specified by its argument. All memory is replaced, but state information such as the current working directory, ignored signals, etc, are retained. Caught signals revert to their default behavior.

3: `dup` and `dup2` creates a new descriptor which will refer to the same object as the argument descriptor. `dup2` allows the calling process to specify the value of this descriptor; if this descriptor is already in use, it is first closed.

4: `pipe` creates a new pipe object, which is a kernel-managed FIFO used for inter-process communication.

6 (Processes) Enumerate all the state information that is inherited by a child process from its parent.

(10pts)

It is okay to have fewer than listed here. I expect at least 3 of these.

1: Current working directory

2: I/O descriptors

3: Signal handler state

4: All memory contents (includes environment variables)

7 (Shell Syntax) What does the shell syntax

```
cmda && cmdb
```

do? What is the exit status of this compound command?

(10pts)

It runs `cmda` and test its exit status. If it is failure (non-zero), the exit status of the compound command is that of `cmda`. If `cmda`'s exit status is success (zero), then `cmdb` is run and the exit status of the compound command is that of `cmdb`.

8 (Shell Syntax) What does the following command output to standard output? To the file `foo`?

```
i=0
while echo hi; test $i -lt 2
do
        i=$(($i + 1))   # bash notation for incrementing $i
        echo bye
done > foo
```

(10pts)

It outputs nothing to standard output. The contents of the file `foo` is

```
hi
bye
hi
bye
hi
```

9 (Shell Syntax Extension) Suppose you wish to extend the Bourne (or descendant) shell to enable you to use a "not" keyword to write

```
if not test_command args
then
        command_A
        command_B
fi
```

instead of the slightly more awkward

```
if test_command args
then
        ;
else
        command_A
        command_B
fi
```

Describe how this is done, and write the C code to implement it. Don't worry if you can't remember all of the necessary `include` files, just convey precisely what system calls are to be invoked under what conditions.

(20pts)

We write a program named `not` which just runs its argument as a subprocess and exits with success when the subprocess exist with failure and vice versa.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

int     main(int        argc,
             char       **argv)
{
        int     pid, status;

        switch (pid = fork()) {
        case -1:
                perror("not");
                exit(1);
        case 0:
                execvp(argv+1,argv+1);
                perror("not");
                _exit(0); /* will be negated by parent */
        default:
                waitpid(pid,&status,0);
                if (WIFSIGNALED(status)) {
                        kill(getpid,WTERMSIG(status));
                }
```

```
                /*
                 * only WIFEXITED is possible, since we did not ask
                 * for the WUNTRACED option.
                 */
                exit(WEXITSTATUS(status) == 0);

        }
}
```

10 (Unix History) Is Prof. Yee older than Unix?
(1pt)

Yes. (This is a freebie.)