Lab Exercise #4 -- UNIX Process Management

This exercise focuses on the system calls and library functions available under Solaris and Linux to create and coordinate processes.

A. The "fork" System Call

The only way to create a new process under UNIX is for an existing process to invoke the "fork" system call. When the existing process (the parent) calls "fork", the kernel creates a new, separate process (the child). The child process is an exact copy of the parent process, except that it has a different process ID and all of its "accounting" information is re-initialized.

Note that the kernel copies the data space of the parent process at the moment that it calls "fork'; thus, both the child and the parent have their own data space from that point forward.

The prototype for "fork" is given below ("pid_t" is the same as "long int"):

```
pid_t fork();
```

When "fork" is called, the kernel creates the child process, then returns to both the parent and the child processes. However, the value which is returned is different: the parent receives the process ID of the child, and the child receives the value 0. If the system call fails, it returns a negative value.

1) Examine the program in "lab04.fork.c", then translate and execute it.

a) Give the output produced by the two processes.

PID: _____ A: _____ B: _____

PID: _____ A: _____ B: _____

PID: _____ A: _____ B: _____

PID: _____ A: _____ B: _____

PID: _____ A: _____ B: _____

b) Explain the changes to the values of variables "A" and "B".

2) Copy the program into your account, then modify it to have only the parent process voluntarily surrender the CPU for 2 seconds using the "sleep" function (for example, "sleep(2);"). Does the call to "sleep" alter the output produced by the two processes? Does it alter the final value of the variables in the two processes? Explain.

B. The "wait" and "exit" System Calls

It is often useful to ensure that the child process completes before the parent process continues with its processing. The "wait" system call causes a process to block, waiting on the completion of a child process:

```
pid_t wait( int *status );
```

Function "wait" returns the process ID of the child that finished, and stores the status of that finishing child process in the location pointed to by its argument ("status", in the prototype above).

When a child process uses the "exit" function, it terminates and passes an integer status flag to the parent process, which can be interpreted by the parent to find out what happened to the child. Its form is:

```
void exit( int );
```

In general, termination of a process generates a signal which says something about how that process halted.

1) Examine the program in "lab04.wait.c". Note the definition of the "display_status" function and the use of the "W****" macros that test the status of the "wait" function.

a) Give the output when the program is translated and executed.

b) What line in the source code needs to be changed to get the first line of output to have an exit status other than 1?

c) What integer value corresponds to an arithmetic exception signal?

d) Explain why none of the child processes call the "wait" function.

2) Copy the program into your account, then modify it to create a fourth child process which terminates due to a segmentation fault. Does the addition of the fourth process alter the output produced? Explain.

C. The "exec" System Call

Function "fork" is the way to duplicate a process, and function "exec" is the way to make that duplicated process perform a different task.

There are six forms of the "exec" function, each with its own special characteristics. Two of those are outlined below:

> Function "execle" expects its arguments to be the full pathname of the program to execute, a list of command-line arguments (terminated by a NULL pointer), and a set of environment variables (as in function "main").
>
> Function "execlp" expects its arguments to be the relative pathname of the program to execute, and a list of command-line arguments (terminated by a NULL pointer). It inherits the current set of environment variables.

1) Examine the program in "lab04.exec.c", then translate and execute it.

a) What are the names and locations of the programs that get executed by the calls to "exec"?

b) Explain the difference in output produced by the two different uses of the "exec" functions.

c) Move to a different directory, then execute the program again. Explain why an error is reported.

2) Copy the file "lab04.exec.c" into your account and modify the program to use "execvp" instead of "execlp". Note that "execvp" uses an array of C-style strings, where each string is a command-line argument (as in function "main").