Assignment 1: Fork

Recitation for ECES 338
February 1, 1999

System Calls

`pid_t fork(void)`

The `fork()` system call creates a new process. The new process (child process) is an exact copy of the calling process (parent process). This means that the child process contains all of the same variables, values, and code as the parent process.

However, there will be one (important) difference between the parent and child processes. Basically, the `fork()` system call will return different values, depending on whether-or-not the process is the parent or the child.

To the parent process, `fork()` returns the `pid` of the new child process that it made. To the child process, `fork()` returns zero. Thus, whenever using `fork()`, it is pretty much imperative that you wrap it up in a conditional. Basically, you should have code that looks like this:

```
if (fork() == 0)
        {
        /* Child-only code here */
        }
else
        {
        /* Parent-only code here */
        }

/* Common code here */
```

It's important to note the distinction that I made here -- the new child process contains **all** of the code in the parent process, even the stuff that falls outside of it's conditional. So,

you have to be careful that the child process only runs the code that it is supposed to – and not the parent's code.

`int execl(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/);`

Overlays a new process image on an old image. The first argument is the pathname to the executable, the following variable argument list consists of the parameters to said executable. It is imperative to make the last parameter `NULL`.

`pid_t getpid(void)`

Returns the process id of the current process.

`pid_t getppid(void)`

Returns the process id of the parent process.

`char *getlogin(void)`

Returns a pointer to a string that contains the current login name. You don't have to `malloc()` any strings for this system call.

`int getrusage(int who, struct rusage *rusage)`

This function can be used to query several important characteristics of a running process. The first parameter can be either `RUSAGE_SELF` or `RUSAGE_CHILDREN`, specifying that the function should collect statistics on its won process or it's child processes, respectively. This function will fill out an entire structure full of data. Here is a partial definition of this structure:

```
struct rusage {
       struct timeval ru_utime; /* user time used */
       struct timeval ru_stime; /* system time used */
       ...
};
```

In particular, we are only interested in two fields: `ru_utime` and `ru_stime`, which represent the time spent in user and system modes, respectively. These two variables are of type `struct timeval`, the definition for which I have included here.

```
struct timeval {
        long    tv_sec;         /* seconds */
```

```
        long    tv_usec;        /* and microseconds */
};
```

`void _exit(int status)`

This system call terminates the current process, returning the contents of `status` as the return value. According to the man page for `fork()`:

> `Be careful to call _exit() rather than exit(3C) if you cannot execve(), since exit(3C) will flush and close standard I/O channels, and thereby corrupt the parent processes standard I/O data structures. Using exit(3C) will flush buffered data twice. See exit(2).`

`pid_t wait(int *stat_loc)`

This function waits for a child process to terminate. If this occurs normally, it will return the `pid` of the child that exited. If `stat_loc` is not `NULL`, then `wait()` will place the return value of the child process in the area of memory pointed to by `stat_loc`.

`char *ctime(const time_t *clock)`

Takes a pointer to a `time_t` variable (generated by `time()`), and returns a pointer to a 26 character string, which prints out the current date/time in human-readable format.

`time_t time(time_t *tloc)`

Returns the number of seconds since January 1, 1970.

Header-files needed:

```
<sys/types.h>
<unistd.h>
<sys/wait.h>
<stdlib.h>
<time.h>
<sys/resource.h>
```

What to hand-in:

1. Source code
2. Script output
3. Answers to the questions

About the grading:

If the assignment works, then it will automatically receive 95% of the grade. However, I am a bit of a nitpick when it comes to the details, so I will use the last 5% in order to differentiate between 'perfect' and 'working' assignments, based upon the following order:

1. Error-Handling
2. Readability (includes style, modularity)
3. Comments
4. Output