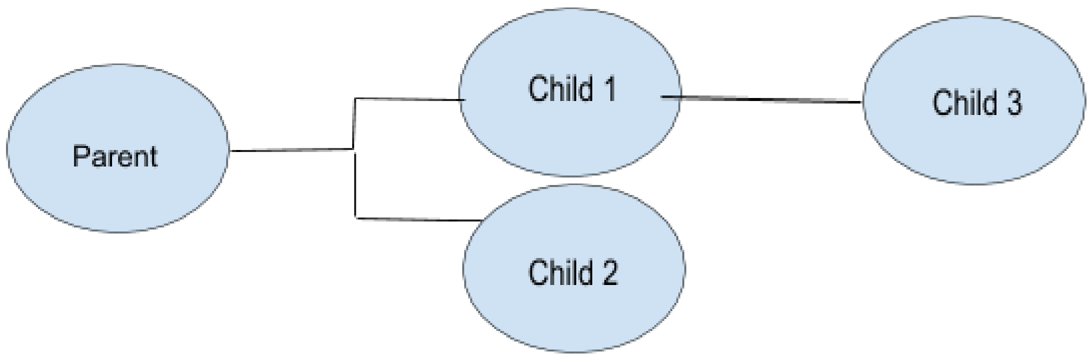
Matt Kennedy
William Shreeve

CIS 452 - 10

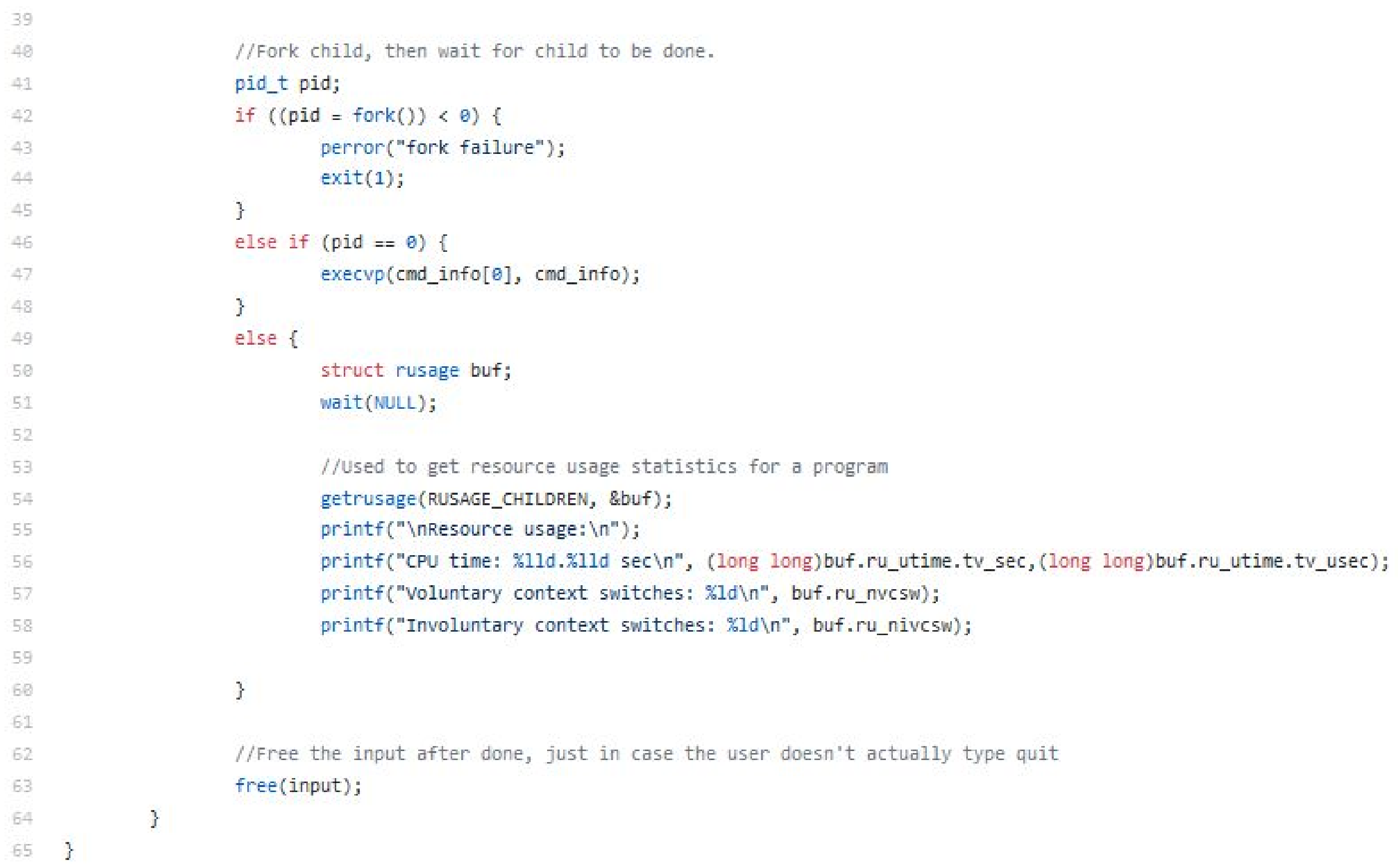
Lab 2 - Process Management Concepts

1. How many lines are printed by the program?
    a. Three lines are printed by the program:
        i. Before fork
        ii. After fork
        iii. After fork
2. Describe what is happening to produce the answer observed for the above question.
    a. The reason why 'After fork' prints twice is because when fork() is called, it creates another of the same process from the fork statement, and completes it. This is proven by using the command 'ps' to see that before the fork is called, only one process of a.out is running. After fork is called, and for the rest of the program until completion, there are two a.out processes.
3. There are two a.out programs listed in the table. The PID's are 18680 and 21730. Both a.out programs share the same ADDR_SZ value, which signifies they are in the same address space. Additionally, the F flag is 1 for the child a.out, signifying it is forked.
4.



5. Using an example input of 2: The program will reach the fork lines. At this point, child 1 and 2 are created. The parent is going to finish executing before child 1 and 2 continue. Now, child 1 and 2 are executing simultaneously. This is why we 2 lines executing at the same time (i.e. both PID print's, both 0's, then both 1's). Both children are being scheduled, a task is being completed, then they are swapped out with the other. Lastly, child 1 spawns another child, as there is another fork line after the first. This will run after both children finish, with all lines printing normally.
6. waitpid(child, status, 0);
7. Child prints first. The child is forked, then the wait command will wait for the child to finish and return a status, at which point the parent continues executing.

8. In the waitpid function, the status variable we pass in is assigned the exit status code that the exit call parameter is. If the value is not 0, this signifies an error, and that is reflected in printing status. It is always a multiple of 256 due to the fact that the value is encoded. To further analyze this, we would use provided macros in the sys/wait library.
9. Never. The execvp will replace the current process, so anything that happens after execvp doesn't happen since the program stops when the function returns. To fix this, we would have to fork the process, then call exec.
10. The second parameter &argv[1] is for being able to read and execute any options included with the first.
    a. For example, ./a.out ls -la uses argv[2] = -la for options. Because execvp has the pointer to the array, it is able to iterate through it and use the following parameters as options, like a typical command line could.



```

        //Fork child, then wait for child to be done.
        pid_t pid;
        if ((pid = fork()) < 0) {
            perror("fork failure");
            exit(1);
        }
        else if (pid == 0) {
            execvp(cmd_info[0], cmd_info);
        }
        else {
            struct rusage buf;
            wait(NULL);

            //Used to get resource usage statistics for a program
            getrusage(RUSAGE_CHILDREN, &buf);
            printf("\nResource usage:\n");
            printf("CPU time: %lld.%lld sec\n", (long long)buf.ru_utime.tv_sec,(long long)buf.ru_utime.tv_usec);
            printf("Voluntary context switches: %ld\n", buf.ru_nvcsw);
            printf("Involuntary context switches: %ld\n", buf.ru_nivcsw);

        }

        //Free the input after done, just in case the user doesn't actually type quit
        free(input);
    }
}
```