Signal6 Demonstration

Files

The files for this demonstration can be found in the rlogin cluster in the directory

`/web/courses/cs3214/spring2014/butta/examples/signal-demo/signal6`

The files are `esh-sys-utils.c esh-sys-utils.h list.c list.h Makefile print-pid.c print-pid.h race.c norace.c`

The `“make” command` by default will create an executable named `race`. A variant of the program is created by `“make norace`”. This program creates many child processes and maintains a list of process ids of child process that have been created but not yet terminated. The termination of a child process is detected by installing a signal handler for the `SIGCHLD` signal. Output is generated whenever a child’s process id is add to or removed from the list. The output is generated using reentrant code (`snprintf` and `write`). At the end of its execution the program outputs “`Terminating Successfully`.”

Purpose

The purposes of this demonstration are

- to see how a parent process can detect the termination of a child process
- to see an instance of a “race condition”
- to see how `gdb` can be used to examine a stopped process
- to see how to resolve the race condition by temporarily blocking the delivery of a signal.

Part 1: Steps

1. Use the `Makefile` to create the executable program `race` using the command `“make”`.
2. At the shell prompt execute the `race` program.
3. Allow the program to run until either the program terminates or the program output ceases without a shell prompt being given. In the later case, end the process by using a `control-c.`
4. Use the `ps` and `kill` commands to get rid of any child processes that might remain.
5. To understand what is happening run the `race` code again. When the program stops producing output use a `control-z` to stop the `race` code.
6. use the ps command to find the process id of the parent race process (it will be the process with the smallest id among all those running the race code and not labeled <defunct>). This process id will be referred to as `<pid>` below.
7. Examine the stopped race code using gdb as follows:
    a. at the shell prompt enter: `gdb ./race`
    b. at the gdb prompt enter: `attach <pid>`

8. The line of code being executed is likely the following one in the `remove_child` function:

   ```
       while(current && current->child_pid != pid)
   ```

9. Use the gdb `step` command to execute the next line of code and the gdb `print current` command to examine the value of `current`. Note: `current` is pointing to an element in the list of pids.
10. Use the gdb commands to examine the list of pids: `print head` will show the address of the first element of the list, `print head->next` will show the address of the first element of the list, `print head->next->next` will show the address of the second element of the list, and so on.

Part 2: Steps

1. Use the `Makefile` to create the executable program `norace` using the command `“make norace”`.
2. At the shell prompt execute the `norace` program.
3. Allow the program to run until either the program terminates or the program output ceases without a shell prompt being given. In the later case, end the process by using a `control-c.`
4. Compare the `race.c` and `norace.c` code. Identify the difference in these two programs. Hint: two lines of code are different and they are near the bottom of the main program.

Questions

Based on your observations, answer these questions.

1. Does the process executing `race` always complete normally?
2. Based on your use of gdb in step 9 what diagnosis can you offer to explain why the program ceases to produce output and terminate normally?
3. Based on your use of gdb in step 10 what it is about the list of pids that causes the behavior leading to your diagnosis?
4. Does the process executing `norace` always complete normally?
5. Explain why the difference between the `race.c` and `norace.c` code accounts for the difference in their behaviors.