Comp111 Assignment 3: sandboxing a process

Overview

A very common problem in process management is to manage the behavior of a process whose source code and behavior you cannot control, e.g., a student-submitted program for a programming class. A systematic problem in grading student programs is that their program can do anything at all that the user who is running the program can do.

One solution to this problem is to create a "sandbox" for the process to be graded, that keeps the process from doing things it should not be able to do. Sandboxing is so important that whole operating systems (most recently, Google Chrome OS -- a version of linux not to be confused with the Chrome browser) have been written to provide the service.

In this assignment, you will write a program that executes another program (e.g., a program to be tested) in a controlled environment. When executing in this environment, the program should only do things that are explicitly allowed according to a predetermined policy, and should be killed (and a note made on stderr) if the program does anything not allowed by the policy.

Just to make this interesting, I am going to give you some behaviors to stop and to identify. I will provide several misbehaving programs and you have to keep them from misbehaving. Each program will try to take over the resources of the whole machine and keep everyone from working. To be fair about this, I am going to craft them so that a simple control-C will stop them; in the worst case I could craft them so that this won't work!

To get the code for this assignmnet:

```
cp /comp/111/assignments/a3/*.c ./
```

and optionally, if you want the Makefile:

```
cp /comp/111/assignments/a3/Makefile ./
```

Objectives

Your objective is to write a C program watch that, when invoked, runs another given program and attempts to control behaviors of the child process. The program to run should be specified in watch's first argument.

```
watch ./a.out
```

should invoke your program watch to invoke and watch the execution of the child process a.out. watch should react to several conditions in the child process with preventative actions, and report each prevented behavior and action to stderr.

1. If the child occupies more than 4 MB of stack memory, it should be killed and this event should be reported. The program `1.c` does this.
2. If the child occupies more than 4 MB of heap memory, it should be killed and this event should be reported. The program `2.c` does this.
3. If the child forks, it should be killed and the event should be reported. The program `3.c` does this.
4. If the child creates a thread, it should be killed and the event should be reported. The program `4.c` does this.
5. If the child opens any file (other than the pre-opened files stdin, stdout, stderr), this should be prevented and the program should be terminated. The program `5.c` does this.

When the child dies for any reason -- including a normal exit -- its exit status code, total runtime, number of lines printed to stdout, and (wallclock) time of death should be reported. In all of the above cases, one should:

1. Report the behavior on stderr.
2. Kill all instances of the child with an unblockable kill (-9).
3. Report the child's exit status code, total runtime, number of lines printed to stdout, and (wallclock) time of death, as if it had died normally.

After any of these, the `watch` program should exit.

Part of the problem is to distinguish which behaviors can be controlled through the operating system itself, which ones can be controlled through monitoring, and which can only be controlled partially due to extenuating circumstances. Some of the above conditions are easy to assure, and *some are impossible to completely control.* Some can be "controlled" but one cannot be sure exactly which condition was violated. It is part of your task to determine which parts of these requirements are *possible* to accomplish, and which ones are not!

Extra Credit

For 10% extra credit (1 point of 10), prevent the program from declaring more than 1 MB of global variables, as embodied in `6.c`

Where to work

Since we are going to be doing something "dangerous", we will work on our own machines, `comp111-01` through `comp111-06`. These machines are *inside the firewall* and cannot be accessed directly. To work there, *ssh* to `homework.cs.tufts.edu` *first*, then to `comp111-01.cs.tufts.edu` (or `-02` to `-06`). **It is important that you do *not* test this program on homework.cs.tufts.edu, especially since you will be running (and trying to control) intentionally antisocial programs!**

If you manage to crash one of the servers, please don't try to run your program on the other one until you're sure that you've fixed it. I say this because in the past, students have managed to crash *all* servers from a single cause. This is why there are six of them!

Notes

1. You must thwart all behaviors not conforming to the requirements, not just the behaviors of the simple sample programs.
2. You must forward everything the child prints to *stdout*, even if you capture it yourself. Accordingly, you should not print anything of your own to *stdout*; use *stderr* (that's what it's for).
3. You cannot use privilege in any way. All processes must run as your user, not root.
4. It is not permissible to rewrite the operating system! (I only say this because people have tried this in the past!)
5. It is possible to invoke the *gdb* interface and single-step the child program, but this is very difficult and not recommended.
6. There are more informative versions of "`wait`" that you can use instead of "`wait`", "`wait3`". See `man waitpid`, `man waitid` for details.
7. Read up on "resource limits" (`man setrlimit`) as a hint to addressing some of the requirements. Take care just to limit the child and not watch itself!
8. Some child conditions can be detected via signals. Use signals if at all possible! In particular, you will want to trap SIGCHLD in the parent and react accordingly!
9. Part of your program can be some form of "monitoring loop" in which the program repeatedly measures the behavior of the child.
10. You might want to poll the status of your process via the `/proc` filesystem. See `proc.c` for one way of parsing the `/proc` data. You are free to use this parser (but caveat emptor; I have not updated it for the new kernel!)
11. You should presume that the child can do anything, including changing its behavior according to all available system calls. You cannot assume that the child won't try to get around your controls via malicious means. In particular, you cannot assume that the child won't block all blockable signals!
12. Note that *you cannot install a signal handler in the child*, because during an exec, that handler would get overwritten!
13. But the child is permitted to install its own signal handlers to thwart your controls, and you must be able to handle that!

Submitting completed assignments

The whole program should be in a single c program `watch.c`. Programs in other languages are unacceptable. The beginning of the file should describe how to compile the file, in comments. A typical compilation command might be:

```
gcc -g -std=c99 -o watch watch.c -lpthread -lrt
```

This allows use of POSIX threads library (-lpthread) and the real-time high-res clock library (-lrt).

To submit this program, first ssh to `comp111-01` (or `-02` to `-06`), then type:

```
provide comp111 a3 watch.c
```

where `watch.c` is a file containing your program.

If you use shadow libraries, please instead type:

```
provide comp111 a3 watch.c shadow.c
```

where `shadow.c` is the code for your shadow program.

You may add `Makefile`, `shadow.so`, etc. if you wish.

Your submission will be graded offline. To see grading status or comments, type:

```
progress comp111
```

Checklist

	watch.c is a self-contained C program.
	Your shadow code (if any) is contained in `shadow.c`, which is also a self-contained C program
	The method for compiling `watch.c` is documented in a comment at the top of the program.
	The method for compiling `shadow.c` is documented in a comment at the top of the program.
	Your program, when compiled into the file "watch", takes one argument that is the name of a compiled child program. Typing watch `./a.out` will watch whatever process results from executing `a.out`. *It is not acceptable to make us change your program to test each case.*
	At the top of your `watch.c` program, in a block comment, you have specified how you handled each condition, and whether there are cases that your solution does not catch, and why.
	Case 1: limit size of stack.
	Case 2: limit size of heap.
	Case 3: prohibit forking.
	Case 4: prohibit thread creation.
	Case 5: prohibit opening files.
	(extra credit) Case 6: Limit size of global variables.