Lab Worksheet 03 - Signals

Objectives

1. Enter the wonderful world of inter-process communications.
2. Program basic signal exchanges among processes.
3. Redefine signal handlers within programs.

Remarks

- The cchars section in the output of command line `$stty -a` provides the list of shortcuts for sending a signal to a process from your terminal. They are system-dependent, but most OSes use `ctrl-C` for `SIGINT` and `ctrl-\` for `SIGQUIT`.
- The masking of signals often results in programs that one does not simply interrupt with a `SIGINT`. The last resort is to use `SIGKILL` from a terminal: `$kill -9 <pid_victim>`

Exercise 1: First tests with signals

Start by downloading this archive (https://newclasses.nyu.edu/access/content/group/51ce8755-5381-4dd5-bae5-8ae3b3c862d0/Worksheets/Skeleton-Code/init-sigs.tgz) that contains source codes for various short programs.
Compile and test them to see:

1. if they end by themselves, or if they need to be unlocked using one (or more) `SIGINT`
2. if all the display calls they contain get carried out.

Briefly explain the behavior of each of these mini-programs.

Exercise 2: Zombie vs. SIGKILL

Compile the following code (https://newclasses.nyu.edu/access/content/group/51ce8755-5381-4dd5-bae5-8ae3b3c862d0/Worksheets/Skeleton-Code/voodoo.c). This program creates a zombie (a child process whose parent never acknowledges its termination -- sad, I know).
Observe the output of command `$ps u`
Try to force the termination of the zombie process with a `SIGKILL`. What is going on ?

Exercise 3: Pending signals

Write a program called `pending-signals` that displays whether it has received signals but has yet to deliver them. This program works as follows: it masks `SIGINT` and `SIGQUIT`, then sleeps for 5 seconds using the sleep function. Upon awakening, the program should display whether any or both of these signals are pending.
What happens if the program unmasks `SIGINT` and `SIGQUIT` before displaying whether they are pending? Explain why.

Exercise 4: Changing the default behavior (SIG_IGN)

Write a modification of the previous program called `pending-signals-2`, where the behavior is redefined so as to ignore a `SIGINT` or a `SIGQUIT`.
What happens *now* if the program unmasks `SIGINT` and `SIGQUIT` before displaying whether they are pending? Explain why.

Exercise 5: Changing the default behavior (new routine)

Write a modification of the previous program called `pending-signals-3`, where the behavior is redefined so as to increment a counter and display the value of this counter when receiving a `SIGINT`.
What is the maximum value that the counter can reach? Will it be reached upon every execution? Justify your answer.
What happens *now* if the program unmasks `SIGINT` and `SIGQUIT` before displaying whether they are pending? Explain why.

Exercise 6: Signal Recognition

Write a program called `accountant` that keeps count of the number of signals it delivers.
Once launched, accountant waits for signals in a loop. For each delivered signal, it increments two counters: a global counter which sums up all signals, and a counter associated with the value of the delivered signal.
The program ends once it has delivered a number `MAX_INTR` of `SIGINT`. It displays all of its statistics: the value of each of its value-specific counters, as well as the value of its global counter.

Exercise 7: Buffering shell commands

Write a program called `command-salvo` that acts as a buffering layer for your terminal.
Once started, `command-salvo` sits on top of the shell program and awaits commands entered by the user (you can reuse the skeleton code " (https://newclasses.nyu.edu/access/content/group/51ce8755-5381-4dd5-bae5-8ae3b3c862d0/Worksheets/Skeleton-Code/spy-incomplete.c)`spy-incomplete.c` (https://newclasses.nyu.edu/access/content/group/51ce8755-5381-4dd5-bae5-8ae3b3c862d0/Worksheets/Skeleton-Code/spy-incomplete.c)" (https://newclasses.nyu.edu/access/content/group/51ce8755-5381-4dd5-bae5-8ae3b3c862d0/Worksheets/Skeleton-Code/spy-incomplete.c) for this purpose). Upon every user command, `command-salvo` prepares its execution via an `execvp` call performed in a child process. However, a child does not execute the command immediately after being created. Instead, it waits for a signal from its parent. The parent process waits until `NBUF` user commands have been entered, and then notifies all its children that they can execute their respective command comcurrently. After launching a command salvo, the parent waits for the completion of all its children before preparing the next salvo.