Exercise 4 : Process Pairs

A process pair is a program that acts as its own supervisor, by creating its own backup (a duplicate of itself) in case it crashes.

This is done to handle all crashes in the same way: restart. But we also don't want to lose all the work we have done, which means we have to periodically transfer the last known "good" data to the backup.

The process pair mechanism on its own forms one of the three parts of making bug-repellent software, that can handle both known and unknown bugs:

- A robust self-diagnosing integrity check
  Also known as "acceptance test", which aims to detect success instead of detecting failure
- A self-terminating mechanism
  To provoke the backup into taking over
- An auto-restarting mechanism
  Where process pairs are one solution

Some engineering questions, before you continue:

- Why would we aim to detect success in results, instead of the more classical way of detecting errors/failures?
- Why would we want to self-terminate, instead of handling the error immediately in the "primary" (as opposed to deferring it to the backup as it "spins up")?
- Is there any reason to prefer a process pair style, as opposed to making a separate supervisor-like program whose sole purpose is to restart the main program?

Create a program (in any language, on any OS) that uses the process pair technique to print the numbers 1, 2, 3, 4, etc. to a terminal window. The program should create its own backup: When the primary is running, only the primary should keep counting, and the backup should do nothing. When the primary dies, the backup should become the new primary, create its own new backup, and keep counting where the dead one left off. Make sure that no numbers are skipped!

View a demo of process pairs in action (YouTube) (https://youtu.be/HCgj9pqrTW4)

You cannot rely on the primary telling the backup when it has died (because it would have to be dead first ...). Instead, have the primary broadcast that it is still alive, and have the backup become the primary when a certain number of messages have been missed.

You will need some form of communication between the primary and the backup. Some examples are (roughly in order of ease-of-use):

- Files: The primary writes to a file, and the backup reads it.
  - Either the time-stamp of the file or the contents can be used to detect if the primary is still alive.
- Network: The easiest is to use UDP on `localhost`. TCP is also possible but may be harder (since both endpoints need to be alive).

    - Use the ability to set a "read deadline" on the socket. You will either get a message (from the primary), or an error (indicating a timeout).
    - Remember to close the socket you are reading from (while in the backup phase) before spawning your backup, so the next (newly created) backup can re-use the socket's port.
- IPC, such as POSIX message queues: see `msgget()` `msgsnd()` and `msgrcv()` (http://pubs.opengroup.org/onlinepubs/7990989775/xsh/sysmsg.h.html). With these, you can create FIFO message queues.
- Signals (http://pubs.opengroup.org/onlinepubs/7990989775/xsh/signal.h.html): Use signals to interrupt other processes (You are already familiar with some of these, such as SIGSEGV (Segfault) and SIGTERM (Ctrl+C)). There are two custom signals you can use: SIGUSR1 and SIGUSR2. See `signal()`.
    - Note for D programmers: SIGUSR is used by the GC. (http://dlang.org/phobos/core_memory.html)
- Controlled shared memory: The system functions `shmget()` and `shmat()` (http://pubs.opengroup.org/onlinepubs/7990989775/xsh/sysshm.h.html) let processes share memory.

You will also need to spawn the backup somehow. There should be a way to spawn processes or run shell commands in the standard library of your language of choice. The name of the terminal window is OS-dependent:

- Ubuntu: `"gnome-terminal -- commands "`.
  Example: `"gnome-terminal -- ./pheonix"`.
- Windows: The native shell on Windows is `cmd.exe`.
    - If you want to use `cmd`: Use `start program_name` . Example: `"start pheonix.exe"`, from whatever function spawns an OS process.
      You may need to use `"start \"title\" call args "` for more complex commands.
    - If you want to use `powershell`: Use `start powershell args` .
      Example (Dlang): `executeShell("start powershell rdmd pheonix.d");`.
- OSX: `"osascript -e 'tell app \"Terminal\" to do script \"terminal commands \"'"`.
  Note the quotes around `"Terminal"` and your commands. Be sure to escape these appropriately.

Some OS- or language-specific tips:

- Linux: You can prevent a spawned terminal window from automatically closing by going to:
    - Menu (the hamburger icon)
    - Preferences
    - Profile (by default the "Unnamed" profile)
    - When command exits
    - -> Hold the terminal open
- Golang: `os.exec.Command` takes its arguments oddly (https://golang.org/pkg/os/exec/#Command), so you will have to separate out the program and its arguments. And use backticks for raw strings (https://golang.org/ref/spec#String_literals).
  Example (OSX): `exec.Command("osascript", "-e", `tell app "Terminal" to do script "go run ` + filename + `.go"`).Run()`
  Example (Linux): `exec.Command("gnome-terminal", "--", "go", "run", "filename.go").Run()`

- Windows: If you get the error `"start": executable file not found in %PATH%`, try inserting `"cmd /C start ...."`.
  This seems to mostly apply to Go, so here's the full command to start your program in a PowerShell window:
  ```
  err := exec.Command("cmd", "/C", "start", "powershell", "go", "run", "filename.go").Run()
  ```

Be careful! You don't want to create a chain reaction (http://en.wikipedia.org/wiki/Fork_bomb) ... If you do, you can use `pkill -f program_name` (Windows: `taskkill /F /IM program_name /T`) as a sledgehammer. Start with long periods and timeouts, so you have time to react if something goes wrong ... And remember: It's not murder if it's robots.

Try to think about the order in which you make things, so you don't have to get everything right at the same time. It might be useful to start by making two programs:

- `A`: Make a program that just terminates after a few seconds
- `B`: Make a program that starts program `A` in a separate window
- `B`: After starting `A`, send/write some data periodically
- `A`: Instead of just waiting, read data that `B` creates
- `A`: Add some kind of timeout to detect when `B` dies
- Combine into one program

Make sure you don't over-complicate things. This program should be two loops (one to see if a primary exists, and one to do the work of the primary) with a call to create a new backup between them. You should not need any extra threads.

In case you want to use this in the project: Usually, a program crashes for a reason. Restoring the program to the same state as it died in may cause it to crash in exactly the same way, all over again. How would you prevent this from happening?