L2: The Entire Nation Builds Its Capital

January 19, 1971. The Political Bureau of the Central Committee of the Polish United Workers' Party decides to rebuild the Royal Castle in Warsaw, completely destroyed by the Germans during the Warsaw Uprising. Convinced that the "symbol of Polish sovereignty" should be rebuilt by Poles themselves, it was decided that the reconstruction would be financed by a nationwide collection.

The collection was located, of course, on Castle Square, at the foot of the destroyed monument. There was no need to even announce a relevant message in the media - Warsaw residents, but also many visitors, spontaneously, enthusiastically, and in large numbers, rushed to support this noble cause. Due to such great interest, several collection boxes were set up in the square and a highly innovative solution was adopted - the collection would be coordinated by an electronic system!

Delegated Party members for the collection are looking for a programmer to create the necessary software. The choice fell on the technical university closest to the Castle. From the very entrance, the delegates were greeted by a cheering crowd of students ready to undertake this extremely responsible task. At some point, you notice that the gaze of one of the visitors falls on you. You know exactly what that means.

> all file operations should be low-level.

Stages

Stage 1 (5 points)

The program starts with one argument $1 \leq n \leq 4$, specifying the number of collection boxes set up in Castle Square. The launched program will act as the collection coordinator. Create n child processes representing the collection boxes. Store the PIDs of

these processes in any data structure - they will be useful later. Each collection box process prints `[<PID>] Collection box opened`, creates a file named `skarbona_<PID>`, immediately closes it, and then terminates. After all child processes have terminated, the coordinator process prints `Collection ended!`, then releases all resources and terminates.

Remember to check the correctness of the passed arguments.

If the `skarbona_<PID>` file already exists for a given PID, it should be overwritten.

> *Hint:* to avoid clutter with many collection box files, you can use the `make tidy` command to remove them.

Stage 2 (6 points)

After creating the collection box processes, the coordinator process `ITER_COUNT` times creates a child process representing a new donor. After creating the donor process, the coordinator waits 100ms, and then sends a random signal from the set `{SIGUSR1, SIGUSR2, SIGPIPE, SIGINT}` to the donor. Each donor process should block the listed signals, and then wait for them. The value of the received signal will be used to read the assigned collection box number to which the donor should go - in the listed order from 0 to 3. The donor process prints the collection box number in the message `[<PID>] Directed to collection box no <nr>`, and then terminates.

The coordinator, wanting to perform its work diligently, before handling the next donor, waits for the current one to go to the collection box, i.e., waits for the process to finish before starting the next one.

Stage 3 (8 points)

Implement the process of making a donation to the collection box. If the collection box number is greater than or equal to n, the donor process prints `[<PID>] Nothing here, I'm going home!`, and then terminates. Otherwise, the donor process opens the `skarbona_<PID_S>` file in **append** mode, where `<PID_S>` is the PID of the collection box process with the number received from the coordinator. Exactly two four-byte integers are appended to the file: the first is the donor's PID, and the second is the donated amount in PLN - a random number from the range `[100, 2000]`. After writing to the file, the donor prints `[<PID>] I'm throwing in <x> PLN` and notifies the collection box process about the donation with the `SIGUSR1` signal. Then it terminates.

The collection box process, after creating its file, now blocks `SIGUSR1`, and then waits for

it in an infinite loop. After receiving this signal, it reads the newly appended content of its file and for the read data prints `[<PID>] Citizen <PID_D> threw in <X> PLN. Thank you! Total collected: <S> PLN`, where `<PID_D>` is the donor's PID, `<X>` is the donated amount, and `<S>` is the sum of donations collected so far in that collection box.

The coordinator process, after completing the main loop, sends `SIGTERM` to the collection box processes to terminate their operation. You do not yet need to implement proper handling of this signal in the collection box processes.

> *Hint 1:* To read newly appended data from a file, it may be necessary to move the cursor; the `lseek` function and the `SEEK_END` flag (`man 3p lseek`) will be helpful.

> *Hint 2:* A negative number can also be passed as `offset` to the `lseek` function, which is worth considering.

Stage 4 (3 points)

Replace the finite loop in the coordinator process with an infinite one, and then implement `SIGTERM` signal handling in all processes. Upon receiving this signal, the process sends `SIGTERM` to all processes in its process group, then breaks the loop or waiting for further signals, releases all resources, waits for child processes, and terminates.

> *Hint:* You can send `SIGTERM` to one of the processes with the following command:
>
> ```
> kill -SIGTERM `pgrep sop-collection | head -n1`
> ```
>
> It is also worth trying to replace `head` with `tail` in the given command.

Starting code

- Makefile
- sop-collection.c

Download as zip