Task 'Executor'

Sara develops an algorithm and writes programs that run for a long time, writing out information about progress from time to time. Sara would like to test her programs by running them on different examples and checking what they write. However, there are a lot of examples to check and print information. Prepare an executor for Sarah, which will help by performing the tasks indicated, checking their last words (while the program is running) and killing them.

The executor should handle the following commands (each command is one line of input):

Run command

Command **run A B C . . .** creates a new task, starting the execution of program A with arguments B C . . . in the background. Tasks are identified by the numbers 0, 1, 2, . . . , in the order they start (i. e. run commands). The executor prints **Task T started: pid P. \n**, where T is the task ID and P is the pid of the process executing A. Anything A prints to standard and error output should not be shown. Program A will never expect standard input (you can leave it unchanged).

Command out

The **out T** command prints **Task T stdout: 'S'. \n**, where S is the last line (without a line break) previously printed by program A from problem T. In a pointless solution, the executor can wait for program A to print the next line (or EOF) and use it as S. In a pointless solution, the executor should print the last line immediately (unless program A prints anything else for a long time; if program A soon prints lines, you can also use any of them). Programme A should continue.

If the program has not printed anything yet, you can use an empty S. If the program has finished, S should be the last line printed by the whole program.

err command

The **err T** command prints **Task T stderr: 'S'. \n**, similar to out T, but for A's standard error output. It prints this to the standard executor output (like all executor commands).

The kill command

The **kill T** command sends a SIGINT signal to program A from job No. T. If the program has already finished, you can either send a signal (unsuccessfully, ignoring the error) or do nothing.

Auxiliary commands

sleep N – simply sleeps the executor for N milliseconds (using e. g. usleep, which uses microseconds), where N is an integer. Pauses processing further commands for the time being; without pausing any tasks.
quit or end of input (EOF) – exits the executor (and all programs).
empty line – does nothing, goes to the next command.

Completion of tasks
When the program in task no T ends, the executor should inform you by printing **Task T ended: status X. \n**, where T is the job ID and X is its exit code. If the exit code is not available (because the task was interrupted by a signal or by the system), print **Task T ended: signalled\n** instead.

If the program exits while the executor is handling a command, the information should not be logged out until the execution is finished. In the solution for fewer points, you can wait until the next command (and its completion). In the full number of points solution, the information should be unsubscribed immediately after the end of the program, i. e. while the executor is waiting for the next commands (unless the command is being handled).

When the executor announces the completion of the last (of the already started) task, there should be no process or thread created by the executor, except the main executor thread itself and possibly one auxiliary thread/process.

Example
Made in Bash

echo -e "foo\nbar" > in. txt;
echo -e "run cat in. txt\nsleep 100\nout 0" | . /executor
should write for example:

Task 0 started: pid 1234.
Task 0 ended: status 0.
Task 0 stdout: 'bar'.

Comments
The memory consumption of the executor should be independent, O(1), of the output length and runtime of the executed programs.
The executor cannot create ordinary files.
The executor should not start accepting the next command on the input until the previous one is finished.
It is forbidden to use poll() and variants (ppoll, select, pselect, epoll).
In addition to the functions discussed in the class, you can also use other C or POSIX functions, except poll(). None are necessary, waitpid (with the WNOHANG option to check if someone is finished) and fdopen (to be able to use functions operating on streams, e. g. getline or fgets).

Active waiting is forbidden; more precisely, if for a long time no more commands appear on the input and programs do not print anything, you cannot consume the CPU during this time. In particular, you can not periodically wake up the process.
Running programs should not have any open descriptors other than the standard ones.
To minimize memory difficulties and errors in C:

All executor commands will be correct and no longer than 511 characters (including the end of the line).
All lines printed by executing programs will be no longer than 1022 characters (including the end of the line; the last line may or may not contain the end of the line).
Total quests will be at most 4096. The executor can use memory proportional to that number, multiplied by the maximum line length, even if we do not run as many tasks. For example, you can have a global array struct Task tasks[MAX_N_TASKS];.
Executed programs will never print a zero sign. Executor commands will never contain a zero character, quotation marks, or \. Specifically, program arguments are simply space-separated and can be processed using the split_string function of utils. h.
In the event of any error, you can simply terminate the executor, e. g. exit(1).
You can print any diagnostic information, in particular about errors, on stderr.