A basic process scheduler example for Linux

Written by Felix Stegmaier

0. tl;dr

This is a simple process scheduler for Linux that executes multiple processes pseudo-simultaneously using a round robin algorithm.

Build with project by executing `make` in the main folder.
Start the example program by executing `main`.

1. Requirement

This project is part of my Operating Systems I lecture.
The literal exercise was to write a "basic process scheduler."
What was meant by this is that I should write a program that demonstrates how to create processes and schedule them, so that only one process is active at a time and the processes still appear as to be pseudo-parallel, i.e. they run in alternation.

2. Specification

The scheduler is a piece of software that has the capability to create processes and controll their execution. It is written as a C library that makes use of the Linux/GNU system calls to fork of a new child process and to issue signals to other processes and to respond to system signals. It is

runnable under GNU Linux.

At each time only one process is running (see above). The processes that are active (i.e. they want to run) are stored in queue. The scheduler uses a round robin algorithm to give each process that wants to run the same amount of processing time, called the quantum. This quantum shall be 10 milliseconds, so that it can be reasonable easy be seen that the scheduler does its work when the processes it creates take longer than that time to execute. When this amount time is up, the scheduler pauses the current process, enqueues it and make the next process in the queue active.

It offers a similar functionality for processes like pthread offers for threads. Especially those are:

- Creating new processes with a given function reference
- Joining a running process, i.e. wait for the process to finish
- Pausing and continuing a process
- Killing a process, i.e. terminating it

3. Implementation of features

Explanation: A **signal** is a kind of interupt on operating system level, that can be trigerred by various events and is than send to a process. This process can register an signal handler routine that is called, every time a specific signal is send to it. This signals can be keyboard interrupts, termination of an process, a timer and many more. If a process does not register a signal handler, the deafault action is taken, i.e. to terminate the process that does not handle the signal. (This is why some programs terminate with Ctrl-C and some don't.)
See https://de.wikipedia.org/wiki/Signal_(Unix) for further details and all the available signals.

The scheduler registers two signal handlers, one on a timer (SIGALRM) and one to find out when a (child) process it controlls terminates (SIGCHLD).

The SIGALRM handler is called by a signal every quantum amount of time. In that case the currently running process has used up is current time slot and the next one may run. It is used to send an pause signal (SIGSTOP) to the active process, enqueue it at the end of the queue for the active processes and pop the next process out of that queue by sending it a signal to continue its work (SIGCONT).

The SIGCHLD handler is called, whenever a child process, which is handeled by the scheduler, has finished it work and has terminated. The child process my not be made active again after this point. Therefore it must be erased from the queue of active (and paused if applicable, see below) processes.

To **create a process** the scheduler forks itself and lets the given function reference execute. This process will be halted imideatly and enqued into the active process queue, so that, when the time comes, the new process will be made active by the scheduler.

To **join a process** means to wait blocking for it to finish its work. To do so the scheduler uses a systemcall *waitpid* to wait blocking until the child terminates.

To **pause a process** the scheduler issuses a signal to pause the given process, takes it out of the queue of active processes and put it into another queue used to hanlde currently paused processes.

To **continue a process** after it has been paused, the process is taken out of the queue of paused processes and enqueued at the end of the queue of active processes. When the scheduler goes through the queue of active

processes, it will finally make that process be running for the a quantum amount of time.

To **kill a process**, i.e. terminating it immideatly, the scheduler sends a termination signal to that process and removes it from the queues of active and paused processes, since it can not be resumed.

3. Testing

The program should include several test cases. These test cases create multiple "tasks" as processes to be scheduled by the scheduler and perform different actions on them (e.g. joining, pausing and killing in various orders). Each of these tasks runs in pseudo-parallel and counts up a variable which it outputs to the console.

1. Create 3 processes and join with them.
2. Create 3 processes, pause one of them, join with the other two, continue the former and join it.
3. Create 3 processes, pause one, kill another one, continue the former and wait for all living processes to finish.
4. Create 3 processes, kill all of them.

The test cases are chosen, so that all functionalities in various orders are covered.

4. Running

Build with project by executing `make` in the main folder.
Start the example program by executing `build/main`.

All the tests specified in 3. will be executed now and you will see the output

of the tests.

Clean the projects `build` folder by running `make clean`.

5. Conclusion

This scheduler implementation makes use of the underlying operating system and its capabilities to create processes via fork and to control them using signals. Furthermore is takes advantage of the capabillity of the system to backup and restore the current state of a process (e.g. stack and instruction pointer). Therefore it still uses rather high level system calls (in contrast to implementing this features in Assembler).

Overall it shows how to use this system calls to schedule processes.