Individual project report

DTEK 2085-3002

October 23, 2024

Kalle Sova

Inter-process communictaion using shared memory and pipe

This project implements inter-process communication (IPC) between two main processes, **init** and **Scheduler**, using shared memory and pipes in Python. The project involves creating four child processes (P1-P4) that communicate with the **init** process through pipes, passing randomly generated numbers. The **Scheduler** process manages a shared memory segment where **init** writes the random numbers, and **Scheduler** sorts and outputs them. Upon completion, the shared memory is detached and unlinked to ensure proper resource cleanup.

Overview

1. init Process

The **init** process is the parent process that performs the following functions:

- It forks four child processes (**P1-P4**) using the os.fork() system call.
- **P1-P4** generate a random integer between 0-19 and send these numbers to **init** via pipes.
- **init** receives the numbers from each child process through the pipe's read end.
- After receiving the numbers, **init** attaches to a shared memory segment created by the **Scheduler** process and writes the received numbers into it.
- Once all data is written to shared memory, **init** detaches from the shared memory.

2. Child Processes (P1-P4)

Each of the four child processes (**P1-P4**) executes the following:

- A random integer between 0-19 is generated using Python's random.randint() function.
- The generated number is written to the parent process (**init**) via the pipe's write end.
- After sending the data, each child process terminates.

3. Scheduler Process

The **Scheduler** process operates independently and performs the following:

- **Scheduler** creates a shared memory segment using multiprocessing.shared_memory.SharedMemory with a size sufficient to hold four integers.
- It sends the shared memory segment's name to **init** via a pipe, allowing **init** to attach to it.
- After receiving the random numbers written by **init** into the shared memory, **Scheduler** reads them, sorts them in ascending order, and prints the sorted list.
- Finally, **Scheduler** detaches from the shared memory, and the shared memory segment is unlinked (deleted) to free system resources.

Conclusion

This project demonstrates the use of IPC mechanisms, including pipes and shared memory, to enable communication and data sharing between processes. The system ensures proper synchronization and resource cleanup using Python's os and multiprocessing.shared_memory libraries. The workflow successfully captures the essence of process communication and scheduling policies as modeled in real operating systems.