COMPSCI 3SH3 Winter, 2021
Student Name: Jatin Chowdhary
Mac ID: Chowdhaj
Student #: 400033011
Date: February 12th, 2021

Lab 2 Report

Q1 – time-shm

Module Description

The *time-shm* program demonstrates interprocess communication (IPC) using shared memory. The program starts off by setting up a shared memory object, called file descriptor. Then, it forks the parent process and creates a child process. An *if-else* conditional is used to check whether the fork was successful, execute the child process' code, and execute the parent's code. The child process maps memory to the shared memory object, and sets the size, permissions, etc. The child process gets the *timeval* struct and stores/prints the microseconds time in the shared memory. Subsequently, the size of the pointer to the shared memory is updated. Finally, the child process executes the command that is part of the main function's arguments. Once the child process finishes, the parent process terminates the child process, and records the current epoch time in microseconds. At this point, only the parent process is running. Previously, both processes were running, but the parent process was waiting for the child process to finish. Now, the parent process sets up a file descriptor (FD) for the shared memory object. Then, it uses the FD and maps to the shared memory object, and sets the size, permission, etc. Once this finishes, the shared memory is read, the data is copied to the parent's local variables and converted into an operable type. The finish time is subtracted from the start time and the difference is printed to the terminal. Note that the result is in microseconds. Last, but not least, the parent process removes the shared memory object, and terminates. For more information – like functions, names, variables, etc. – please reference the source code and review its comments.

The parent and child process in the *time-shm* program use shared memory to communicate. In the shared memory model, interprocess communication (IPC) is done by establishing a region of memory that is shared by cooperating processes.

The advantages in using shared memory for IPC are:
=> Easy to setup; the kernel is required, once, to setup the shared memory region.
=> Speed; once the shared memory region is established, no more kernel interventions are required. All further read and writes are routine memory accesses.
=>Performance; processes can simultaneously read (from) or write (to) the shared memory region. This helps improve performance because two processes can work on the same task and share results.
=> Scalability; multiple processes can share the same region of memory.

The disadvantages in using shared memory for IPC are:
=> Concurrency; the processes need to be synchronized so they don't write to memory while the other process reads from memory, and vice versa.
=> Overwriting; the processes are responsible for ensuring that they are not writing to the same location, simultaneously.
=> Deadlock; depending on how the concurrency issues are addressed, a poor implementation can lead to deadlock. For instance, both child and parent want to write to the same memory location, but the parent process does not release the "lock" on the memory, so the child process is waiting indefinitely.

***Source Code:** time-shm.c, Makefile

Q2 – time-pipe

Module Description

The *time-pipe* program demonstrates interprocess communication (IPC) using pipes. The program starts off by declaring variables, one of which is a file descriptor (FD). The FD is used to create a pipe. Then, a child process is created using *fork()*, and the return value is stored in a variable. An *if-else* statement uses this value to check whether the fork was successful, execute the child process' code, and execute the parent's code. The child process starts off by closing the read end of the pipe. Then, it calls the *gettimeofday()* function, which copies Epoch time values into a local struct. Next, the write end of the pipe is used, and a pointer to the struct, from earlier, is written into the pipe. Once this process has completed, the write end of the pipe is closed, and the child process executes the command in the main function's arguments. The child process has finished, and is terminated by the parent process. At this point, only the parent process is operating. The parent process starts by recording the current Epoch times. These are stored in a *timeval* struct, and will be used later. Then, the parent process closes the write end of the pipe. Next, it reads the other end of the pipe and copies the data, which is a pointer to a memory address. Once the copy is complete, the read end of the pipe is closed. At this point all the pipes, read and write, between parent and child are closed. Finally, the two (microsecond) time values are subtracted – start time subtracted from finish time – and the result is printed to the terminal. Last, but not least, the program terminates. For more information – like functions, names, variables, etc. – please reference the source code and review its comments.

The parent and child process in the *time-pipe* program use an ordinary pipe to communicate.

The advantages are:
=> Flexibility; there are two types of pipes – bidirectional and unidirectional. You can use either depending on your needs. If you only want data to be sent from the child process to the parent process, then unidirectional is the best option. Alternatively, if you need the child and parent process to be able to send information to each other, then bidirectional is the best option.
=> Concurrency; since each process has its own write end of the pipe, there are no concurrency issues that can negatively affect the processes. For instance, processes cannot write to the same block of memory, or a process cannot write to memory while another process reads from it.
=> Easy to use; as a programmer, a lot of the details are hidden, and you do not have to specify them (i.e. Buffer size, file permissions, pointers, etc.)

The disadvantages are:
=> Speed; reading (from) and writing (to) the pipe requires system calls. This can slow down the process if there are multiple reads and writes, because kernel intervention is required every time.
=> Scope; a parent-child relationship is required for an ordinary pipe. This heavily limits the use of this interprocess communication implementation.
=> Persistence; once the processes have terminated, the pipe is closed
=> Bandwidth; data can be half duplex or full duplex depending on the type of the communication

***Source Code:** time-pipe.c, Makefile

**Source Code is attached in a zip file*