Project Report #2

Message Queue: Hot Potato Between Processes

Sujeet Ojha
sxo160430@utdallas.edu
February 26, 2020

I. Problem Statement

The assigned task was to utilize POSIX message queues, fork(), and pipe() in order to play hot potato between multiple processes by passing messages. The task stated that the parent process must send the first message onto the message queue while the child processes were to read from the message queue and enqueue a message onto the queue once a message was read. This method of “hot potato passing” must be performed until each child receives a thousand messages. When the last message is received, the process must send pipe() information in order to alert the parent that it has lost. The parent, upon receiving this notice, enqueues another message onto the message queue in order to ensure that the queue is not empty. This sequence of events must continue until the last process is terminated, which is announced the winner of the game. The key goal of this task is to analyze how the operating system handles message passing.

II. Importance of used functions

The function fork() was utilized in order to create five child processes with their own independent address spaces and a copy of the parent process’ memory segments. POSIX message queues were used to ensure that a means of communication is established between all processes. As POSIX message queues exist independently of the processes that use them, the use of message queues to pass the “hot potato” ensures that the termination of one process does not terminate the message queue. The function pipe() was used to establish a one way communication between the child and the parent in order to ensure that the child is able to announce to the parent whenever it receives a thousand messages.

III. Approach

The programming language selected for this project is C. The approach taken was to first set up an anonymous pipe and a message queue which can be used for communication and message passing. The initial setup requires the message queue to be opened in read/write mode as every child process needs to not only receive messages from the queue, but also append to the queue as required. Once the initial setup was complete, five distinct child processes were created to partake in the game of “hot potato.” Messages were then sent and received in correspondence to the problem statement and the winner of the game was announced. In order to better visualize the way the operating system handles message passing, all key events in the program such as messages being sent/received, pipe communication, and wins and losses were displayed in the terminal. This display of messages was later used to analyze the sequence of events taken along with the priorities the system sets for sending and receiving messages.

IV. Challenges

The challenges present in this problem dealt with handling child processes. As different systems execute tasks in a distinct manner, one of the key challenges was determining if a child process should be terminated once it has completed its given tasks. Terminating child processes at the end, however, resulted in nondeterministic errors. One of the main errors was that the child process would exit before pipe information was sent to the parent, resulting in the program crashing. Rigorous testing determined that in order to avoid this error, the exit() function should not be utilized to terminate child processes.

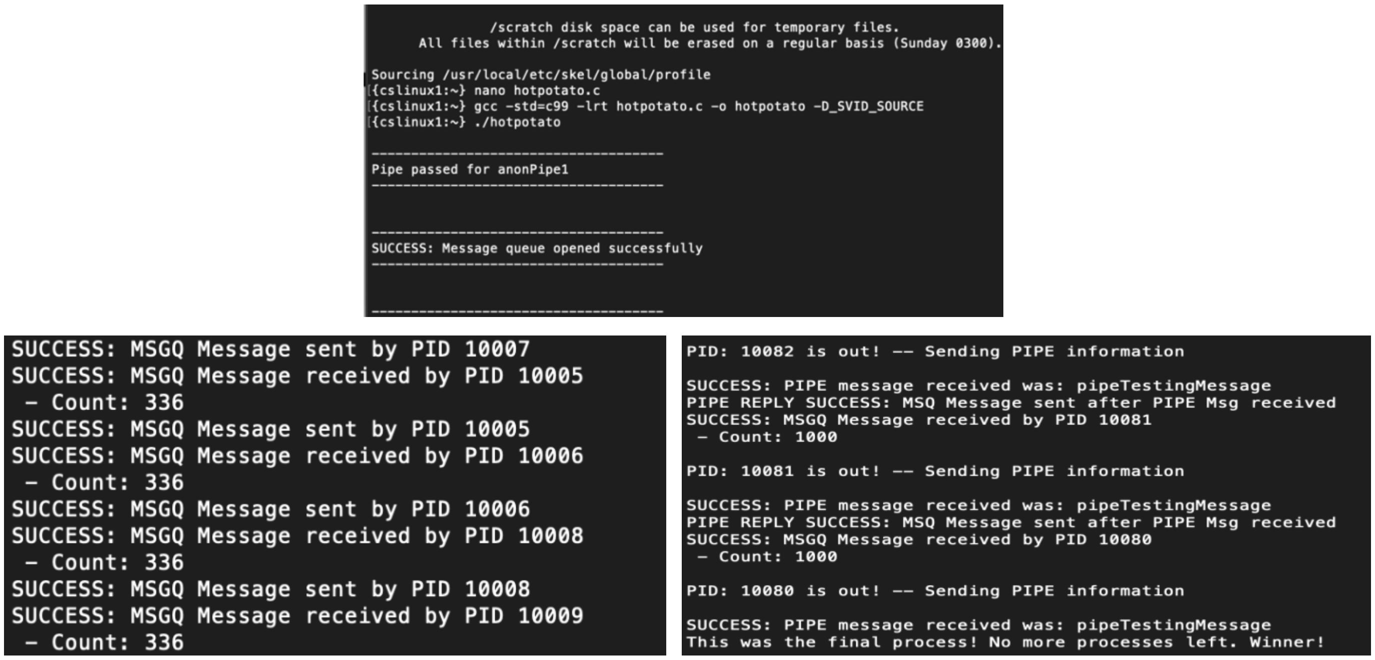
V. Solution

In order to ensure that accurate determinations are made regarding the way the specific way message passing is handled in a specific operating system, the code was executed multiple times. Information regarding important tasks along with the PID of the process executing the task was displayed during every iteration. A file (terminationOrder.txt) was used to keep track of losses for all processes. This file was also used to analyze if the order in which processes are created correlates to the order of which process receives a thousand messages first. In order to avoid preventable errors and get accurate representations, multiple error-handlers were utilized. The message queue used for passing messages was also unlinked at the end in order to ensure that the queue itself is destroyed and any process that has the queue open discards descriptors referring to the message queue.

The following analysis is based on the implementation where the parent process does not wait for child processes to terminate before processing pipe information.

Figures:

- *Plot (1):* Plot (1) depicts the build and execution of the program in the CS1 Linux server.
- *Plot (2):* Plot (2) illustrates the text file (terminationOrder.txt) created by the program.
- *Plot (3):* Plot (3) depicts the important tasks performed by the program.



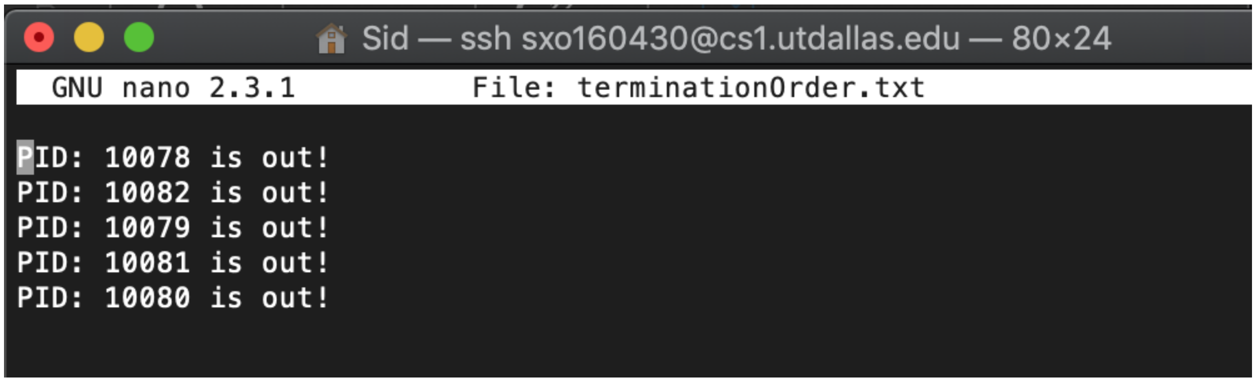
```
            /scratch disk space can be used for temporary files.
     All files within /scratch will be erased on a regular basis (Sunday 0300).

Sourcing /usr/local/etc/skel/global/profile
{cslinux1:~} nano hotpotato.c
{cslinux1:~} gcc -std=c99 -lrt hotpotato.c -o hotpotato -D_SVID_SOURCE
{cslinux1:~} ./hotpotato

-------------------------------------
Pipe passed for anonPipe1
-------------------------------------


-------------------------------------
SUCCESS: Message queue opened successfully
-------------------------------------


-------------------------------------
```

```
SUCCESS: MSGQ Message sent by PID 10007
SUCCESS: MSGQ Message received by PID 10005
 - Count: 336
SUCCESS: MSGQ Message sent by PID 10005
SUCCESS: MSGQ Message received by PID 10006
 - Count: 336
SUCCESS: MSGQ Message sent by PID 10006
SUCCESS: MSGQ Message received by PID 10008
 - Count: 336
SUCCESS: MSGQ Message sent by PID 10008
SUCCESS: MSGQ Message received by PID 10009
 - Count: 336
```

```
PID: 10082 is out! -- Sending PIPE information

SUCCESS: PIPE message received was: pipeTestingMessage
PIPE REPLY SUCCESS: MSQ Message sent after PIPE Msg received
SUCCESS: MSGQ Message received by PID 10081
 - Count: 1000

PID: 10081 is out! -- Sending PIPE information

SUCCESS: PIPE message received was: pipeTestingMessage
PIPE REPLY SUCCESS: MSQ Message sent after PIPE Msg received
SUCCESS: MSGQ Message received by PID 10080
 - Count: 1000

PID: 10080 is out! -- Sending PIPE information

SUCCESS: PIPE message received was: pipeTestingMessage
This was the final process! No more processes left. Winner!
```

Plot (1): Build and Execution of the program

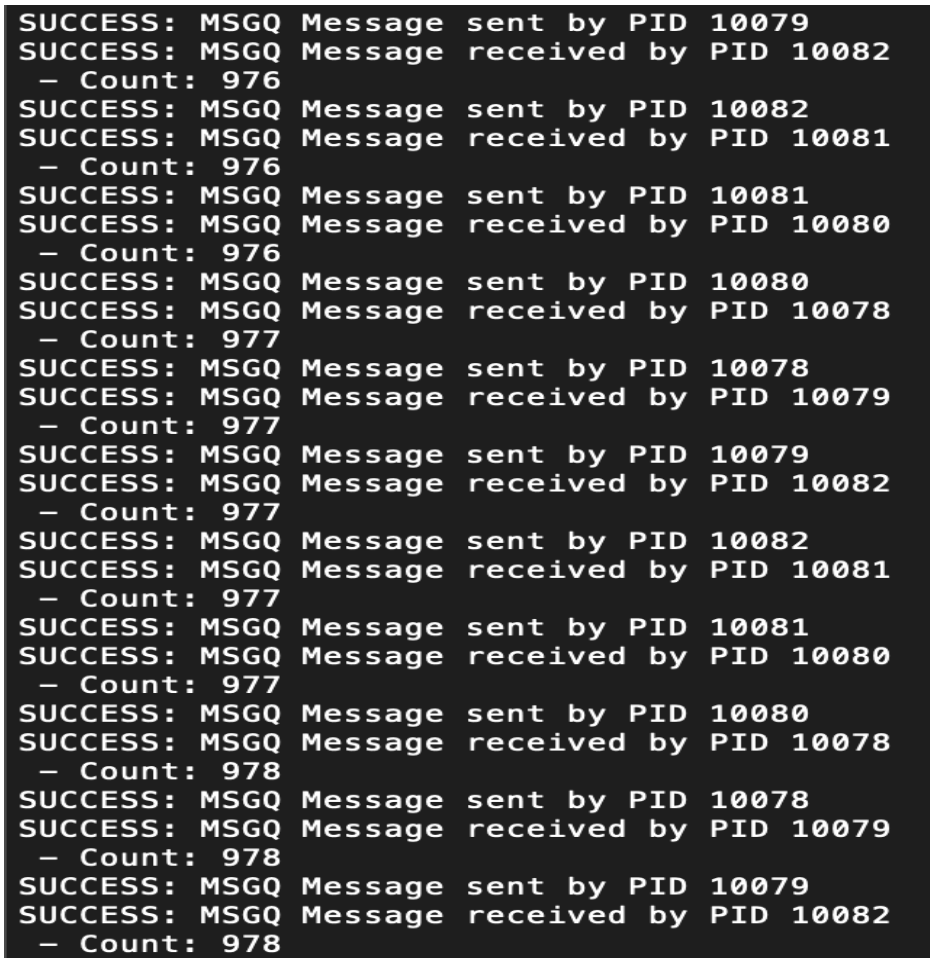
The “cslinux1” server was used to build and execute the code. The term “-D_SVID_SOURCE” is used while compiling the program in order to ensure that definitions for extra functions that are defined in POSIX standards are included. The output shown clearly illustrates the order in which the main events are executed. Additionally, information regarding the losers and the winner of the “hot potato” game, along with their corresponding PID is illustrated.



```
Sid — ssh sxo160430@cs1.utdallas.edu — 80×24
  GNU nano 2.3.1            File: terminationOrder.txt

PID: 10078 is out!
PID: 10082 is out!
PID: 10079 is out!
PID: 10081 is out!
PID: 10080 is out!
```

Plot (2): Text file including the order of termination

The text file illustrated above is generated by the program to store information regarding the losers and the winner of the “hot potato” game. It can be observed that PID: 10078 was the first process to receive a thousand messages. PID: 10080, on the other hand, was the last process to receive a thousand messages, therefore, it is the winner of the game. Analyzing the data, it is clear that the system does not have a specific method that defines the order in which process can access the message queue first. If a sequential method was used where the order of process creation determined the order of access to the message queue, PID: 10078 would be the first loser and PID: 10082 would be the winner. However, as illustrated above, this is not the case.



```
SUCCESS: MSGQ Message sent by PID 10079
SUCCESS: MSGQ Message received by PID 10082
 – Count: 976
SUCCESS: MSGQ Message sent by PID 10082
SUCCESS: MSGQ Message received by PID 10081
 – Count: 976
SUCCESS: MSGQ Message sent by PID 10081
SUCCESS: MSGQ Message received by PID 10080
 – Count: 976
SUCCESS: MSGQ Message sent by PID 10080
SUCCESS: MSGQ Message received by PID 10078
 – Count: 977
SUCCESS: MSGQ Message sent by PID 10078
SUCCESS: MSGQ Message received by PID 10079
 – Count: 977
SUCCESS: MSGQ Message sent by PID 10079
SUCCESS: MSGQ Message received by PID 10082
 – Count: 977
SUCCESS: MSGQ Message sent by PID 10082
SUCCESS: MSGQ Message received by PID 10081
 – Count: 977
SUCCESS: MSGQ Message sent by PID 10081
SUCCESS: MSGQ Message received by PID 10080
 – Count: 977
SUCCESS: MSGQ Message sent by PID 10080
SUCCESS: MSGQ Message received by PID 10078
 – Count: 978
SUCCESS: MSGQ Message sent by PID 10078
SUCCESS: MSGQ Message received by PID 10079
 – Count: 978
SUCCESS: MSGQ Message sent by PID 10079
SUCCESS: MSGQ Message received by PID 10082
 – Count: 978
```

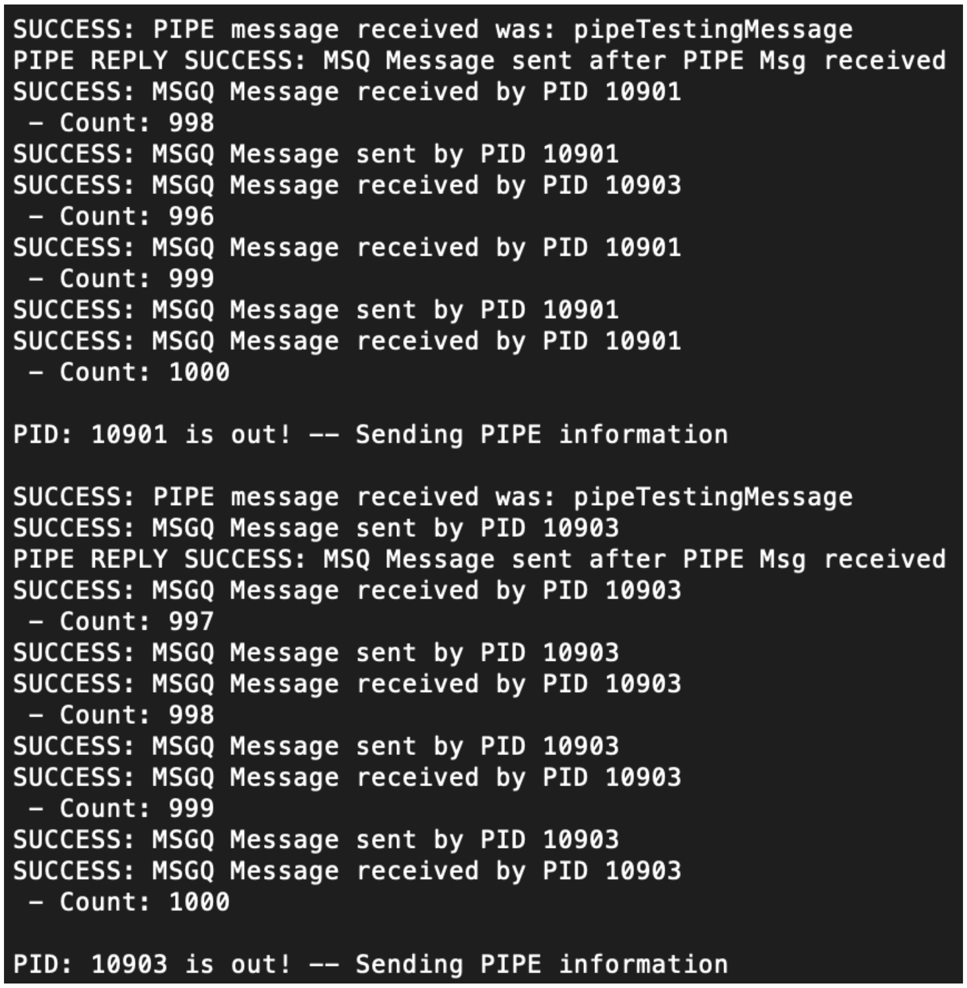
Plot (3): Execution of important tasks

The analysis of the figure above leads to one important observation regarding the execution: a single process cannot "starve" other processes by having continuous access to the message queue. During this specific execution, the system ensured that after a process completes a set of tasks, it has to wait for all other processes to complete the same set of tasks before starting a new iteration. If this was not the case, the "Count" as displayed above, would not increase sequentially. Instead, the "Count" would be displayed in a random fashion as a process could send and receive messages as many, or as little, times as possible before another process accessed the message queue. Although this seems to be a pattern for majority of the executions, there are a few cases where the order is not strictly sequential. These outlier cases conclude that depending on execution conditions, the system can choose to behave differently.

VI. Additional Observations

Figures:

- *Plot (4):* Plot (4) depicts execution of a program where the wait() function is utilized by the parent before processing pipe information.



```
SUCCESS: PIPE message received was: pipeTestingMessage
PIPE REPLY SUCCESS: MSQ Message sent after PIPE Msg received
SUCCESS: MSGQ Message received by PID 10901
 - Count: 998
SUCCESS: MSGQ Message sent by PID 10901
SUCCESS: MSGQ Message received by PID 10903
 - Count: 996
SUCCESS: MSGQ Message received by PID 10901
 - Count: 999
SUCCESS: MSGQ Message sent by PID 10901
SUCCESS: MSGQ Message received by PID 10901
 - Count: 1000

PID: 10901 is out! -- Sending PIPE information

SUCCESS: PIPE message received was: pipeTestingMessage
SUCCESS: MSGQ Message sent by PID 10903
PIPE REPLY SUCCESS: MSQ Message sent after PIPE Msg received
SUCCESS: MSGQ Message received by PID 10903
 - Count: 997
SUCCESS: MSGQ Message sent by PID 10903
SUCCESS: MSGQ Message received by PID 10903
 - Count: 998
SUCCESS: MSGQ Message sent by PID 10903
SUCCESS: MSGQ Message received by PID 10903
 - Count: 999
SUCCESS: MSGQ Message sent by PID 10903
SUCCESS: MSGQ Message received by PID 10903
 - Count: 1000

PID: 10903 is out! -- Sending PIPE information
```

Plot (4): Execution with wait() implemented before pipe processing

The figure above clearly illustrates various differences in the order of executing tasks compared to the method discussed in *Section V.* Adding a wait() before pipe information is processed produces different results than not implementing wait(). As illustrated, the order of tasks is no longer done in a sequential manner where all processes complete the same set of tasks before moving on to a new iteration. This is clearly shown by the discrepancies in "Count." A process can access the message queue in any fashion. This unorderly method of access, however, can result in one process being selfish and occupying the message queue for an extended period of time. Therefore, the analysis of this data concludes that code should be written in a specific, optimized way to ensure that the system responds as effectively and efficiently as possible.