Homework 3: Programming Assignment

WORKING WITH SHARED MEMORY AND PIPES FOR INTER PROCESS COMMUNICATION v. 10.02.25 11:30PM

The objective of this assignment is to get comfortable with Inter Process Communication (IPC) using Shared Memory and Pipes. These approaches are among two of the most dominant mechanisms for doing IPC. Familiarity with shared memory will also help you with some of the advanced concepts that we will cover in process synchronization.

DUE DATE: Oct 16, 2025 11 PM
Extended Due Date with 10% per day penalty until Oct 18, 2025 11 PM

1 Description of Task

This assignment builds on the concepts of HW2. In this assignment, however, you will be tasked with creating processes that execute **in parallel** using both shared memory and pipes. To do so, you will create two files:

1. Manager.c
2. Palindrome.c

The Manager (the parent) coordinates the tasks performed by Palindrome (the children). You will be asked to take n words from the command line and distribute the words to the children where they will each perform the same task with their own word. The words provided at the command line will be all lowercase for simplicity. In each child (Palindrome), you will determine if the word received is a valid palindrome and write the result to the Manager.

The **Manager** behaves similarly to **Generator** in the previous assignment, but has the following new capabilities:

1. Creation of unique shared memory segments for the Manager to share an individual word with each Palindrome instance and for each Palindrome instance to store results in.
2. Creation of a pipe for each Palindrome instance that provides it with the name of the shared memory segment created in step (1). The file descriptor (FD) of the pipe is passed as an additional argument to Palindrome.
3. Palindrome processes run concurrently rather than sequentially. This means that the Manager will launch all the child processes and **then** start waiting for results.

As in the previous assignment, each instance of the Palindrome will receive different arguments. As discussed above; to facilitate this, the Manager will take n command line arguments (the words to be validated). For instance,

```
> ./Manager racecar kayak apple abcba hello
```

Would create 5 child processes that would check each word in parallel.

Palindrome requirements are below:

1. It will take a command line argument that gives the FD of the pipe to read from.
2. Using the pipe FD, the Palindrome instance determines the name of the shared memory segment to read its given word from and store its result.
3. Rather than returning the result of the check, the result is stored in the shared memory segment.

2 Overview of assignment requirements:

i. To simplify access and avoid the need to do manual offset calculations we recommend using the struct provided below:

```
typedef struct {
        int result;
        char word[4092];
} shared_mem_t;
```

   a. The Manager writes the input word into word and initializes result = -1. The child reads word, validates, and writes result. This struct will be the basis of your shared memory segment.

ii. **Palindrome** must accept one command line argument, and the **Manager** must accept n command line arguments.

iii. The Manager creates a pipe using the **pipe()** command for each child process. The read end of the pipe will be passed to the Palindrome process, and the write end of the pipe will be used by the Manager to provide the shared memory segment name.

iv. The Manager should spawn n processes using the **fork()** command and print their process IDs as they are created.

v. Child-specific processing immediately following the **fork()** command loads the Palindrome program into the newly created process using the **exec()** command. This ensures that the forked process is no longer a copy of the Manager. This **exec()** command should also pass 1 argument to the Palindrome program: the FD of the read end of the pipe created in (iii).

vi. The Manager sets up the shared memory using **shm_open() and ftruncate()** and writes its name to the pipe.

   a. Attach and initialize the struct (set result = -1, copy the word into the word field), then detach.

vii. The Palindrome process starts executing, prints out its process ID, and retrieves the shared memory segment name from the pipe.

   a. It then retrieves the word from the shared memory segment and check if it is a valid palindrome and prints the information.

   b. If the word is a palindrome, it should write **1** (palindrome) or **0** (not palindrome) to the result field of the struct in shared memory.

viii. Parent-specific processing in the Manager should ensure that the Manager will **wait()** for each instance of the child-specific processing to complete. This is done **AFTER** all the processes have been started. The results retrieved from shared memory should be printed and match up with what was printed in (vii).

ix. Both the Manager and Palindrome should clean up: FDs should be closed and shared memory unlinked (use the **shm_unlink()** command).

x. **Ensure the Manager handles malloc / pipe / fork / exec / shm_open / ftruncate / mmap / munmap / shm_unlink failures with clear error messages and cleanup**

3. Example Output:
This has been color-coded to highlight the three main sections. Do not worry about color-coding your output.

```
> ./Manager racecar kayak apple abcba hello
Manager: forked process with ID 66407.
Manager: wrote shm name to pipe (18 bytes)
Manager: forked process with ID 66408.
Manager: wrote shm name to pipe (18 bytes)
Manager: forked process with ID 66409.
Manager: wrote shm name to pipe (18 bytes)
Manager: forked process with ID 66410.
Manager: wrote shm name to pipe (18 bytes)
Manager: forked process with ID 66411.
Manager: wrote shm name to pipe (18 bytes)
Manager: waiting on child process ID 66407...
Palindrome process [66407]: starting.
Palindrome process [66407]: read word "racecar" from shared memory
Palindrome process [66407]: racecar *IS* a palindrome
Palindrome process [66407]: wrote result (1) to shared memory.
Palindrome process [66408]: starting.
Palindrome process [66408]: read word "kayak" from shared memory
Palindrome process [66408]: kayak *IS* a palindrome
Palindrome process [66408]: wrote result (1) to shared memory.
Palindrome process [66409]: starting.
Palindrome process [66409]: read word "apple" from shared memory
Palindrome process [66409]: apple *IS NOT* a palindrome
Palindrome process [66409]: wrote result (0) to shared memory.
Manager: result 1 read from shared memory. "racecar" is a palindrome.
Manager: waiting on child process ID 66408...
Manager: result 1 read from shared memory. "kayak" is a palindrome.
Manager: waiting on child process ID 66409...
Manager: result 0 read from shared memory. "apple" is not a palindrome.
Manager: waiting on child process ID 66410...
Palindrome process [66410]: starting.
Palindrome process [66410]: read word "abcba" from shared memory
Palindrome process [66410]: abcba *IS* a palindrome
Palindrome process [66410]: wrote result (1) to shared memory.
Palindrome process [66411]: starting.
Palindrome process [66411]: read word "hello" from shared memory
Palindrome process [66411]: hello *IS NOT* a palindrome
Palindrome process [66411]: wrote result (0) to shared memory.
Manager: result 1 read from shared memory. "abcba" is a palindrome.
Manager: waiting on child process ID 66411...
Manager: result 0 read from shared memory. "hello" is not a palindrome.
Manager: exiting.
```

Things to keep in mind for each section:

1. RED: For a problem this size, the OS *should* be able to assign sequential PIDs. This section does happen sequentially as the parent takes each word from the command line and forks the children, so this sequential output makes sense. You should see the parent wait on the first PID it forked as the last statement of this section.
2. BLUE: This is where you will see the effects of parallelism, each time you run your program this section will be interleaved in a different order, so it is okay if your output does not match exactly.
3. GREEN: This is the end so this should always print last.

As with the last assignment, the syntax of each line should match what you see in the example output as we will use regex to assert correctness. Additionally, you will notice the PIDs and shared memory names will change each time you run your program, and this is expected behavior.

Here is the same output with the sections marked by text in case the colors are hard to read (the added lines are not actually apart of the expected output):

```
> ./Manager racecar kayak apple abcba hello
*BEGIN RED SECTION*
Manager: forked process with ID 66407.
Manager: wrote shm name to pipe (18 bytes)
Manager: forked process with ID 66408.
Manager: wrote shm name to pipe (18 bytes)
Manager: forked process with ID 66409.
Manager: wrote shm name to pipe (18 bytes)
Manager: forked process with ID 66410.
Manager: wrote shm name to pipe (18 bytes)
Manager: forked process with ID 66411.
Manager: wrote shm name to pipe (18 bytes)
Manager: waiting on child process ID 66407...
*END RED SECTION*
*BEGIN BLUE SECTION*
Palindrome process [66407]: starting.
Palindrome process [66407]: read word "racecar" from shared memory
Palindrome process [66407]: racecar *IS* a palindrome
Palindrome process [66407]: wrote result (1) to shared memory.
Palindrome process [66408]: starting.
Palindrome process [66408]: read word "kayak" from shared memory
Palindrome process [66408]: kayak *IS* a palindrome
Palindrome process [66408]: wrote result (1) to shared memory.
Palindrome process [66409]: starting.
Palindrome process [66409]: read word "apple" from shared memory
Palindrome process [66409]: apple *IS NOT* a palindrome
Palindrome process [66409]: wrote result (0) to shared memory.
Manager: result 1 read from shared memory. "racecar" is a palindrome.
Manager: waiting on child process ID 66408...
Manager: result 1 read from shared memory. "kayak" is a palindrome.
Manager: waiting on child process ID 66409...
Manager: result 0 read from shared memory. "apple" is not a palindrome.
Manager: waiting on child process ID 66410...
Palindrome process [66410]: starting.
Palindrome process [66410]: read word "abcba" from shared memory
Palindrome process [66410]: abcba *IS* a palindrome
Palindrome process [66410]: wrote result (1) to shared memory.
Palindrome process [66411]: starting.
Palindrome process [66411]: read word "hello" from shared memory
Palindrome process [66411]: hello *IS NOT* a palindrome
Palindrome process [66411]: wrote result (0) to shared memory.
Manager: result 1 read from shared memory. "abcba" is a palindrome.
Manager: waiting on child process ID 66411...
Manager: result 0 read from shared memory. "hello" is not a palindrome.
*END BLUE SECTION*
*BEGIN GREEN SECTION*
Manager: exiting.
*END GREEN SECTION*
```

3 What to Submit

Assignments should be submitted through Canvas. E-mailing the codes to the Professor, GTA, or the class accounts will result in an automatic 1 point deduction.

Use the CS370 *Canvas* to submit a single .zip file that contains:

- Manager.c and Palindrome.c (please document your code)
- A Makefile that performs both a *make clean* as well as a *make all (this will be provided in Teams)*

Filename Convention: Your manager and palindrome must be named Manager.c and Palindrome.c respectively; you can name additional .c and .h files anything you want. The archive file should be named as <LastName>-<FirstName>-HW3.zip. E.g. if you are Cameron Doe and submitting for HW3, then the zip file should be named Doe-Cameron-HW3.zip.

4 Grading

The assignments must compile and function correctly on machines in the CSB-120 Lab. Assignments that work on your laptop on your particular flavor of Linux/Mac OS X, but not on the Lab machines are considered unacceptable. Solutions that do not compile when the `make` command is executed will receive a grade of zero.

Objective	Points
Working Makefile	10
Correct output syntax / Correctly validated inputs	90

You are required to work alone on this assignment.

Notes:

1. This program may not work on your Mac OS X or other systems. Try to run the program on a lab system, especially if you keep getting a segmentation fault when the code seems correct.
    a. Your solution **will** be tested on the lab machines.
2. Please remember to **unlink the shared memory**. Failing to do that may cause problems for other users of the machine.
3. Beware the dangers of race conditions and deadlocks!
    a. Design your programs carefully.
    b. Test your programs for correctness individually before multi-threaded debugging.

5 Late Policy

Click here for the class policy on submitting late assignments.

Revisions: Any revisions in the assignment will be noted below