EECS 678 Lab04 Submission

Question one - fork.c:

Compile and execute the program to understand and answer each question mentioned in the source code file. Revert the state of the program back to its original form before each question.

1. [fork.c - line20] Which process prints this line? **The parent process prints this line.** What is printed? **After fork, Process id = 66957**

2. [fork.c - line35] What will be printed if this line is commented?

print after execlp
Final statement from Process: 67205 (this is the ID of the child process)

3. [fork.c - line38] When is this line reached/printed? **It is reached only when execlp is commented out because execlp overwrites the address space of the process in which it is called with a new executable. In this case, it overwrites the address space of the child process with pid 67205 with the executable /bin/ls.**

4.[fork.c - line30] What happens if the parent process is killed first? Uncomment the next two lines. **The OS assigns the child process to a new parent. In this case, the OS assigns the child process with pid 67743 to the init process with pid 1 as shown in the printed line *In Child: 67743, Parent: 1***

Question two - fork.c:

Execute the program once to understand and answer the question:

1. How many processes are created? **There are 8 processes created.** Why? Because there are three fork system calls in the program. **The total number of processes created is given by 2^n, where n is the number of forks. The parent process with pid 68162 forks three child processes with pids 68163, 68164, & 68165. The child process 68163 forks two processes with pids 68166 & 68167. The child process 68164 forks one process with pid 68168. The grandchild process with pid 68166 forks one process with pid 68169.**

Question three - pipe-sync.c:

Update the program to exhibit the behavior described in the source file. Paste your modified source code (of the main() function only) in the text box provided.

```
int main()

{

 char *s, buf[1024];

 int ret, stat;

 s  = "Use Pipe for Process Synchronization\n";

 /* create pipe */

 int p1[2],p2[2],p3[2];  //file descriptors

 pipe(p1);

 pipe(p2);

 pipe(p3);

 ret = fork();

 if (ret == 0) {

   /* child process. */

   printf("Child line 1\n");

   write(p1[1],s,strlen(s));

   read(p2[0],buf,strlen(s));

   printf("Child line 2\n");

   write(p3[1],s,strlen(s));

 }
```

```
else {

   /* parent process */

   read(p1[0],buf,strlen(s));

   printf("Parent line 1\n");

   write(p2[1], s, strlen(s));

   read(p3[0],buf,strlen(s));

   printf("Parent line 2\n");

   wait(&stat);

 }

}
```

Question four - fifo_producer.c & fifo_consumer.c:

Modify the source files to create and open a FIFO for reading and writing, respectively. Compile the programs. Open multiple terminal windows and answer the following questions:

1. What happens if you only launch a producer (but no consumer)? **The program prints "waiting for readers . . .", which means it's blocking until a writer is available.**

2. What happens if you only launch a consumer (but no producer)? **The program prints "waiting for writers . . ." which means it's blocking until a writer is available.**

3. If there is one producer and multiple consumers, who gets the message sent? **The consumers read the FIFO in an interleaved manner (one after the other) starting with the first consumer that was created, followed by the second, and so on.**

4. Does the producer continue writing messages into the FIFO if there are no consumers? **No, the producer terminates immediately after writing the next message following the termination of the last consumer.**

5. What happens to the consumers if all the producers are killed? **The consumers are automatically terminated as well.**

Question four - shared_memory3.c:

Read and understand the code. Compile the code and run it. Explain the output of the program.

The program basically uses the child process to update a shared and unshared memory segment, showing the distinction between the two. The child process calls a function do_child() which performs this action.

shared_buf before fork: First String

unshared_buf before fork: First String

The first two lines above from the output were printed by the parent process just before the fork() system call. We see that the contents of shared_buf and unshared_buf are the same as the value used to initialize them on lines 52 and 53. The parent process waits for the child to finish and collects the return value in the variable status.

shared_buf in child: First String

unshared_buf in child: First String

These next two lines were printed from the child process. Both shared_buf and unshared_buf contain "First String" and are passed as arguments to the do_child() routine, which is only accessible from the child (i.e. when p1d == 0).

shared_buf after fork: Second String

unshared_buf after fork: First String

These last two lines are printed when the parent process resumes. We see that output “Second String” from shared_buf was accessible from the parent process, though it was written to by the child process. On the other hand, the content of unshared_buf in the parent process was unchanged (not accessible to the parent process), though it was written to by the child process.

Question five - thread-1.c:

Observe the execution and answer the two questions in the source file:

1. Are changes made to the local or global variables by the child process reflected in the parent process? Explain.

No, this is because the child process merely copies the address space of the parent process after a fork() system call. Therefore, it can only modify its own copy of the local and global variables.

2. Are changes made to the local or global variables by the child thread reflected in the parent process? Explain.

Yes, this is because as a thread, it shares with the parent its code section, data section, and other operating system resources. Therefore, when it calls child_fn() routine to modify local and global variables, this is also reflected in the parent process.