Kam Bielawski - 2972621 - EECS 678 Lab 5

Practice Process, Thread, and IPC Concepts

fork.c

1. Execute the program to understand and answer each question mentioned in the source code file

a. Get the program back to the original state for each question

b. Question 1: Which process prints this line? What is printed?

Both the parent and the child process will print this line - for the child process, “After fork, Process id = 0” will be printed, and the parent will be some positive integer that is its process id (in my case it was “After fork, Process id = 116827”)

c. Question 2: What will be printed if this like is commented?

“Print after execlp” will be printed because the previously exec’d process will no longer replace the child process.

d. Question 3: When is this line reached/printed?

Only if the execlp call has failed, or if it is commented out.

e. Question 4: What happens if the parent process is killed first? Uncomment the next two lines.

It looks like when the parent process is done, we go back to the shell program. But the child process is still running, and after sleep(4) is done, the two lines are executed and we see “In Child: 117012, Parent: 1531” and the /bin/ls results. This contradicts what I expected, because I thought orphaned processes would become the child of process 1 (init).

mfork.c

1. Execute the program once to understand and answer the question

a. Question 1: How many processes are created? Explain.

8 *total* processes are created, including the parent. The parent is 1, the first fork() creates a 2nd process, the 2nd fork() is exec’d by the two existing processes which makes 4, and then all 4 exec the third fork() which makes 8 total processes.

pipe-sync.c

1. Update the program to answer the question in the source code file.

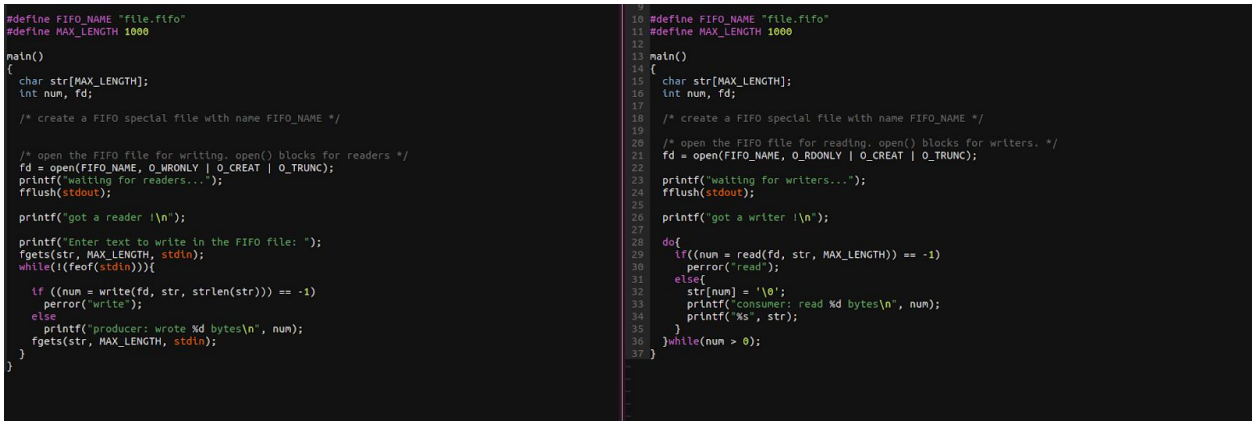
```
[10:33] ~/dev/spring21/operating_systems_678/lab5 >>> gcc -o pipe-sync pipe-sync.c
[10:33] ~/dev/spring21/operating_systems_678/lab5 >>> ./pipe-sync
Child line 1
Parent line 1
Child line 2
Parent line 2
```

```c
int main()
{
  char *s, buf[1024];
  int ret, stat;
  s  = "Use Pipe for Process Synchronization\n";

  /* create pipe */
  int p[2];
  pipe(p);

  ret = fork();
  if (ret == 0) {

    /* child process. */
    printf("Child line 1\n");
    // redirect stdout to the write end
    dup2(p[1], STDOUT_FILENO);
    printf("Child line 2\n");
    close(p[0]);
  } else {

    /* parent process */
    // wait for child process to write
    int n;
    while ((n = read(p[0], buf, sizeof(buf))) == 0)
        ;

    printf("Parent line 1\n");
    printf("%s", buf);
    printf("Parent line 2\n");

    wait(&stat);
  }
}
```

fifo_producer.c and fifo_consumer.c

1. Create a fifo and open it for writing and reading, respectively

a. Use slides 37 and 38 in Chapter 3



```c
#define FIFO_NAME "file.fifo"
#define MAX_LENGTH 1000

main()
{
  char str[MAX_LENGTH];
  int num, fd;

  /* create a FIFO special file with name FIFO_NAME */


  /* open the FIFO file for writing. open() blocks for readers */
  fd = open(FIFO_NAME, O_WRONLY | O_CREAT | O_TRUNC);
  printf("waiting for readers...");
  fflush(stdout);

  printf("got a reader !\n");

  printf("Enter text to write in the FIFO file: ");
  fgets(str, MAX_LENGTH, stdin);
  while(!(feof(stdin))){

    if ((num = write(fd, str, strlen(str))) == -1)
      perror("write");
    else
      printf("producer: wrote %d bytes\n", num);
    fgets(str, MAX_LENGTH, stdin);
  }
}
```

```c
#define FIFO_NAME "file.fifo"
#define MAX_LENGTH 1000

main()
{
  char str[MAX_LENGTH];
  int num, fd;

  /* create a FIFO special file with name FIFO_NAME */

  /* open the FIFO file for reading. open() blocks for writers. */
  fd = open(FIFO_NAME, O_RDONLY | O_CREAT | O_TRUNC);

  printf("waiting for writers...");
  fflush(stdout);

  printf("got a writer !\n");

  do{
    if((num = read(fd, str, MAX_LENGTH)) == -1)
      perror("read");
    else{
      str[num] = '\0';
      printf("consumer: read %d bytes\n", num);
      printf("%s", str);
    }
  }while(num > 0);
}
```

I tried for a while to get proper behavior on this but couldn't figure it out

2. Compile the programs

3. Open 4 terminals and answer the following questions

a. What happens if you only launch a producer (but no consumer)?

The producer does indeed write to the file (confirmed by 'cat file.fifo'), but there is no consumer to read it.

b. What happens if you only launch a consumer (but no producer)?

If there is anything in file.fifo, the consumer will read and output it. Otherwise, it reads 0 bytes and exits.

c. If one producer and multiple consumers, then who gets the message sent?

Each consumer

d. Does the producer continue writing messages into the fifo, if there are no consumers?

Yes

e. What happens to the consumers, if all the producers are killed?

Nothing

shared_memory3.c

1. Understand the code

2. Complete the "do_child()" function as indicated by the comments there

There doesn't seem to be anything to do?

```
13 /* Child process */
14 int do_child(char *shared, char *unshared)
15 {
16   /* Print original values in the two buffers */
17   fprintf(stdout, "shared_buf in child: %s\n", shared);
18   fprintf(stdout, "unshared_buf in child: %s\n", unshared);
19
20   /* Update the two buffers with STR2 */
21   strcpy(shared, STR2);
22   strcpy(unshared, STR2);
23
24   return 0;
25 }
```

3. Compile/execute the program

```
[10:41] ~/dev/spring21/operating_systems_678/lab5 >>> gcc -o shmem shared_memory3.c
[10:41] ~/dev/spring21/operating_systems_678/lab5 >>> ./shmem
shared_buf before fork: First String
unshared_buf before fork: First String
shared_buf in child: First String
unshared_buf in child: First String
shared_buf after fork: Second String
unshared_buf after fork: First String
```

4. Question-1: Explain the output.

The first two lines are before we've forked, displaying the values in both buffers.

The second two are *after* the fork, displaying from the child process (printed from do_child) - the values of each buffer have not been changed

The last two lines are the interesting ones: The shared_buf has been updated, because of line21 in do_child(). This is displayed in the 5th line of the output, "Second String". The last line shows that unshared_buf of course has not been updated in the parent even though the child updated it in its own process. This is because it is not shared between the parent and child.

thread-1.c

1. Compile and execute the program

a. gcc -o thread1 thread-1.c -pthread

b. ./thread1

```
[10:41] ~/dev/spring21/operating_systems_678/lab5 >>> gcc -o thread1 thread-1.c -pthread
[10:47] ~/dev/spring21/operating_systems_678/lab5 >>> ./thread
-bash: ./thread: No such file or directory
[10:47] ~/dev/spring21/operating_systems_678/lab5 >>> ./thread1
Original values, global = 100, local = 678
After fork, global = 100, local = 678
After thread, global = 678 local = 100
```

2. Observe and execution and answer the two questions referenced in the source code file

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

Of course not - a fork creates a brand new process, making a copy of the entire process state, *even the global variables,* for the child to use and alter independent of the parent's values for these variables.

b. Question 2: Are changes made to the local or global variables by the child thread reflected in the parent process? Separately explain what happens for the local and global variables.

Yes, because a thread shares memory with the process that calls it. The thread only gets its own stack in memory, nothing else - so it shares global and heap and stack variables with the process that calls it, which is why after the thread is done executing the parent's local and global variables have been updated.

The local variable's address has been passed to the thread, the thread updates it, and is finished executing. There has been no copy of the local variable in this process, so the parent which called the thread has a new value of local.

The global variable is even simpler - since the thread does not create a copy of the process's memory, the thread is updating the parent process's global directly, which is why we see it updated even when the thread is done executing.