Bi-directional fifo communication blocks when reading an intermediate value

Asked 11 years, 11 months ago Modified 11 years, 11 months ago Viewed 2k times

2

I am trying to send information back and forth between two processes using a fifo. It works up to a point, but then a read blocks. I suspect Process2 is where the bug is.

Process1:

```
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
main()
{
 char oprtr;
 int fd1,fd0;
 float oprnd1,oprnd2,result;

 mkfifo("fifo1",0777);
 fd1=open("fifo1",O_RDWR);

 printf("fd1:%d\n",fd1);
 printf("Add(+)\n");
 printf("subtract(-)\n");
 printf("multiply(*)\n");
 printf("division(/)\n");

 printf("Enter operator\n");
 scanf("%c",&oprtr);
 getchar();
 write(fd1,&oprtr,sizeof(oprtr));

 printf("Enter oprnd1\n");
 scanf("%f",&oprnd1);
 getchar();
 write(fd1,&oprnd1,sizeof(oprnd1));

 fd0=dup(fd1);
 printf("Enter oprnd2\n");
 scanf("%f",&oprnd2);
 getchar();

 if(write(fd0,&oprnd2,sizeof(oprnd2))==0)
  perror("write : oprnd2:");
 else
  printf("writing oprnd2 done\n");

 read(fd1,&result,sizeof(result));
 printf("Result:%f\n",result);
}
```

Process2:

```
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
main()
{
 int fd2,fd3;
 char oprtr;
 float oprnd1,oprnd2,result;
 fd2=open("fifo1",O_RDWR);
 printf("fd2:%d\n",fd2);

 read(fd2,&oprtr,sizeof(oprtr));
 printf("oprtr:%c\n",oprtr);

 read(fd2,&oprnd1,sizeof(oprnd1));
 printf("oprnd1:%f\n",oprnd1);

 fd3=dup(fd2);
```

This is where the read function is getting blocked

The above two read() calls appear to be working fine, but the following read() call is getting blocked. Why?

```
 if(read(fd3,&oprnd2,sizeof(oprnd2))==0) ////This is the problem
  perror("read : oprnd2:");
 else
  printf("oprnd2:%f\n",oprnd2);

switch(oprtr)
 {
 case '+':result=oprnd1+oprnd2;
          write(fd2,&result,sizeof(result));break;
 case '-':result=oprnd1-oprnd2;
          write(fd2,&result,sizeof(result));break;
 case '*':result=oprnd1*oprnd2;
          write(fd2,&result,sizeof(result));break;
 case '/':result=oprnd1/oprnd2;
          write(fd2,&result,sizeof(result));break;
 default: printf("Wrong Choice\n");break;
 }
}
```

Terminal 1:

```
Add(+)
subtract(-)
multiply(*)
division(/)
Enter operator
+
Enter oprnd1
14.56
Enter oprnd2
16.44
```

```
writing oprnd2 done
Result:16.440089
```

Terminal 2:

```
fd2:3
oprtr:+
oprnd1:14.560000
```

Then it just gets blocked

c linux named-pipes system-calls fifo

Share

Improve this question

Follow

edited Jan 25, 2013 at 17:08

Scott - Слава Україні
678 ● 16 ● 28

asked Jan 25, 2013 at 2:49

Neeraj
239 ● 5 ● 14

It looks like you are trying to create a client-server relationship between two processes. If you are, and this is a serious project (i.e., not just a demonstration or toy), I suggest that you consider using Remote Procedure Calls (RPCs). – Scott - Слава Україні Jan 25, 2013 at 15:25

1 Answer

Sorted by: Highest score (default)

2

The short and simple answer is that you shouldn’t use a single fifo for bi-directional communication without some external synchronization mechanism.

To elaborate: a Unix pipe, or fifo, is not best visualized as a plumbing-type pipe. It’s more like a storage tank, with pipes leading to and from it. The standard usage is that one process writes, thus filling the tank, and another process reads, thus draining the tank. In your program, everything that gets written to the fifo sits in the tank until somebody comes along and reads it, first-come-first-served. And so your Process1 program reads the 16.44 value, that it wrote from `oprnd2`, back into `result`. This leaves the tank empty, so there is nothing for Process2 to read. It comes down to a race condition. I suspect that, if you ran these two commands a few hundred times, they would work the way you want a few times.

If you just want proof-of-concept, add a `sleep` to Process1 before the `read` from the fifo. A better solution would be to use two fifos, one for each direction. Or you could devise some way for Process2 to let Process1 know that it (Process2) has read the operator and both operands and has written the result — e.g., a signal or a semaphore — but that would probably be more trouble than it’s worth.

On another matter, I strongly recommend against the use of `scanf` in anything but a toy, demonstration, or throw-away prototype type of program. If the user hits just Enter, the program will just sit there forever. I recommend reading a line (being sure to address buffer-overflow) and then call `sscanf` on it.

Oh, and also, please indent your code.

Share Improve this answer Follow

answered Jan 25, 2013 at 3:17

Scott - Слава Україні
678 ● 16 ● 28

Another reason not to use `scanf` — you obviously know that it leaves the newline in the input buffer. I hope you understand that, in general, it reads just as much input as it needs to, and leaves the rest unread. So, if your user gets confused and types “ `+-*` ” in response to the “ `Enter operator` ” prompt, the `scanf` will read the `+`, your `getchar()` will read the `-`, and the `*` will be left in the input stream, where it will corrupt the `scanf("%f", &oprnd1);`. – Scott - Слава Україні Jan 25, 2013 at 15:24

yes that's true ,I'll have to take care of that.Thanks again for this support @Scott – Neeraj Jan 26, 2013 at 1:56