All about Cdac Hyderabad - DESD

Friday, 21 October 2016

All About Inter-Process Communication

Note : - This post has been written for CDAC Hyd DESD - August - 2016 batch Students.

Lets start with some of the basic questions :
(I hope you know about processes, if No then you can check my previous posts in your facbook group)

1) What is IPC- Inter-Process Communication
2) Why it is needed ?
3) And How this process Communication is done.

Answers :
1) What is IPC- Inter-Process Communication ?

As the name itself denotes i.e a way for communicating between processes(two processes or more than two) . Right !! OK ... i have said **TWO OR MORE PROCESSES.**

Concentrate please ... These said two processes can be either related or unrelated.(What the hell is this related and unrelated).

RELATED Processes : Example : Parent and children or siblings. Why related - Because these have some relations between them(Common father or grand father). Someaning is processes generated by fork are related processes.

UNRELATED Process : Assume that there are two programs . add_main.c (add_main.o) and subtract_main.c. Assume that i am running these two programs in a while(1) loop and running them at the same time (How !! how will run two programs at the same time).Simple : Just open three terminals and run your two programs in two different terminals. Open the third terminal and type **ps -elf** . You can see pid of your processes running.

These two programs are known as UNRELATED processes.(Why i have explained this you will come to know soon)

2)Why IPC is needed ?
Why you need a mobile phone ??

3) **HOW**

Inter process communication is done using :

1. Signals - Sent by other processes or the kernel to a specific process to indicate various conditions(Less applicable)
2. Pipes -
3. FIFOS -
4. Message queues - Message queues are a mechanism set up to allow one or more processes to write messages that can be read by one or more other processes.
5. Semaphores - Counters that are used to control access to shared resources. These counters are used as a locking mechanism to prevent more than one process from using the resource at a time.
6. Shared memory - The mapping of a memory area to be shared by multiple processes.

Now Lets come to the main topic :

Pipes : - Do one thing. Open terminal Type ps -elf
You will get something like this -

```
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root         1     0  0  80   0 -  8478 poll_s 21:01 ?        00:00:02 /sbin
1 S root         2     0  0  80   0 -     0 kthrea 21:01 ?        00:00:00 [kthr
```

Blog Archive

```
1 S root         3     2  0  80   0 -     0 smpboo 21:01 ?        00:00:00 [ksof
1 S root         5     2  0  60 -20 -     0 worker 21:01 ?        00:00:00 [kwor
```

A big list of all the processes.This ps -elf is a command (actually a process) that is **PRODUCING OUTPUT.**

Now i will tell you one more command **grep** . This command takes a **INPUT** and searches a specific word provided with grep.
USED AS :

```
grep "literal_string" filename
```

Now i want to search pid of a process named bash .What will i need?
I will need to send the OUTPUT OF ps -elf as input to grep command . Isn't it ??

So don't you see that here there is a need of communication between these two commands.

OK here PIPE CHARACTER " | " comes into picture.

now type this in the terminal : ps -elf | grep bash

And see the result.(Must try it. First try the commands individually and next with pipe) Hope you have understood how and why we are using pipe.

Ok.. this was related to command line. What if i need to do the same in my programs ??

Related Programs : We need to use **pipe system call.**
Ex- If We need to communicate between processes created by fork(), we need to use unnamed pipe.

```
#include <unistd.h>
int pipe(int file_descriptor[2])
```

pipe() creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by *filedes*. *filedes[0]* is for reading, *filedes[1]* is for writing. On success returns 0;

How we use pipes and how pipe works ?

step 1:
we create an array of int type of size two - int arr[2];
the elements will be arr[0] and arr[1], [0] and [1] are the index number of the array nothing else.
(what is **arr** here ?? fd is the address of the array itself or the address of the first element,isn't it?).

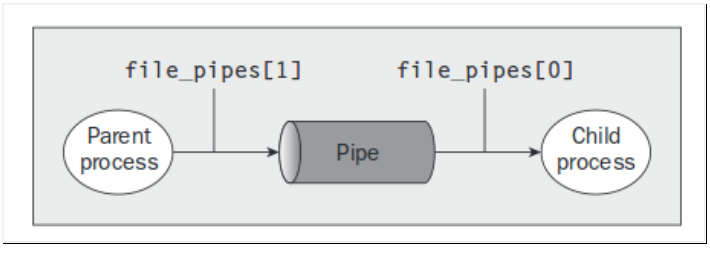
step 2:
we call the pipe system call. We pass the address of that array created earlier as argument to the system call.
Like - pipe(**arr**) ; This will update the elements of the array **arr** created earlier.
i.e the values of arr[0] will be "a file descriptor that will be used for reading the values or accepting the data passed through the pipe **(Hope u remember that | pipe character on command line. (we used it to just pass the output of one command to input of another)Here one process was accepting the data i.e reading instead).**

similarly arr[1] will be updated with a file descriptor that will be used for writing(). (send the data).

step 3:
writing and reading
Suppose that we have two process - one parent and one child.
parent want to send something to child.

what will we do. In parent we we create a pipe and then we will call fork()...

Now in parent process we will write something using :
write(arr[1],some_buffer,size)
similarly we will read in child process using
read(arr[0],some_another_buffer,size)



parent process is writing while child process is reading.

whole program :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
int data_processed;
int file_pipes[2];
const char some_data[] = “123”;
char buffer[BUFSIZ + 1];
pid_t fork_result;
memset(buffer, ‘\0’, sizeof(buffer));
if (pipe(file_pipes) == 0) {
fork_result = fork();
if (fork_result == -1) {
fprintf(stderr, “Fork failure”);
exit(EXIT_FAILURE);
}
if (fork_result == 0) {
data_processed = read(file_pipes[0], buffer, BUFSIZ);
printf(“Read %d bytes: %s\n”, data_processed, buffer);
exit(EXIT_SUCCESS);
}
else {
data_processed = write(file_pipes[1], some_data,
strlen(some_data));
printf(“Wrote %d bytes\n”, data_processed);
}
}
exit(EXIT_SUCCESS);
}
```

NOTE :
A read call will normally block; that is, it will cause the process towait until data becomes available. (Blocking means waiting : Just like weuse scanf or getch() and the program waits until we provide some input to these calls) If the other end of the pipe has been closed, then noprocess has the pipe open for writing, and the read blocks.

.....to be continued

Posted by Prakash Arunakar at 11:25

Labels: os_tut

No comments:

Post a Comment

Enter your comment...

Comment as: Mohit Ingale (G ▾ Sign out

Publish Preview Notify me

Newer Post Home

Subscribe to: Post Comments (Atom)