OS Tutorial

BITS Pilani
Pilani Campus

Interprocess communication (IPC)

IPC methods

- Signal
- Pipe
- FIFO
- Message passing
- Shared Memory
- Semaphores

Pipe

- Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems
- Most commonly used form of IPC
- Historically, they have been half duplex (i.e., data flows in only one direction).
- Because they don't have names, pipes can be used only between processes that have a common ancestor.
  - Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

Pipe

- Process can communicate through pipe.
- A pipe *is used for one-way communication* of a stream of bytes.
- Signals inform processes of the occurrence of asynchronous events.

Pipe

- Pipes are familiar to most Unix users as a shell facility.
- **Example:** To print a sorted list of who is logged on, you can enter this command line:

  who|sort|lpr
- There are three processes here, connected with two pipes. Data flows in one direction only, from **who** to **sort** to **lpr.**

Pipe

- It is also possible to set up bidirectional pipelines (from process A to B, and from B back to A) and pipelines in a ring (from A to B to C to A) using system calls.
- Related system calls
  - ✓pipe
  - ✓dup/dup2

pipe() system call

- **Syntax**

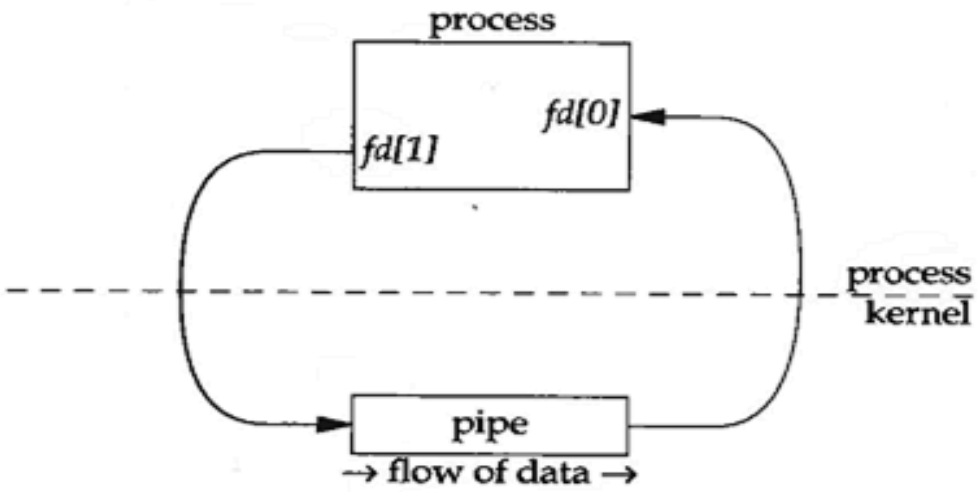
  int pipe (pfd);

  int pfd[2];

- Returns 0 on success or -1 on error

pipe() system call

- Creates a pipe and stores a pair of file descriptors into **pdf**.
- **pdf[2]** is an array.
  - fd[0] is open for reading
  - fd[1] is open for writing



The following program creates, writes to, and reads from a pipe.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
void main()
{
int pfd[2];
char buf[30];
if (pipe(pfd) == –1)
{
perror("\n Error in pipe creation \n");
exit(1);
}
printf("\n writing to file descriptor %d\n",
    pfd[1]);
write(pfd[1], "test", 5);
printf("\n reading from file descriptor %d\n",
    pfd[0]);
read(pfd[0], buf, 5);
printf("read: %s \n", buf);
}
```

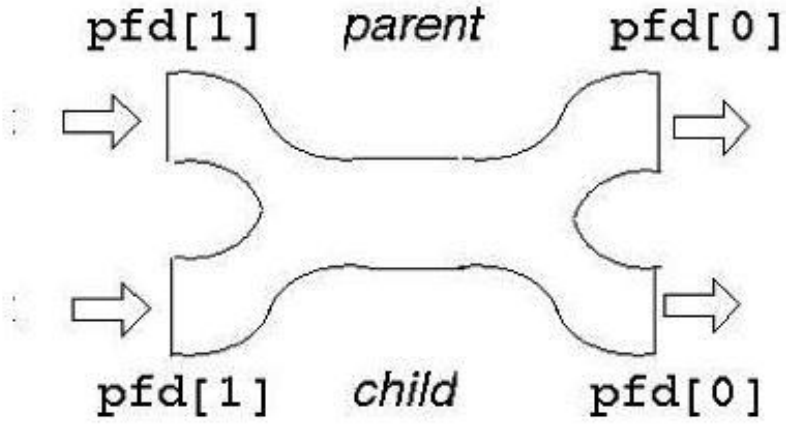
pipe() and fork() system calls

- A single process would not use a pipe.
- They are used when two processes wish to communicate in a one-way fashion.
- **A process splits in two using fork()**.
- A pipe opened before the fork becomes shared between the two processes.

pipe() and fork() system calls

Before fork

After fork

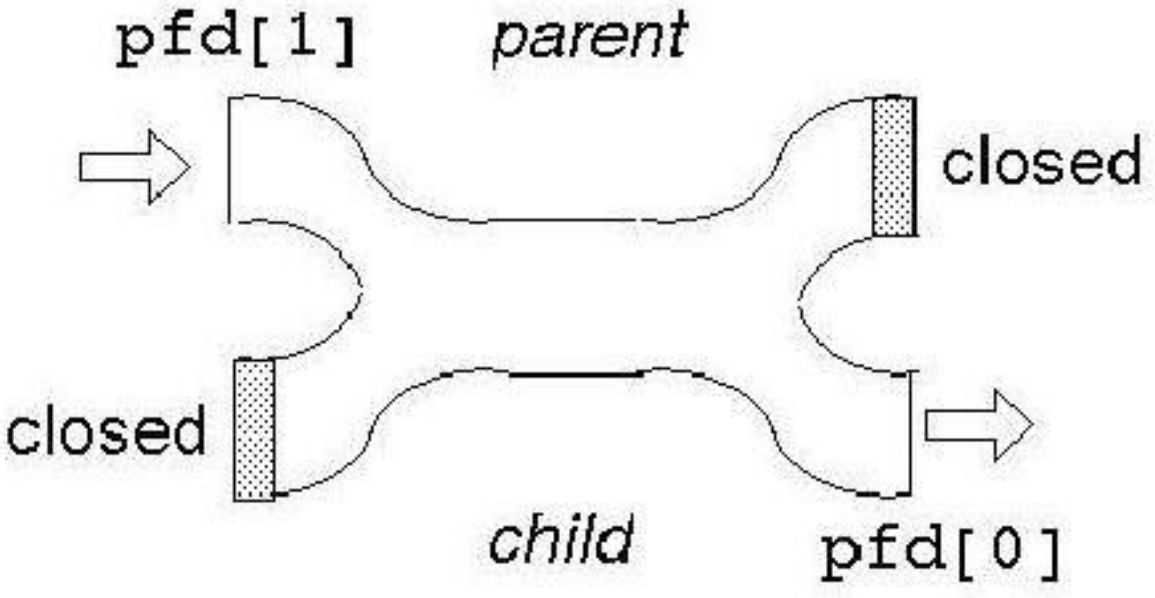


pipe() and fork() system calls

- It gives two read ends and two write ends.
- The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.
- Either process can write into the pipe, and either can read from it. Which process will get what is not known.

pipe() and fork() system calls

- For predictable behavior, one of the processes must close its read end, and the other must close its write end. Then it will become a simple pipeline again.



```
#include <stdio.h>
#define SIZE 1024
main( )
{
int pfd[2];
int nread;
int pid;
char buf[SIZE];
if (pipe(pfd) == -1)
{
perror("pipe failed");
exit(1);
}
if ((pid = fork()) < 0)
{
perror("fork failed");
exit(2);
}
if (pid == 0)
{
/* child */
close(pfd[1]);
while ((nread = read(pfd[0], buf, SIZE)) != 0)
printf("child read %s\n", buf);
close(pfd[0]);
}
else
{
/* parent */
close(pfd[0]);
strcpy(buf, "hello...");
/* include null terminator in write */
write(pfd[1], buf, strlen(buf)+1);
close(pfd[1]);
}
}
```

OUTPUT

child read hello...

```
#include <stdio.h>#include <stdlib.h>#include <sys/types.h>#include <unistd.h>
int main()   {
int pfd[2];
Int pid;
char buf[30];
pid=fork();
pipe(pfd);
if (fork()=0) {
printf("\n CHILD: writing to the pipe\n");
write(pfd[1], "test", 5);
printf("\n CHILD: exiting\n");
exit(0);  }
 else {
printf("\n PARENT: reading from pipe\n");
read(pfd[0], buf, 5);
printf("\n PARENT: read \"%s\"\n", buf);
wait(NULL);  }
}
```

PARENT: reading from pipe
CHILD: writing to the pipe

PARENT: read "test"

CHILD: exiting

PARENT: reading from pipe
CHILD: writing to the pipe
CHILD: exiting
PARENT: read "test"

```
#include <stdio.h> #include <stdlib.h> #include <unistd.h>
int main() {
int pfds[2];
pipe(pfds);
if (fork()==0)
{                   /* In the child close the writing end of the pipe,*/
close(1);       /* close normal stdout */
dup(pfds[1]);     /* make stdout same as pfds[1] */
close(pfds[0]);    /* we don't need this */
execlp("ls", “-l", NULL);
}
else
{           /* In the parent close the reading end of the pipe. */
close(0);     /* close normal stdin */
dup(pfds[0]);   /* make stdin same as pfds[0] */
close(pfds[1]);  /* we don't need this */
execlp("wc",  "–l", NULL);
} }
```

- This is the program that we will use to run (ls –l) and (wc –l).
- dup() takes an open *file descriptor* and makes a clone (a duplicate) of it. This is how we will connect the *standard output* of the ls to the standard input of wc.
- stdout of ls flows into the pipe, and the stdin of wc flows in from the pipe.
- close(1) frees up file descriptor 1 (standard output).

- dup(pfds[1]) makes a copy of the write–end of the pipe in the first available file descriptor, which is "1", since we just closed that.
- In this way, anything that ls writes to standard output (file descriptor 1) will instead go to pfds[1] (the write end of the pipe).
- The **wc** section of code works the same way, except in reverse.

This program uses a pipe to allow the parent to read a message from its child


```
#include <stdio.h>
 #include <string.h>
#define   READ     0
#define   WRITE   1
char* phrase = "This is OS Tute Class" ;
void  main ( )
{
int  fd[2], bytesread ;
char message [100] ;
pipe (fd) ;
if ( fork() == 0 ) /* child, writer */
{
close (fd [READ] ) ; /* close unused end */
write (fd [WRITE], phrase, strlen (phrase) + 1) ;
close (fd [WRITE] ) ; /* close used end */  }
else /* parent, reader */
{
close (fd [WRITE] ) ; /* close unused end */
bytesread = read (fd [READ], message, 100) ;
printf ("Read %d bytes : %s\n", bytesread, message) ;
close ( fd [READ] ) ; /* close used end */
}
 }
```

dup() system call

- Syntax

  int dup (int oldfd)

- dup() finds the smallest free file descriptor entry and points it to the same file as oldfd.

dup2() system call

- Syntax

  int dup2(int oldfd, int newfd)

- An existing file descriptor *oldfd* is duplicated as file descriptor *newfd*.
- If the file corresponding to descriptor newfd is open, then it is closed.
- In both cases, the original and copied file descriptors share the same file pointer and access mode.

dup/dup2 system call

- They both return the index of the new file descriptor if successful, and –1 otherwise.
- dup/dup2 duplicates an existing file descriptor, giving a new file descriptor that is open to the same file or pipe.
- The call fails if the argument is bad (not open) or if 20 file descriptors are already open.

Redirection

- In computing, redirection is a function common to most command-line interpreters, including the various Unix shells that can redirect standard streams to user-specified locations.

Redirection

- In unix-like OSs programs do redirection with the dup2() system call.
- date > today
- date < today
- Sort < infile > outfile (sort reads from infile and writes to outfile)
- date >> today (to append output at the end of file)

Implementation of Redirection

- When a process *fork*s, the child inherits a copy of its parent's file descriptors.
- When process *exec*s, the standard input, output, and error channels remain unaffected.
- The UNIX shell uses these two pieces of information to implement redirection.

Implementation of Redirection

To perform redirection, the shell performs the following series of actions:

- The parent shell forks and then waits for the child shell to terminate.
- The child shell opens the file “output”, creating it or truncating as necessary.
- The child shell then duplicates the file descriptor of “output” to the standard output file descriptor, number 1, and then closes the original descriptor of “output”. All standard output is therefore redirected to “output”.

Implementation of Redirection

- The child shell then exec's the ls utility. Since the file descriptors are inherited during an exec(), all of standard output of ls goes to "output".
- When the child shell terminates, the parent resumes. The parent's file descriptors are unaffected by the child's actions, as each process maintains its own private descriptor table.

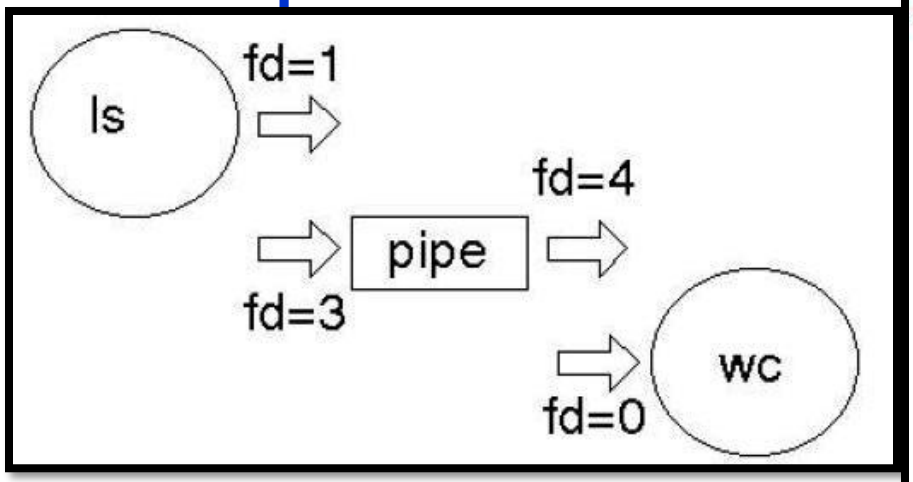
```
#include <fcntl.h>
#include <stdio.h>
#include <sys/file.h>
main (argc, argv)
int argc ;
char *argv[ ] ;
{
int fd ;
/* file descriptor or pointer
*/
fd = open (argv[1], O_CREAT |
O_TRUNC | O_RDWR, 0777) ;
/* open file named in argv[1] */
dup2 (fd, 1) ; /* and assign it
to fd file pointer */
close (fd) ; /* duplicate fd with 1
which is standard output (the
monitor) */
execvp (argv[2],
&argv[2]) ;
/* the output is not printed on
screen but is redirected to "output"
file */
printf ("End\n") ; /*
should never execute */
}
```

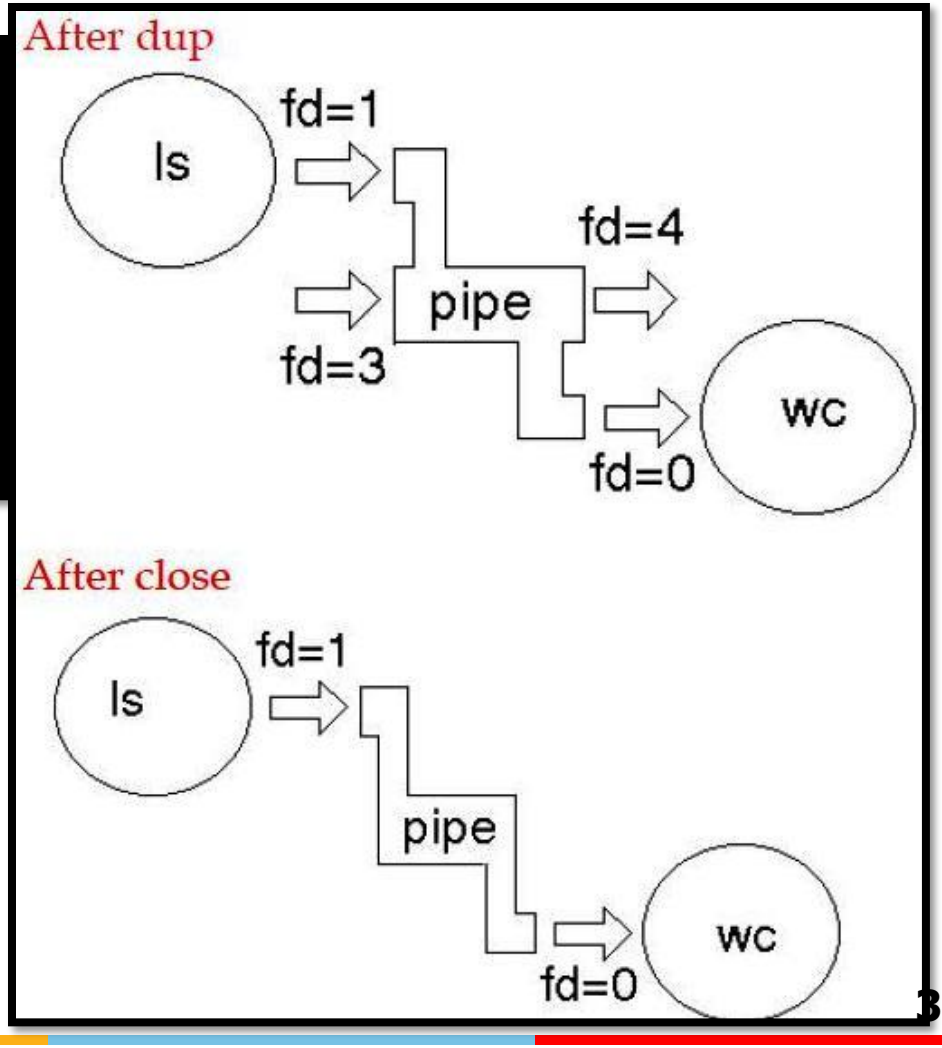
Example

- To implement "ls|wc" the shell will have created a pipe and then forked.
- The parent will exec to be replaced by "ls", and the child will exec to be replaced by "wc".
- The write end of the pipe may be descriptor 3 and the read end may be descriptor 4.
- "ls" normally writes to 1 and "wc" normally reads from 0.

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/file.h>
#include <string.h>
#define READ     0
#define WRITE    1
void main (argc, argv)
int argc ;
char* argv[] ;
{
int pid, fd [2] ;
if (pipe(fd) == -1)
 {
perror("pipe failed");
exit(1);
 }
if ((pid = fork( )) < 0)
 {
perror("fork failed");
exit(2);
}
if ( pid != 0 ) /* parent, writer */
{
close ( fd [READ] ) ; /* close unused end
*/
dup2 ( fd [WRITE], 1) ; /* duplicate used
end to standard out */
close ( fd [WRITE] ) ; /* close used end */
execlp ( argv[1], argv[1], NULL) ; /*
execute writer program */
}
else /* child, reader */
{
close ( fd [WRITE] ) ; /* close unused end
*/
dup2 ( fd [READ], 0) ; /* duplicate used
end to standard input */
close ( fd [READ] ) ; /* close used end */
execlp ( argv[2], argv[2], NULL) ; /*
execute reader program */
}
}
```

Before dup


ls
fd=1
fd=4
pipe
fd=3
wc
fd=0


After dup
ls
fd=1
fd=4
pipe
fd=3
wc
fd=0
After close
ls
fd=1
pipe
wc
fd=0