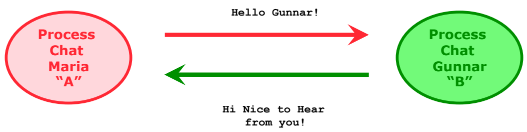
CSCI 6730/ 4730 Operating Systems

Dup & Pipe

Two Communicating Processes



- Concept that we want to implement

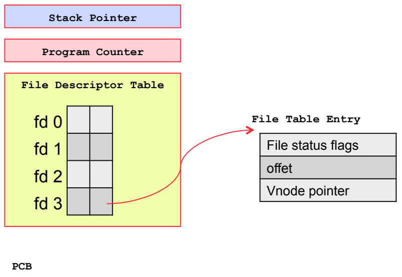
On the path to communication…

- Want: A communicating processes
- Have so far: Forking – to create processes
- Problem:
  - After fork() is called we end up with two independent processes.
  - Separate Address Spaces
- Solution? How do we communicate?

Review 1730 - File: The Unix Way

- One easy way to communicate is to use files.
  - Process A writes to a file and process B reads from it
- File descriptors
  - Mechanism to work with files
  - Used by low level I/O
    - Open(), close(), read(), write()
  - file descriptors generalize to other communication devices such as pipes and sockets

Big Picture (more on this later)



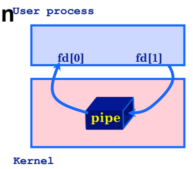
Producer -> Consumer Problems

- Simple example: who | sort
  - Both the writing process (who) and the reading process (sort) of a pipeline execute concurrently.
- A pipe is usually implemented as an internal OS *buffer* with 2 file descriptors.
  - It is a resource that is concurrently accessed
    - by the reader and the writer, so it must be managed carefully (by the Kernel)

Buffering: Programming with Pipes

```
#include <unistd.h>

int pipe( int fd[2] );
```

- `pipe()` binds `fd[]` with two file descriptors:
  - » `fds[0]` used to read from pipe
  - » `fds[1]` used to write to pipe
- Half-Duplex (one way) Communication
- Returns 0 if OK and -1 on error.

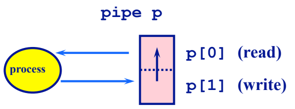


Example: `pipe-yourself.c`

```
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE  16   /* null */

char *msg1="hello, world #1";
char *msg2="hello, world #2";
char *msg3="hello, world #3";

int main()
   {
   char inbuf[MSGSIZE];
   int  p[2], i;

   if( pipe( p ) < 0 )
     {  /* open pipe */
        perror( "pipe" );
        exit( 1 );
     }
```

```
write( p[1], msg1, MSGSIZE );
write( p[1], msg2, MSGSIZE );
write( p[1], msg3, MSGSIZE );

for( i=0; i < 3; i++ )
   {  /* read pipe */
   read( p[0], inbuf, MSGSIZE );
   printf( "%s\n", inbuf );
   }
return 0;
}
```

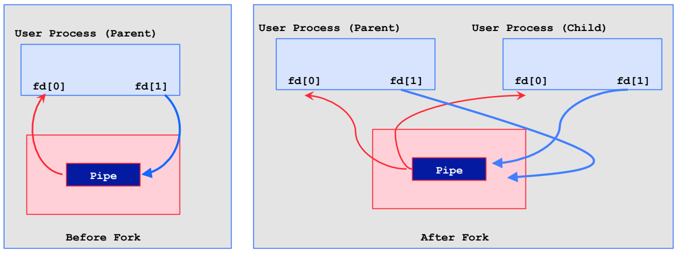
```
{saffron:ingrid:4} pipe-yourself
hello, world #1
hello, world #2
hello, world #3
```



Things to Note

- Pipes uses FIFO ordering: *first-in first-out*.
- Read / write amounts do not need to be the same, but then text will be split differently.
- Pipes are most useful with `fork()` which creates an IPC connection between the parent and the child (or between the parents children)

What Happens After Fork?



- **Design Question:**
  - » Decide on : Direction of data flow – then close appropriate ends of pipe (at both parent and child)

- A forked child
  - » inherits file descriptors from its parent
- pipe()
  - » creates an internal system buffer and two file descriptors, one for reading and one for writing.
- After the pipe call,
  - » the parent and child should close the file descriptors for the opposite direction.
  - » Leaving them open does not permit full-duplex communication.

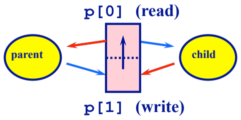
Example: `pipe-fork-close.c`

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#define MSGSIZE  16

char *msg1="hello, world #1";
char *msg2="hello, world #2";
char *msg3="hello, world #3";

int main()
    {
    char inbuf[MSGSIZE];
    int p[2], i, pid;

    if( pipe( p ) < 0 )
       {  /* open pipe */
          perror( "pipe" );
          exit( 1 );
          }
    if( (pid = fork()) < 0 )
       {
       perror( "fork" );
          exit( 2 );
       }
```

```
if( pid > 0 ) /* parent */
    {
    close( p[0] ); /* read link */
    write( p[1], msg1, MSGSIZE );
    write( p[1], msg2, MSGSIZE );
    write( p[1], msg3, MSGSIZE );
    wait( (int *) 0 );
    }
```

```
if( pid == 0 ) /* child */
    {
    close( p[1] ); /* write link */
    for( i=0; i < 3; i++ )
      {
      read( p[0], inbuf, MSGSIZE );
          printf( "%s\n", inbuf );
          }
    }
```

```
return 0;
}
```



Some Rules of Pipes

- Every pipe has a size limit
  - » POSIX minimum is 512 bytes -- most systems makes this figure larger
- read() blocks if pipe is empty *and* there is a a write link open to that pipe
- read() from a pipe whose write() end is closed *and* is empty returns 0 (indicates EOF)
  - » Close write links or read() will never return
- write() to a pipe with no read() ends returns -1 and generates SIGPIPE and errno is set to EPIPE
- write() blocks if the pipe is full or there is not enough room to support the write().
  - » May block in the middle of a write()

Pipes and exec()

How can we code who | sort ?

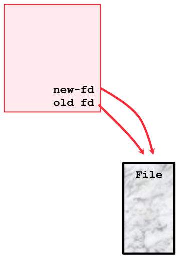
1. Use exec() to start two processes (one runs who the other sort) which share a pipe (exec's start a new program within a copy of the 'parent' process).

2. Connect the pipe to stdin and stdout using dup2().

Duplicate File Descriptors

```
#include <unistd.h>
int dup2( int old-fd, int new-fd );
```

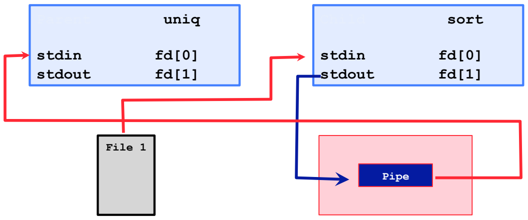
- Set one FD to the value of another.
- new-fd and old-fd now refer to the same file
- if new-fd is open, it is first automatically closed
- Note that dup2() refer to fds not streams
- Example:
  - » `dup2( fd[1], fileno(stdout));`


new-fd
old fd
File

Example : sort < file1.txt | uniq

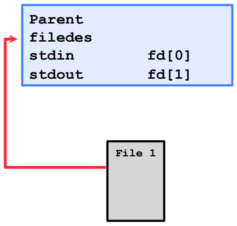
- What does this look like? How would a shell be programmed to process this?
  - » Well we know we need a parent & child to communicate though the pipe!
  - » Parent
  - » Child
  - » We need to open a file and read from it – and then read it as we read it from standard input.

Want: sort < file1.txt | uniq


uniq
stdin fd[0]
stdout fd[1]
sort
stdin fd[0]
stdout fd[1]
File 1
Pipe

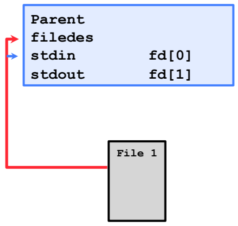
- Want: How do we get there?

Want: "sort < file1 | uniq"


Parent
filedes
stdin fd[0]
stdout fd[1]
File 1

```
fileDES = open( "file1.txt", O_RDONLY );
```

Want: “sort < file1 | uniq”

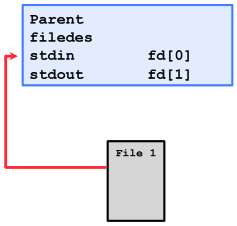


Parent
filedes
stdin fd[0]
stdout fd[1]

File 1

```
fileDES = open( "myfile.txt", O_RDONLY );
dup2( fileDES, fileno( stdin) );
```

Want: “sort < file1 | uniq”

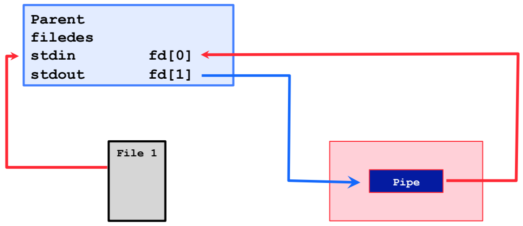


Parent
filedes
stdin fd[0]
stdout fd[1]

File 1

```
fileDES = open( "myfile.txt", O_RDONLY );
dup2( fileDES, fileno( stdin) );
close( fileDES );
```

Want: “sort < file1 | uniq”



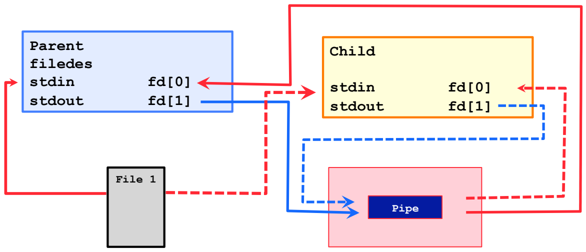
Parent
filedes
stdin fd[0]
stdout fd[1]

File 1

Pipe

```
pipe( fd );
… fork() …
```

Want: “sort < file1 | uniq”



Parent
filedes
stdin fd[0]
stdout fd[1]

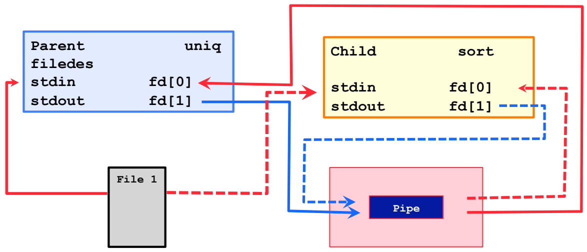
Child
stdin fd[0]
stdout fd[1]

File 1

Pipe

```
fork();
/* now do the plumbing */
```

Want: “sort < file1 | uniq”



Parent uniq
filedes
stdin fd[0]
stdout fd[1]

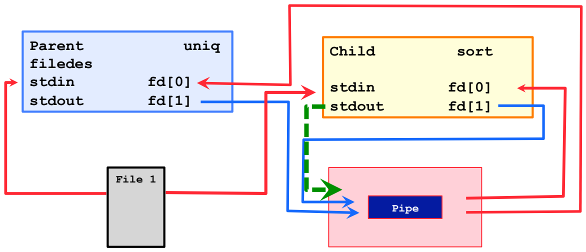
Child sort
stdin fd[0]
stdout fd[1]

File 1

Pipe

```
fork();
/* decide who does what */
```

Want: “sort < file1 | uniq”



Parent uniq
filedes
stdin fd[0]
stdout fd[1]

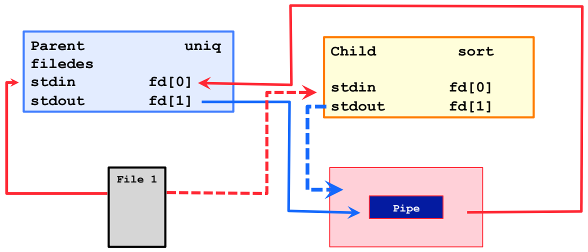
Child sort
stdin fd[0]
stdout fd[1]

File 1

Pipe

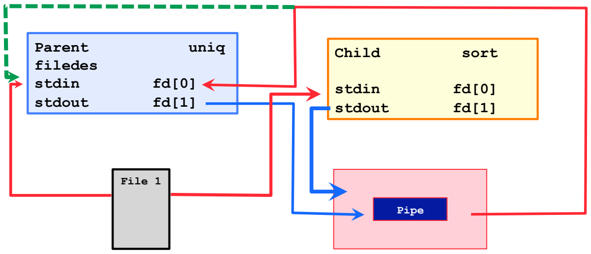
```
/* make writing to the pipe the same
/* as writing to stdout */
dup2( fd[1], fileno(stdout)); /* in green */
```

Want: “sort < file1 | uniq”



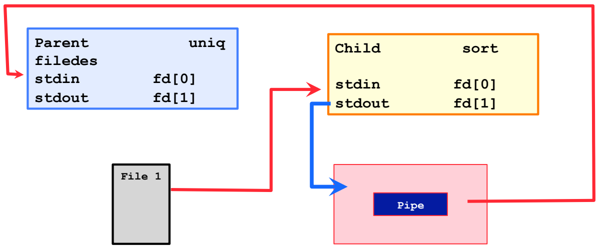
```
close(fd[0]); close(fd[1]);  /* child */
/* leaving the ---- connections for child */
```

Want: “sort < file1 | uniq”



```
dup2(fd[0], fileno(stdin));  /* parent */
/* parent reads from pipe */
```

Want: “sort < file1 | uniq”



```
close(fd[1]); close(fd[0]);  /* parent */
```

Example : “sort < file1 | uniq”

```
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <fcntl.h>

/* child            | parent */
/* sort < file1.txt | uniq */
int main()
{
int status;
int fileDES;
int pipeDES[2];
pid_t pid;

fileDES = open( "myfile.txt", O_RDONLY );
dup2( fileDES, fileno( stdin) );

/* don't need to read via this one anymore */
close( fileDES ) ;

/* create a child that communicate via a pipe */
/* parent reads from pipe, child writes to pipe */
pipe( pipeDES );
```

```
pid = fork();
if( pid < 0 )
  {
  perror("fork");
  exit(1);
  }
else if( pid == 0 ) // child
  {
  close( pipeDES[0] );
  dup2( pipeDES[1], fileno(stdout) );
  close( pipeDES[1]);
  execl( "/usr/bin/sort", "sort", (char *) 0 );
  }
else if( pid > 0 ) // parent
  {
  close( pipeDES[1] );
  dup2( pipeDES[0], fileno(stdin) );
  close( pipeDES[0]);
  execl( "/usr/bin/uniq", "uniq", (char *) 0 );
  }
}
```

Thought questions

- Other ways of designing this task?