# CSCI 4730/6730
# Systems Programming Refresher

**Pipes & FIFOs**



## Outline

- What is a pipe?
- UNIX System review
- Processes (review)
- Pipes
- FIFOs



## What is a Pipe?

- A pipe is a one-way (half-duplex) communication channel which can be used to link processes.
- Can only be used between processes that have a common ancestor
- A pipe is a generalization of the file concept
  - » can use I/O functions like `read()` and `write()` to receive and send data

SVR4 UNIX - uses full duplex pipes (read/write on both file descriptors)



## Example: Shell Pipes

- Example:
  - » `who`
    - – outputs "who" is logged onto the system (e.g. on atlas)
  - » `wc -l hello.txt`
    - – outputs counts the number of lines in the file hello.txt
- You have seen pipes at the UNIX shell level already:
  - » `who | wc -l`
- Shell starts the commands `who` and `wc -l` to run concurrently.
- `|` tells the shell to create a *pipe* to couple standard output of "`who`" to the standard input of "`wc -l`", logically:
  - » `{atlas:maria:195} who > tmpfile`
  - » `{atlas:maria:196} wc -l < tmpfile`
  - » `17`
  - » `{atlas:maria:197}`

Recall < and > redirects standard input and output to a file (i.e., file descriptor 0 for input, and 1 for output.



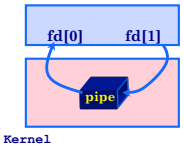
```
{atlas:maria:195} who > tmpfile
{atlas:maria:196} wc -l < tmpfile
       17
{atlas:maria:197} who | wc -l
       17
```

```
{atlas:maria:197} cat tmpfile
luffman    pts/44       Apr 26 10:17    (h198-137-28-67.paws.uga.edu)
imacs      pts/25       Apr 26 08:43    (128.192.4.35)
cai        pts/38       Apr 26 09:15    (user-1121m0h.dsl.mindspring.com)
maher      pts/20       Apr 26 04:57    (adsl-219-4-207.asm.bellsouth.net)
luffman    pts/50       Apr 26 09:52    (h198-137-28-67.paws.uga.edu)
moore      pts/55       Apr 26 10:43    (adsl-219-226-14.asm.bellsouth.net)
tanner     pts/117      Apr 26 08:46    (cmtspool-48.monroeaccess.net)
weaver     pts/106      Apr 26 08:12    (creswell-s218h112.resnet.uga.edu)
dimitrov   pts/39       Apr 26 09:01    (128.192.42.142)
steward    pts/23       Apr 26 09:16    (128.192.101.7)
weaver     pts/12       Apr 26 08:14    (creswell-s218h112.resnet.uga.edu)
dme        pts/6        Apr 25 09:34    (128.192.4.136)
ldeligia   pts/40       Apr 26 10:10    (128.192.4.72)
brownlow   pts/13       Apr 26 09:48    (68-117-218-71.dhcp.athn.ga.charter.com)
misztal    pts/30       Mar 27 10:32    (kat.cs.uga.edu)
james      pts/51       Apr 26 09:28    (adsl-35-8-252.asm.bellsouth.net)
cs4720     pts/107      Mar 27 15:06    (druid)
```

## Programming with Pipes

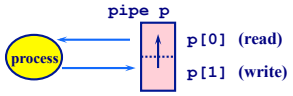

```
#include <unistd.h>
int pipe( int fd[2] );
```

- `pipe()` binds `fd[]` with two file descriptors:
  - » `fd[0]` used to read from pipe (consumer)
  - » `fd[1]` used to write to pipe (producer)
- Returns 0 if OK and -1 on error.
- Example error:
  - » too many fd open already.



# Example: Pipe within a single process

- **Simple example:**
  - » creates a pipe called '`p`'
  - » writes three messages to the pipe (down the pipe)
  - » reads (receives) messages from the pipe
- **Process (user) view:**



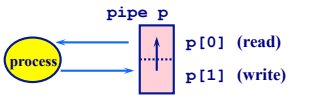



# Example: `pipe-yourself.c`

```
#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;
}
```

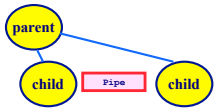
```
{saffron:ingrid:4} pipe-yourself
hello, world #1
hello, world #2
hello, world #3
```






# Things to Note

- **Pipes uses FIFO ordering: *first-in first-out*.**
  - » messages are read in the order in which they were written.
  - » `lseek()` does not work on pipes.
- **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)**




# Example: Pipe between a parent and child

1. **Creates a `pipe`**
2. **Creates a new process via `fork()`**
3. **Parent writes to the pipe (fd 1)**
4. **Child reads from pipe (fd 0)**



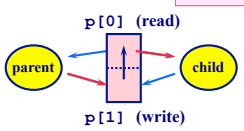
# Example: `pipe-fork.c`

```
#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 */
   {
   write( p[1], msg1, MSGSIZE );
   write( p[1], msg2, MSGSIZE );
   write( p[1], msg3, MSGSIZE );
   wait( (int *) 0 );
   }
```

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

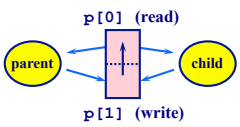
```
return 0;
}
```






# Things to Note

- **Pipes are intended to be unidirectional channels if parent-child processes both read and write on the pipe at the same time confusion.**
- **Best style is for a process to close the links it does not need. Also avoids problems (forthcoming).**



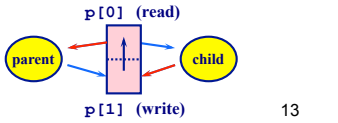
# Example: pipe-fork-close.c

```
#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 [it hangs]
- `read()` from a pipe whose `write()` end is closed *and* is empty returns 0 (indicates `EOF`) [but it doesn't hang]
  - » Lesson Learned: Close write links o/w `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()`

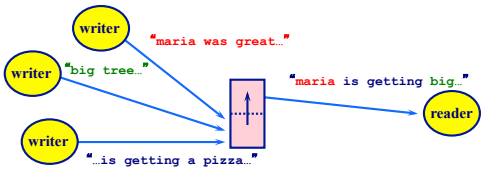


- Perfectly possible to have multiple readers / writers attached to a pipe
  - » can cause confusion



# Example: Several Writers

- Since a `write()` can suspend in the middle of its output then output from multiple writers output may be mixed up (or *interleaved*).






# Avoid Interleaving

- In `limits.h`, the constant `PIPE_BUF` gives the maximum number of bytes that can be output by a `write()` call without any chance of interleaving.
- Use `PIPE_BUF` is there are to be multiple writers in your code.



# Non-blocking `read()` & `write()`

- Problem: Sometimes you want to avoid `read()` and `write()` from blocking.
- Goals:
  - » want to return an error instead
  - » want to poll several pipes in turn until one has data
- Approaches:
  - » Use `fstat()` on the pipe to get #characters in pipe (caveat: multiple readers may give a "race condition")
  - » Use `fcntl()` on the pipe and set it to `O_NONBLOCK`

# Using `fcntl()`

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
        :       .
if( fcntl( fd, F_SETFL, O_NONBLOCK ) < 0 )
        perror("fcntl");
        :
```

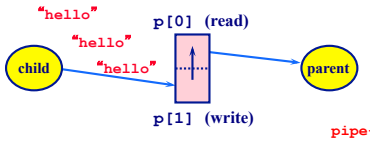
- **Non-blocking write:** On a write-only file descriptor, `fd`, future `writes` will never block
  - » Instead return immediately with a -1 and set `errno` to `EAGAIN`
- **Non-blocking read:** On a read-only file descriptor, `fd`, future `reads` will never block
  - » return -1 and set `errno` to `EAGAIN` unless a flag is set to `O_NDELAY` then return 0 if pipe is empty (or closed)



pipe-nonblocking.c

# Example: Non-blocking with -1 return

- Parent "launches' one child
- Child writes "hello" to parent every 3 seconds (3 times).
- Parent read what the child writes
- Parent does a non-blocking read each second.
- Child does blocking write – … there nothing else to do than write.

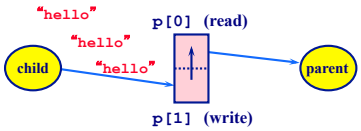





pipe-nonblocking.c

# Example: Non-blocking with -1 return

- Parent's behavior →
  - » Need to check for
    - – No data in pipe
    - – Pipe is closed (EOF)
    - – Errors (in general)
  - » Continuously Read
- Child's behavior →
  - » Just writes and blocks if the pipe is full






pipe-nonblocking.c

# Example: `pipe-nonblocking.c`

```
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

#define MSGSIZE  6
char *msg1="hello";

void parent_read( int p[] );
void child_write( int p[] );

int main()
{
int pfd[2];
if( pipe( pfd ) < 0 )
   {  /* open pipe */
   perror( "pipe" );
   exit( 1 );
   }
```

```
if( fcntl( pfd[0], F_SETFL, O_NONBLOCK ) < 0 )
   { /* read non-blocking */
   perror( "fcntl" );
   exit( 2 );
   }
switch( fork() )
   {
   case -1:    /* error */
         perror("fork" );
         exit(3);
   case 0:     /* child */
         child_write( pfd );
         break;
   default:    /* parent */
         parent_read( pfd );
         break;
   }
return 0;
}
```



pipe-nonblocking.c

# `void parent_read()`

```
void parent_read( int p[] )
{
int nread;
char buf[MSGSIZE];
close( p[1] );  /* write link */
while( 1 )
   {
   nread = read( p[0], buf, MSGSIZE );
   switch( nread )
     {
     case -1:
        if( errno == EAGAIN )
          {
          printf("(pipe empty)\n");
          sleep( 1 );
          break;
          }
```

EAGAIN: No data in pipe (yet) try AGAIN later

```
        else
          {
          perror( "read" );
          exit(4 )
          }
     case 0:
          /* pipe has been closed */
          printf( "End conversation\n" );
          close( p[0] ); /* read fd */
          exit(0);
     default: /* text read */
          printf( "MSG=%s\n", buf );
     } /* switch */
   } /* while */
} /* parent_read */
```



# `void child_write()`

```
void child_write( int p[] )
  {
  int i;
  close( p[0] ); /* read link */
  for( i = 0; i < 3; i++ )
        {
        write( p[1], msg1, MSGSIZE );
        sleep( 3 );
        }
  close( p[1] ); /* write link */
  }
```

```
{saffron} pipe-nonblocking
(pipe is empty)
MSG=hello
(pipe is empty)
(pipe is empty)
(pipe is empty)
MSG=hello
(pipe is empty)
(pipe is empty)
MSG=hello
(pipe is empty)
(pipe is empty)
(pipe is empty)
End of conversation
```

## Non-blocking with 0 error

- If non-blocking `read()` does not distinguish between end-of-input and an empty pipe (e.g. `O_NDELAY` is set ) then can use special message to mean end:
  - » e.g. send "bye" as last message

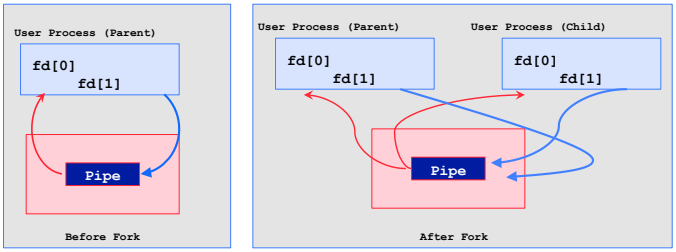


## Review, and reflect

- We created a pipe in a single process, and communicated via the pipe (pipe-yourself.c)
  - » Not pragmatic
- We created a pipe between [a] child(ren) and a parent
  - » Interesting!
  - » Lets look more deeply into what happens after fork?




## What Happens After Fork?




- Design Question:
  - » Need to decide on :
    - – The direction of the data flow – then close appropriate ends of pipe (at both parent and child)



## Some Pipe Rules

- A forked child
  - » Inherits file descriptors from its parent
- pipe()
  - » Creates an
    - – internal system buffer and
    - – 2 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 (that it doesn't need).
  - » Leaving them open does not permit full-duplex communication.



## Pipes and `exec()`

Motivation: How can we code `who | sort` ?

Observation: Writes to stdout and reads from stdin.

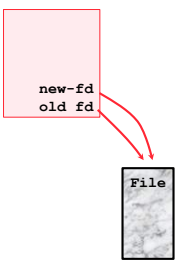
1. Use `exec()` to 'run' code in two different child processes
   - » one runs `who` [child2] and the other `sort` [child1]
   - » `exec` in child(ren) starts a new program within their copy of the 'parent' process
2. Connect the pipe to `stdin` and `stdout` using `dup2()`.



## Manipulating File Descriptors: dup2

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

- Sets one file descriptor to the value of another.,
  - » Existing file descriptor, `old-fd`, is duplicated onto `new-fd` so that they refer to the same file
- If `new-fd` already exists, it will be closed first.



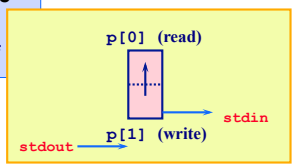

Example:

```
» dup2( fd[1], fileno(stdout));
```

Pipeline.c

# Connecting pipes with stdin & stdout

- Connect the write end of a pipe to stdout
  - » int p[2];
    pipe( p );
    dup2( p[1], STDOUT_FILENO );
- Connect the read end of pipe to stdin
  - » dup2( p[0], STDIN_FILENO );

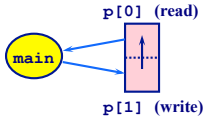



Caveat: Beware of hanging on the 'pipe'
Solution: Close all file descriptors that comprise its pipes so that the pipes don't hang.



# Four Stages to who | sort

1. main() creates a pipe

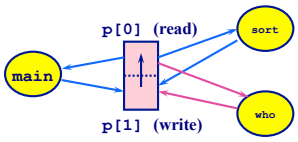



```
int fds[2];
pipe( fds ) ;   /* no error checks */
```



# Four Stages to who | sort

1. main() creates a pipe
2. main() forks twice to make two children that inherits the pipes descriptors



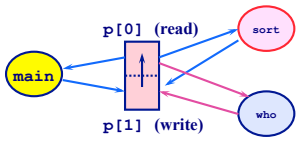

See code… whosort.c



# Four Stages to who | sort

1. main() creates a pipe
2. main() forks twice to make two children and inherits the pipes descriptors
3. Child: Couple standard output to write end
4. Child: Couple standard input to read end
5. Close the pipe links which are not needed

```
dup2( p[0] , STDIN_FILENO );
```

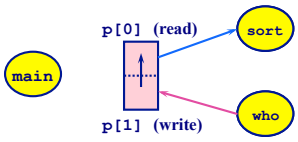



```
dup2( p[1] , STDOUT_FILENO );
```



# Four Stages to ps | sort

1. main() creates a pipe
2. main() forks twice to make two children and inherits the pipes descriptors
3. Close the pipe links which are not needed
4. Replace children by programs using exec()






# who | sort : whosort.c

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main()
  {
  int p[2];
  pipe( p );   /* no error checks */

  if( fork() == 0 )
        { /* 1st child */
        /* fds[0]/stdin --> sort */
        dup2( p[0] , STDIN_FILENO );
        close( p[ 1 ] );
        execlp( "sort", "sort", (char *)
   0 );
        }
  else
        { /* parent - create
```

```
   if( fork() == 0 )
        { /* 2nd child */
        /* who --> fds[1]/stdout --> sort */
        dup2( p[1] , STDOUT_FILENO );
        close( p[ 0 ] );
        execlp( "who", "who", (char *) 0 );
        }
   else
        { /* parent closes all links */
        close( p[ 0 ] );
        close( p[ 1 ] );

        wait( (int *) 0 );
        wait( (int *) 0 );
        } /* else parent second child */
   } /* else parent first child */
 return 0;
 }
```

```
{atlas:maria:169} who-sort
aguda     dtremote     Apr 25 15:46    (128.192.101.83:0)
ananda    pts/25       Apr 25 10:52    (128.192.4.101)
anyanwu   pts/24       Apr 25 11:30    (dhcp183)
bralley   dtremote     Apr 25 15:38    (128.192.101.84:0)
```

## Limitations of Pipes

- Processes using a pipe must come from a common ancestor:
  - » e.g. parent and child
  - » cannot create general servers like print spoolers or network control servers since unrelated processes cannot use it
- Pipes are not permanent
  - » they disappear when the process terminates
- Pipes are one-way:
  - » makes fancy communication harder to code
- Readers and writers do not know each other.
- Pipes do not work over a network

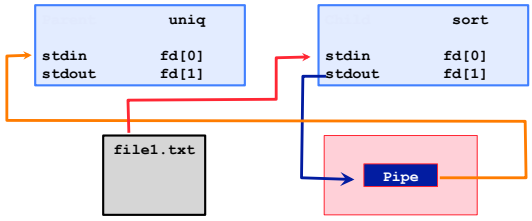


## Something more interesting…

- 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`

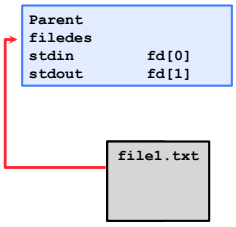



- Want: How do we get there?

Step 1: We want to read from the file



## Want: "`sort < file1 | uniq`"

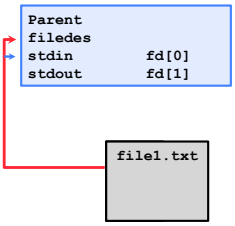



```
fileDES = open( "file1.txt", O_RDONLY );
```

Step 2: Read from the file like it is from stdin



## Want: "`sort < file1 | uniq`"

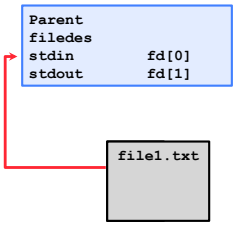



```
fileDES = open( "file1.txt", O_RDONLY );
dup2( fileDES, fileno( stdin ) );
```

Step 3: Don't need fileDES anymore …



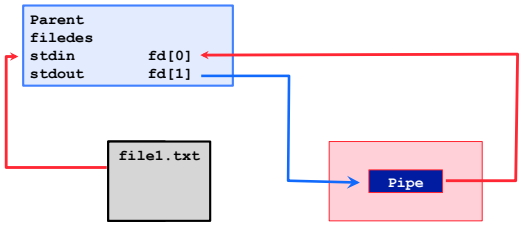
## Want: "`sort < file1 | uniq`"




```
fileDES = open( "file1.txt", O_RDONLY );
dup2( fileDES, fileno( stdin) );
close( fileDES );
```

Step 4: Hairier – now we deal with the pipe…

## Want: "sort < file1 | uniq"




UGH Hairy!

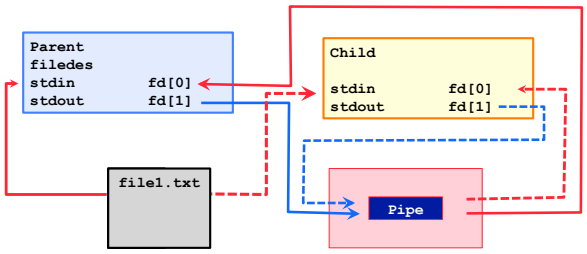
```
pipe( fd );
… fork() …
```



- No : Not really that bad
  - » Just need to create the pipe then create a child (or parent) that is on the other side of the pipe!
    - – Then do the plumbing:
      - Reroute stdin/stdout appropriately….
      - AND THAT IS IT!



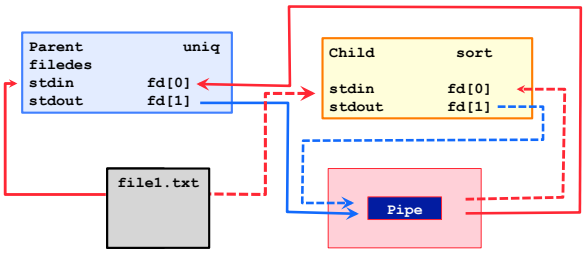
## Want: "sort < file1 | uniq"




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



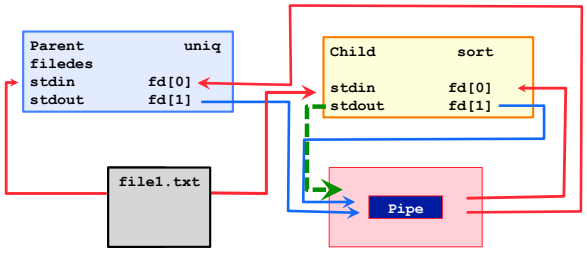
## Want: "sort < file1 | uniq"




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



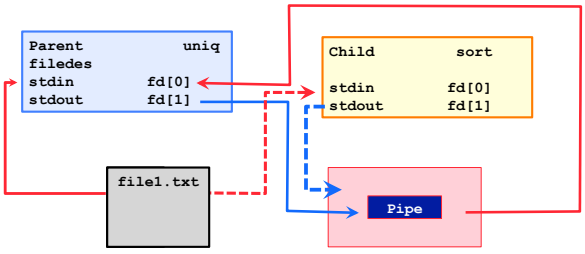
## Want: "sort < file1 | uniq"




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

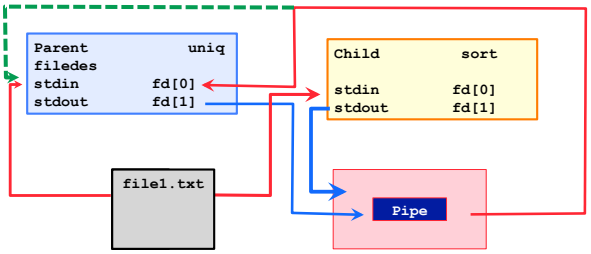


## Want: "sort < file1 | uniq"




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

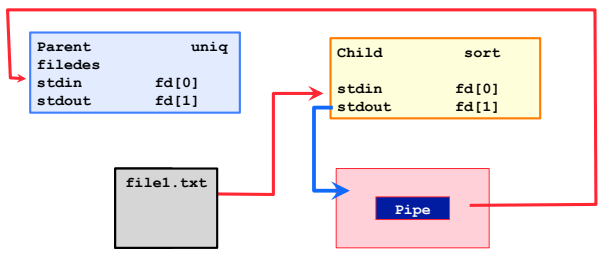
## Want: "sort < file1 | uniq"




```
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?
- Is this the only way?

End of Tutorial



## What are FIFOs/Named Pipes?

- Similar to pipes (as far as `read/write` are concerned, e.g. FIFO channels), but with some additional advantages:
  - Unrelated processes can use a FIFO.
  - A FIFO can be created separately from the processes that will use it.
  - FIFOs look like files:
    - have an owner, size, access permissions
    - open, close, delete like any other file
    - permanent until deleted with `rm`



## Creating a FIFO

- UNIX `mkfifo` command:

```
$ mkfifo fifo1
```

Default mode is the difference: 0666 - umask value

- On older UNIXs (origin ATT UNIX), use `mknod`:

```
$ mknod fifo1 p
```

`p` means FIFO

- Use `ls` to get information:

```
$ ls -l fifo1
prw-rw-r--  1  maria  maria  0 Oct 23  11.45  fifo1|
```

## Using FIFOs: FIFO Blocking

- FIFOs can be read and written using standard UNIX commands connected via "<" and ">" standard input or output

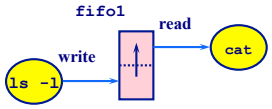
- If there are no writers then a read:
  e.g. `cat < fifo1`
  will block until there is 1 or more writers.

- If there are no readers then a write:
  e.g. `ls -l > fifo1`
  will block until there is 1 or more readers.



## Reader / Writer Example

```
$ cat < fifo1 &
[1] 22341
$ ls -l > fifo1; wait
total 17
prw-rw-r--  1  maria  usr  0  Oct 23  11.45  fifo1   :
[1] + Done              cat < fifo1
$
```




1. Output of `ls -l` is written down the FIFO
2. Waiting `cat` reads from the FIFO and display the output
3. `cat` exits since read returns 0 (the FIFO is not open for writing anymore and 0 is returned as EOF)

`wait` - causes the shell to wait until `cat` exits before redisplaying the prompt.



## Creating a FIFO in C

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo( const char *pathname, mode_t mode );
```

- Returns 0 if OK, -1 on error.
- `mode` is the same as for `open()` - and is modifiable by the process' `umask` value
- Once created, a FIFO must be opened using `open()`

Note: the significant difference between programming with pipes versus FIFOs is initialization.



## Outline on how to program with FIFOs

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define MSGSIZE 63

int main()
{
int fd;
char msgbuf[MSGSIZE+1];

mkfifo( "/tmp/mariafifo", 0666 );
fd = open( "/tmp/mariafifo", O_WRONLY );
.
.
}
```



## Two Main Uses of FIFOs

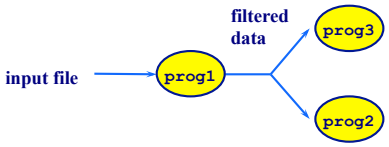
1. Used by shell commands to pass data from one shell pipeline to another without using temporary files.

2. Create client-server applications on a single machine.



## Shell Usage

- *Example*: Process a filtered output stream twice – i.e., pass filtered data to two separate processes:

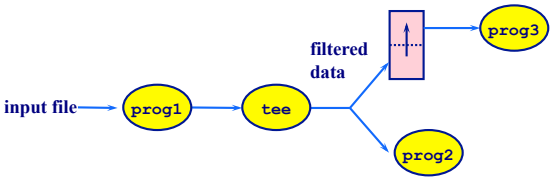



- In contrast to regular pipes, FIFOs allows non-linear connections between processes such as the above, since FIFO's are pipes with names.

# Implementation


UNIX's `tee()` copies standard input to both its

- ~ standard input and to the
- ~ file named on its command line

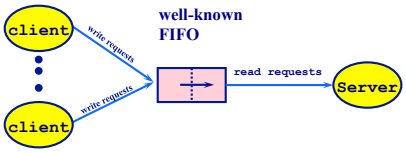



```
$ mkfifo fifo1
$ prog3 < fifo1 &
$ prog1 < infile | tee fifo1 | prog2
```



# A Client-Server Application

- Server contacted by numerous clients via a well-known FIFO

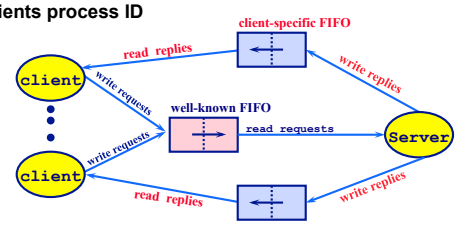



- How are replies from the server sent back to each client?



# Client-Server FIFO Application

- **Problem:** A single FIFO (as before) is not enough.
- **Solution:** Each client send its PID as part of its message. Which the uses to create a speciaal 'reply' FIFO for each client
  - » e.g. `/tmp/serv1.XXXX` where `XXXX` is replaced with the clients process ID






# Problems

- The server does not know if a client is still alive
  - » may create FIFOs which are never used
  - » client terminates before reading the response (leaving FIFO w/ one writer and no reader)
- Each time number of clients goes from 1 client to 0 the clients server reads "0"/EOF on the well-known FIFO, if it is set to read-only.
  - » Common trick is to have the server open the FIFO as read-write (see text book for more details)



# Programming Client-server Applications

- First we must see how to `open` and `read` a FIFO from within C.
- Clients will write in non-blocking mode, so they do not have to wait for the server process to start.



# Opening FIFOs

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
      :
fd = open( "fifo1", O_WRONLY );
      :
```

- A FIFO can be opened with `open()` (most I/O functions work with pipes).

## Blocking open()

- An `open()` call for *writing* will block until another process opens the FIFO for *reading*.
  - this behavior is not suitable for a client who does not want to wait for a server process before sending data.
- An `open()` call for *reading* will block until another process opens the FIFO for *writing*.
  - this behavior is not suitable for a server which wants to poll the FIFO and continue if there are no readers at the moment.



## Non-blocking open()

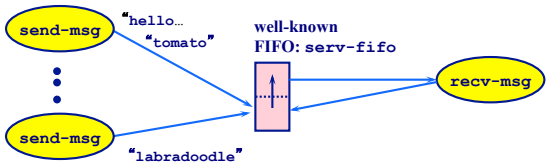
```
if ( fd = open( "fifo1", O_WRONLY | O_NONBLOCK)) < 0 )
      perror( "open FIFO" );
```

- `opens` the FIFO for writing
- returns -1 and `errno` is set to `ENXIO` if there are no readers, instead of blocking.
- Later `write()` calls will also not block.



## Example: send-msg, recv-msg

- `opens` the FIFO for writing
- returns -1 and `errno` is set to `ENXIO` if there are no readers, instead of blocking.
- Later `write()` calls will also not block.






## Some Points

- `recv-msg` can read and write;
  - otherwise the program would block at the open call and
  - avoids responding to reading a "return of 0" when the number of send-msg processes goes from 1 to 0 (and the FIFO is empty) O_RDWR - ensures that at least one process has the FIFO open for writing (i.e. recv-msg itself) so read will always block until data is written to the FIFO
- `send-msg` sends fixed-size messages of length `PIPE_BUF` to avoid interleaving problems with other `send-msg` calls. It uses non-blocking.
- `serv_fifo` is globally known, and previously created with `mkfifo`



## send-msg.c & recv-msg.c

```
{saffron:ingrid:3} recv-msg
serv_fifo: No such file or directory
{saffron:ingrid:4} mkfifo serv_fifo
{saffron:ingrid:5} recv-msg &
[1] 792
Msg: hi
Msg: potato
Msg: pizza
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>

#define SF "serv_fifo"
```

```
{saffron:ingrid:3} send-msg "hi" "potato..." &
[1] 794
{saffron:ingrid:4} send-msg "pizza" &
[2] 795
[1] - Done                    send-msg "hi" "potato"
[2] - Done                    send-msg "pizza"
```



## send-msg.c

```
int main( int argc, char *argv[] )
  {
  int fd, i;
  char msgbuf[PIPE_BUF];

  if( argc < 2 )
        {
        printf( "Usage: send-msg msg...\n" );
        exit( 1 );
        }
  if( (fd = open( SF, O_WRONLY | O_NONBLOCK )) < 0)
        { perror( SF ); exit( 1 ); }
  for( i = 1; i < argc; i++ )
        {
        if( strlen( argv[i] ) > PIPE_BUF - 2 )
                printf( "Too long: %s\n", argv[i] );
        else
                {
                make_msg( msgbuf, argv[i] );
                write( fd, msgbuf, PIPE_BUF );
                }
        }
```

```
/* put input message into mb[] with '$'
 * and padded with spaces */
void make_msg( char mb[], char
    input[] )
  {
  int i;
  for( i = 1; i < PIPE_BUF-1; i++ )
        mb[i] = ' ';
  mb[i] = '\0';
  i = 0;
  while( input[i] != 0 )
        {
        mb[i] = input[i];
        i++;
        }
  mb[i] = '$';
  } /* make_msg */
```

```
close( fd );
return 0;
} /* end main */
```

## recv-msg.c

```
int main( int argc, char *argv[] )
  {
  int fd, i;
  char msgbuf[PIPE_BUF];

  if( (fd = open( SF, O_RDWR )) < 0 )
        {
        perror( SF );
        exit( 1 );
        }
  while( 1 )
        {
        if( read( fd, msgbuf, PIPE_BUF ) < 0 )
                {
                perror( "read" );
                exit( 1 );
                }
        print_msg( msgbuf );
        }
  close( fd );
  return 0;
  } /* end main */
```

```
/* print mb[] up to the  '$' marker */
void print_msg( char mb[] )
  {
  int i = 0;
  printf( "Msg: " );
  while( mb[i] != '$' )
        {
        putchar( mb[i] );
        i++;
        }
  putchar( '\n' );
  } /* make_msg */
```



## Things to Note about recv-msg

- `open()` is blocking, so `read()` calls will block when the pipe is empty
- `open()` uses `O_RDWR` not `O_RDONLY`
  - this means there is a write link to the FIFO even when there are no `send-msg` processes
  - this means that a `read()` call will block even when there are no `send-msg` processes, instead of returning 0.