12-UNIX

Inter-Process Communication (IPC)
Pipe

Chapter 43-44

Overview of IPC in Linux (Chapter 43)

- communication
  - data transfer
  - shared memory
- signal
  - standard signal
  - realtime signal
- synchronization
  - semaphore
  - file lock
  - mutex (threads)
  - condition variable (threads)

Based on chart LPI page 878.

Communication expanded

- data transfer
  - byte stream
    - pipe
    - FIFO
    - stream socket
  - pseudoterminal
  - message
    - System V message queue
    - POSIX message queue
    - datagram socket
- shared memory
  - System V shared memory
  - POSIX shared memory
  - memory mapping
    - anonymous mapping
    - mapped file

Synchronization Expanded

- Semaphore
  - System V semaphore
  - POSIX semaphore
    - Names
    - unnamed
- file lock
  - “record” lock (*fcntl()*)
  - File lock (*flock()*)
- mutex (threads)
- condition variable (threads)

We know:

- What is a process?
- How a process is created?
- How a process can replace its image by running a new program ?
- Parent Process waits for a Child Process.

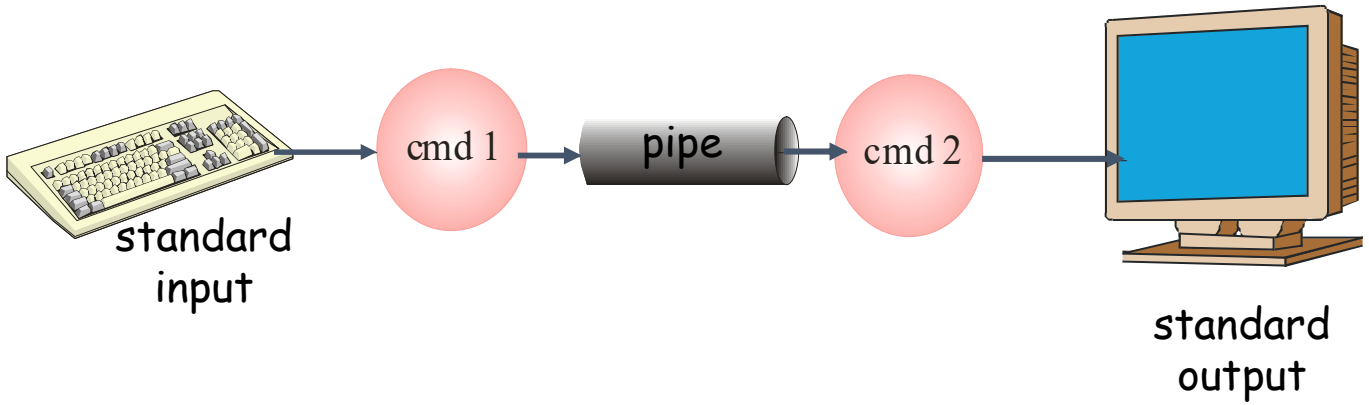
Next:

How do we pass useful information between processes?

How do we synchronize processes?

Idea of Pipe

cmd1 | cmd2



Pipe connects a data flow
from one process to another

Pipelines: ls –l | sort –k5 –n
The starting simple *ls*

athena.ecs.csus.edu - PuTTY

```
[bielr@athena csc60]> ls -l
total 544
-rw-------  1 bielr faccsc      2 Dec 21 12:58 >
-rwx------  1 bielr faccsc   5074 Mar 16 14:08 a.out*
drwx------  2 bielr faccsc   4096 Apr 19 13:07 ClassExamples/
-rw-------  1 bielr faccsc    162 Apr 11 18:16 lsls
-rw-------  1 bielr faccsc    138 Dec 22 09:39 lsout
drwx------ 16 bielr faccsc   4096 Apr 24 10:31 mywork/
drwx------  6 bielr faccsc   4096 Dec 18 15:58 myworkf16/
drwx------  8 bielr faccsc   4096 Dec 16 15:07 myworkS16/
-rwx------  1 bielr faccsc   6438 Sep 19  2016 reverse*
-rw-------  1 bielr faccsc    993 Sep 16  2016 reverse1.c
drwx------  4 bielr faccsc   4096 Apr 23 10:36 student/
-rw-------  1 bielr faccsc    527 Nov 16 08:35 testScript.txt
-rw-------  1 bielr faccsc 235289 Apr 17  2016 tlpi-160401-dist.tar.gz
drwx------ 48 bielr faccsc   4096 Nov 10 09:26 tlpi-dist/
-rw-------  1 bielr faccsc 252898 Sep 21  2016 trylab1.txt
-rw-------  1 bielr faccsc     12 Dec 22 09:40 wcout
[bielr@athena csc60]>
```

Note: -k5 means sorting column 5; -n means sorting by numerical value

Pipelines: ls –l | sort –k5 –n
The result of the sort

athena.ecs.csus.edu - PuTTY

```
[bielr@athena csc60]> ls -l | sort -k5 -n
total 544
-rw-------  1 bielr faccsc      2 Dec 21 12:58 >
-rw-------  1 bielr faccsc     12 Dec 22 09:40 wcout
-rw-------  1 bielr faccsc    138 Dec 22 09:39 lsout
-rw-------  1 bielr faccsc    162 Apr 11 18:16 lsls
-rw-------  1 bielr faccsc    527 Nov 16 08:35 testScript.txt
-rw-------  1 bielr faccsc    993 Sep 16  2016 reverse1.c
drwx------ 16 bielr faccsc   4096 Apr 24 10:31 mywork/
drwx------  2 bielr faccsc   4096 Apr 19 13:07 ClassExamples/
drwx------ 48 bielr faccsc   4096 Nov 10 09:26 tlpi-dist/
drwx------  4 bielr faccsc   4096 Apr 23 10:36 student/
drwx------  6 bielr faccsc   4096 Dec 18 15:58 myworkf16/
drwx------  8 bielr faccsc   4096 Dec 16 15:07 myworkS16/
-rwx------  1 bielr faccsc   5074 Mar 16 14:08 a.out*
-rwx------  1 bielr faccsc   6438 Sep 19  2016 reverse*
-rw-------  1 bielr faccsc 235289 Apr 17  2016 tlpi-160401-dist.tar.gz
-rw-------  1 bielr faccsc 252898 Sep 21  2016 trylab1.txt
[bielr@athena csc60]>
```

Note: -k5 means sorting column 5; -n means sorting by numerical value

Process Pipes (Formally)

A **pipe** is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process.

The data is handled in a first-in, first-out (FIFO) order.

The pipe has no name, so it can only be used by the process that created it **and by descendants that inherit the file descriptors on fork**( ).

Process Pipes

A pipe has to be open at both ends simultaneously.

If you read from a pipe file that doesn't have any processes writing to it (perhaps because they have all closed the file, or exited), the read returns end-of-file (EOF).

Using normal blocking (BLOCK) reads however, the read will block if the pipe is empty.

Writing to a pipe that doesn't have a reading process is treated as an error (ERROR) condition; it generates a SIGPIPE signal, and fails with error code EPIPE if the signal is handled or blocked.

Process Pipes

Pipes do not allow file positioning (i.e. lseek). Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end. System keeps track of last read/write location.

Figure 5.1 Conceptual data access using a pipe.

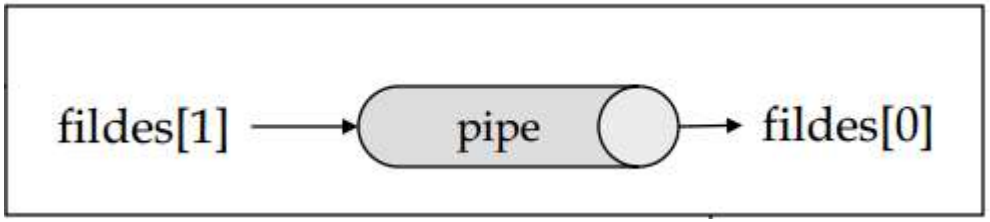
A common use of pipes is to send data to or receive data from a program being run as a sub-process.

The Pipes system call (2)

The pipe call is a **system call** (man 2 pipe)

#include <unistd.h>

int **pipe**(int fildes[2]);



returns
0 if successful
-1 if the call fails

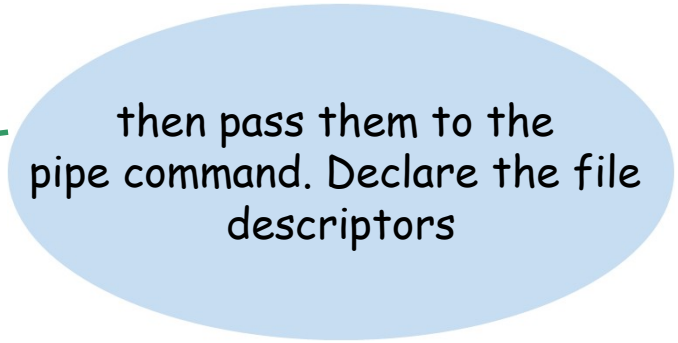
The pipe call fills in two file descriptors
fildes[0] is the file descriptor for reading from the pipe
fildes[1] is the file descriptor for writing to the pipe

It is easy to remember which comes first if you remember that 0 is standard in and 1 is standard out.

You read from fildes[0] (think STDIN==0)
You write to fildes[1] (think STDOUT=1)

The Pipe - Declaration

The following code fragment creates an un-named pipe:




```
int fd[2];

if (pipe(fd) == -1)
    perror("Failed to create pipe…");
```

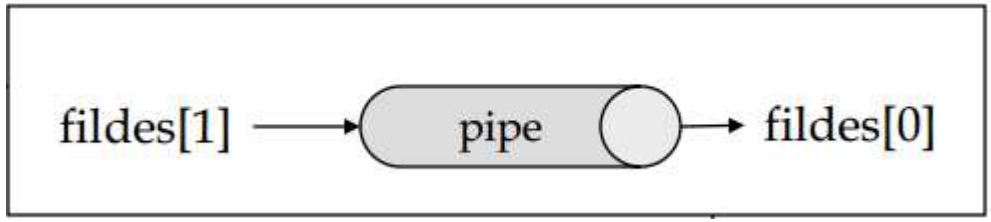
Using Pipe

In typical use, a process creates a pipe just before it forks one or more child processes.

The pipe is then used for communication either between the parent or child processes,
or between two sibling processes.

Pipe Example

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#define MAX_LENGTH 100
int main(int argc, char ** argv) {
  int fildes[2];
  char result[] = "";
  pipe(fildes);
  if (fork())
    write(fildes[1], "I am writing into the pipe", MAX_LENGTH);
  else {
    read(fildes[0], result, MAX_LENGTH);
    printf("I read <<%s>> from the pipe.\n", result);
   }
}
```



Output: I read <<I am writing into the pipe>> from the pipe.

Pipe and Fork (LPI – Page 893)

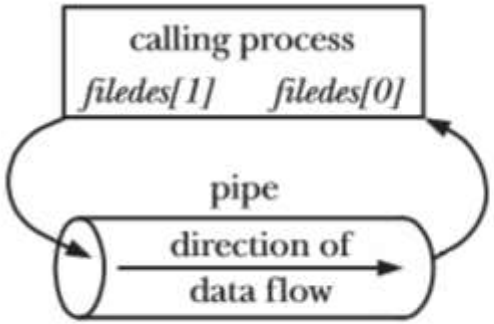
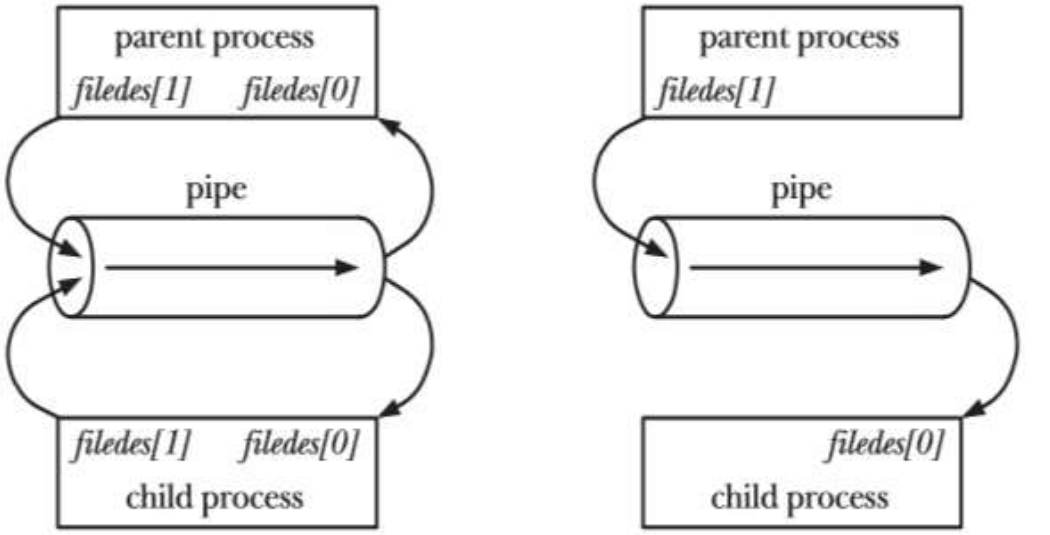


Figure 44-2: Process file descriptors after creating a pipe



a) After *fork()* b) After closing unused descriptors

Figure 44-3: Setting up a pipe to transfer data from a parent to a child

Some good reasons for closing unused file descriptors (See LPI - page 894-895)

- The reading process closes write descriptor in order that it can see “end-of-file” status (if not, instead it sees “block waiting” for data – due to kernel’s indication that there some write descriptor is still opened)

- If the writing process does not close read descriptor, even after the read process closes the read descriptor, it can still write to the pipe’s until it is full. Once the pipe is full, it will block the write process indefinitely.

- Free resources to be used by other processes.

Race Condition (formally)

An unanticipated execution ordering of concurrent flows that results in undesired behavior is called a race condition—a software defect and frequent source of vulnerabilities.

Race Condition - Example

```
char c;
pid_t pid;
int fd = open(filename, O_RDWR);
if (fd == -1) {
  /* Handle error */
}
read(fd, &c, 1);
printf("root process:%c\n",c);

pid = fork();
if (pid == -1) {
  /* Handle error */
}

if (pid == 0) { /*child*/
  read(fd, &c, 1);
  printf("child:%c\n",c);
}
else { /*parent*/
  read(fd, &c, 1);
  printf("parent:%c\n",c);
}
```

Filename = “text.txt”
(Contents = “abc”)

Possible Output:

(1)
root process: a
parent: b
child: c

Or

(2)
root process: a
child: b
parent: c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main (void){
   char c;
   pid_t pid;

   int fd = open("text.txt", O_RDWR);
   if (fd == -1) {
            perror("Error on open");
   }
   read(fd, &c, 1);
   printf("root process:%c\n",c);

   pid = fork();
   if (pid == -1) {
            perror("Error on Fork");
   }
```

```
if (pid == 0) { /*child*/
            read(fd, &c, 1);
            printf("child:%c\n",c);
   }
   else { /*parent*/
            read(fd, &c, 1);
            printf("parent:%c\n",c);
   }

   return(EXIT_SUCCESS);
}
```

```
[bielr@athena ClassExamples]> race
root process:a
parent:b
child:c
```

Code that really works

Using pipe as a method for synchronization (1 of 2)

Listing 44-3: Using a pipe to synchronize multiple processes

pipes/pipe_sync.c

```
#include "curr_time.h"                  /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];                         /* Process synchronization pipe */
    int j, dummy;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);                   /* Make stdout unbuffered, since we
                                               terminate child with _exit() */
    printf("%s  Parent started\n", currTime("%T"));

①   if (pipe(pfd) == -1)
        errExit("pipe");

    for (j = 1; j < argc; j++) {
②       switch (fork()) {
        case -1:
            errExit("fork %d", j);

        case 0: /* Child */
            if (close(pfd[0]) == -1)        /* Read end is unused */
                errExit("close");

            /* Child does some work, and lets parent know it's done */

            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
                                            /* Simulate processing */
            printf("%s  Child %d (PID=%ld) closing pipe\n",
                    currTime("%T"), j, (long) getpid());
③           if (close(pfd[1]) == -1)
                errExit("close");

            /* Child now carries on to do other things... */

            _exit(EXIT_SUCCESS);
```

Enlarged version following

Using pipe as a method for synchronization (2 of 2)

```
            default: /* Parent loops to create next child */
                break;
            }
        }

        /* Parent comes here; close write end of pipe so we can see EOF */

④       if (close(pfd[1]) == -1)                /* Write end is unused */
            errExit("close");

        /* Parent may do other work, then synchronizes with children */

⑤       if (read(pfd[0], &dummy, 1) != 0)
            fatal("parent didn't get EOF");
        printf("%s  Parent ready to go\n", currTime("%T"));

        /* Parent can now carry on to do other things... */

        exit(EXIT_SUCCESS);
    }
```

Using pipe as a method for synchronization

(1 of 4)

```
/* LPI page 897, Listing 44-3 */
#include "curr_time.h"
#include "tlpi_hdr.h"

int main(int agrc, char *argv[ ]) }
   int pfd[2];                    /* Process synchronization pipe */
   int j, dummy;

   if (argc < 2 || strcmp(argv[1] == "--help") == 0)
         usageErr("%s sleep-time...\n", argv[0]);
   setbuf(stdout, NULL);     /* Make stdout unbuffered, since we
                                terminate child with _exit() */
   printf("%s Parent started\n", currTimec));

   if (pipe(pfd) == -1)          /* build the pipe before creating child process */
         errExit("pipe");        /* errExit is a textbook function, not a system function
                                                                                  */
```

Using pipe as a method for synchronization

(2 of 4)

```
for (j = 1; j < argc; j++) {
    switch (fork() ) {          /* Create the child process */
        case -1:
            errExit("fork &d", j);

        case 0:
            if (close(pfd[0] ) == -1    /* Read end is unused */
                    errExit("close");    /* errExit is a textbook function, not a system
                                            function */

            /* Child does some work, and lets parent know it is done */

            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));    /* Simulate
                                                                   processing */

            printf("%s Child %d (PID=$ld) closing pipe \n",
                    currTime("%T"), j, (long) getpid());
```

Using pipe as a method for synchronization

(3 of 4)

```
            if (close(pfd[1]) == -1) /* Each child inherits a fd for the write end of
                                        pipe and closes this fd once it has
                                        completed its action */
                errExit("close");       /* errExit is a textbook function, not a
                                           system function */

            /* Child now carries on to do other things... */

                _exit(EXIT_SUCCESS);

        default:        /* parent loops to create next child */
            break;
    }           /* end of the switch */
}               /* end of the for loop */
```

Using pipe as a method for synchronization (4/4)

```
   /* Parent comes here;  close write end of pipe so we can see EOF */
   /* Note that closing the unused write end of the pipe in the parent is
      essential to the correct operation of this technique;  otherwise, the
      parent would block forever when trying to read from the pipe. */
   if (close(pfd[1]) == -1)            /* Write end is unused */
         errExit("close");
   /* Parent may do other work, then synchronizes with children */
   /* After all the children have closed their file descriptors for the write end
      of the pipe, the parent's read() from the pipe will complete, returning
      end-of-file (0). */
   if (read(pfd[0], &dummy, 1) != 0)
         fatal("parent didn't get EOF");
   printf("%s  Parent ready to go\n", currTime("%T"));
   /* Parent can now carry onto do other things... */
   exit(EXIT_SUCCESS);
}
```

Important Notes:

Communications buffers such as pipes can be empty if all of the information previously written has been read. The empty buffer is not an end-of-file condition. Rather, it reflects the asynchronous nature of inter-process communication.
A read call will normally block, waiting for data to become available.

However, a read on a pipe that has the other end closed for writing will not block, but will return a zero. This allows the reading process to detect the pipe equivalent to an end-of-file condition and react accordingly.

exit Function

- **exit()**
  - The exit() function causes normal process termination and the value of status & 0377 is returned to the parent.
  - All open stdio(3) streams are **flushed and closed**.
    (C standard library - from man 3 exit)
    (informally)
    clean shutdown, flush streams, close files, etc

_*exit* Function

- **_exit()**

  - The function _exit() terminates the calling process "immediately".

  - Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, init, and the process's parent is sent a **SIGCHLD** signal.

  - (System call - from man 2 _exit)

    (informally) drop out, files are closed but streams are not flushed

exit vs. *_exit* Functions

The two functions terminate normally short of return:

Note:

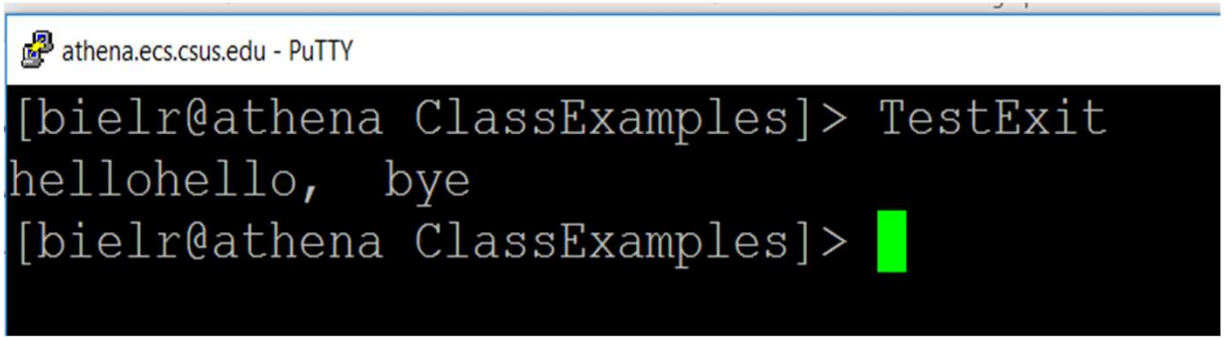
Child and parent could have buffers with a copy of the unflushed data.

If both call exit(), the pending stdio buffers to be **flushed twice**.

Thus, child should call **_exit()** instead.

Example 1

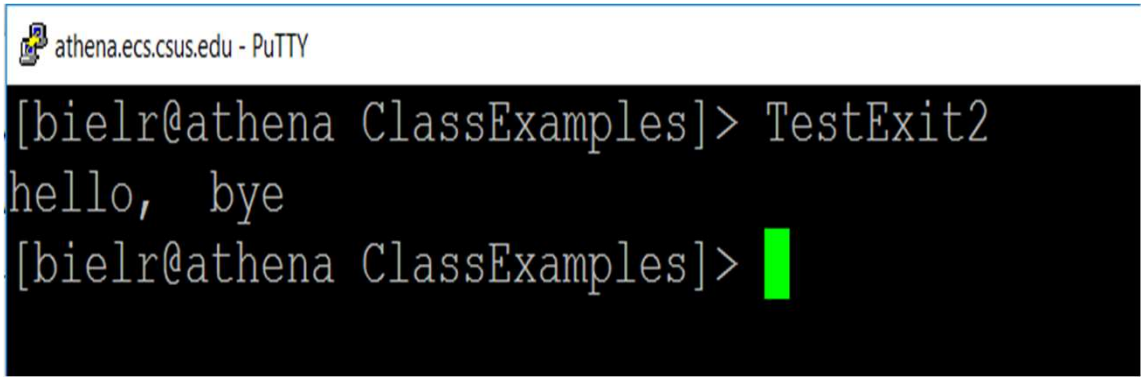
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main () {
  int status;
  printf("hello");
  pid_t pid = fork();
  if (pid == 0) {
     sleep (2);
     exit(0);
  } else {
     wait(&status);
     printf(“,  bye\n");
  }
  exit(0);
}
```



What is going on here? Why there are two hellos displayed ?

Example 2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main () {
  int status;
  printf("hello");
  pid_t pid = fork();
  if (pid == 0) {
     sleep (2);
     _exit(0);
  } else {
     wait(&status);
     printf(“,  bye\n");
  }
  exit(0);
}
```



Make more sense!

Pipelines …

Getting the idea (1 of 2)

What do these commands do?

```
$ ls -l > myfile.txt
$ sort -k5 -n < myfile.txt
```

Redirect standard output of the ls command to the file myfile.txt

Sort is a filter. It normally takes input from stdin and outputs to stdout. In this case we redirected its standard input to come from myfile.txt

Getting the ideas (2 of 2)

We can achieve the same effect by using a pipe.
This eliminates the intermediate file *myfile.txt*

```
ls -l | sort -k5 -n
```

Pipelines: ls –l | sort –k5 –n
The starting simple *ls*

athena.ecs.csus.edu - PuTTY

```
[bielr@athena csc60]> ls -l
total 544
-rw-------  1 bielr faccsc      2 Dec 21 12:58 >
-rwx------  1 bielr faccsc   5074 Mar 16 14:08 a.out*
drwx------  2 bielr faccsc   4096 Apr 19 13:07 ClassExamples/
-rw-------  1 bielr faccsc    162 Apr 11 18:16 lsls
-rw-------  1 bielr faccsc    138 Dec 22 09:39 lsout
drwx------ 16 bielr faccsc   4096 Apr 24 10:31 mywork/
drwx------  6 bielr faccsc   4096 Dec 18 15:58 myworkf16/
drwx------  8 bielr faccsc   4096 Dec 16 15:07 myworkS16/
-rwx------  1 bielr faccsc   6438 Sep 19  2016 reverse*
-rw-------  1 bielr faccsc    993 Sep 16  2016 reverse1.c
drwx------  4 bielr faccsc   4096 Apr 23 10:36 student/
-rw-------  1 bielr faccsc    527 Nov 16 08:35 testScript.txt
-rw-------  1 bielr faccsc 235289 Apr 17  2016 tlpi-160401-dist.tar.gz
drwx------ 48 bielr faccsc   4096 Nov 10 09:26 tlpi-dist/
-rw-------  1 bielr faccsc 252898 Sep 21  2016 trylab1.txt
-rw-------  1 bielr faccsc     12 Dec 22 09:40 wcout
[bielr@athena csc60]>
```

Note: -k5 means sorting column 5; -n means sorting by numerical value

Pipelines: ls –l | sort –k5 –n
The result of the sort

athena.ecs.csus.edu - PuTTY

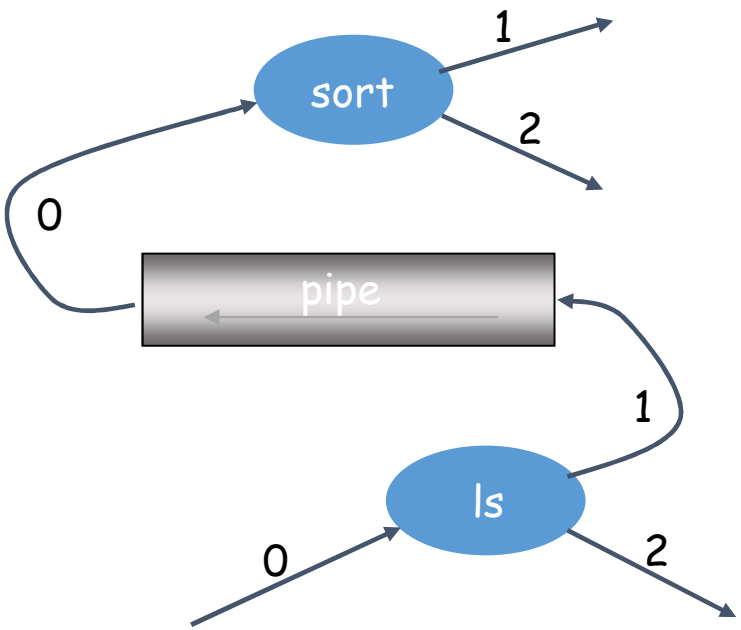
```
[bielr@athena csc60]> ls -l | sort -k5 -n
total 544
-rw-------  1 bielr faccsc      2 Dec 21 12:58 >
-rw-------  1 bielr faccsc     12 Dec 22 09:40 wcout
-rw-------  1 bielr faccsc    138 Dec 22 09:39 lsout
-rw-------  1 bielr faccsc    162 Apr 11 18:16 lsls
-rw-------  1 bielr faccsc    527 Nov 16 08:35 testScript.txt
-rw-------  1 bielr faccsc    993 Sep 16  2016 reverse1.c
drwx------ 16 bielr faccsc   4096 Apr 24 10:31 mywork/
drwx------  2 bielr faccsc   4096 Apr 19 13:07 ClassExamples/
drwx------ 48 bielr faccsc   4096 Nov 10 09:26 tlpi-dist/
drwx------  4 bielr faccsc   4096 Apr 23 10:36 student/
drwx------  6 bielr faccsc   4096 Dec 18 15:58 myworkf16/
drwx------  8 bielr faccsc   4096 Dec 16 15:07 myworkS16/
-rwx------  1 bielr faccsc   5074 Mar 16 14:08 a.out*
-rwx------  1 bielr faccsc   6438 Sep 19  2016 reverse*
-rw-------  1 bielr faccsc 235289 Apr 17  2016 tlpi-160401-dist.tar.gz
-rw-------  1 bielr faccsc 252898 Sep 21  2016 trylab1.txt
[bielr@athena csc60]>
```

Note: -k5 means sorting column 5; -n means sorting by numerical value

Using pipe

We can achieve the same effect by using a pipe.
This eliminates the intermediate file myfile.txt

```
ls -l | sort -k5 -n
```



sort file descriptor table

0	pipe read
1	standard out
2	standard err

ls file descriptor table

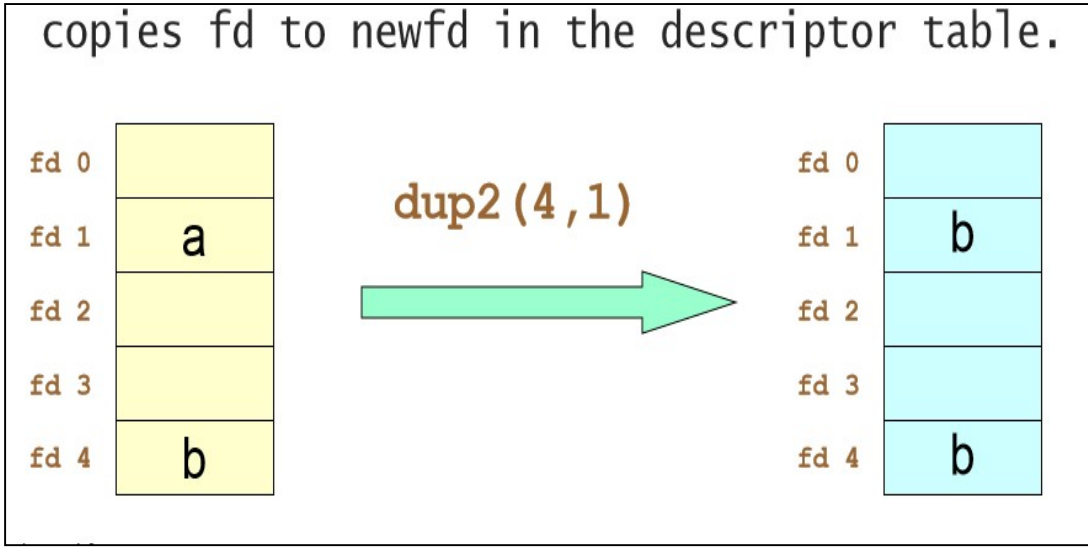
0	standard in
1	pipe write
2	standard err

Recall - dup2

#include <unistd.h>

int **dup2(int *fd1*, int *fd2*);**

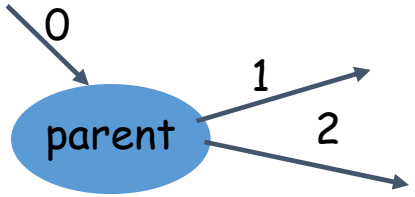
Duplicates this file descriptor on *fd2*.
If the file on *fd2* is open, it is closed first and then the duplicate is made.



In a program … Beginning

```
int main ( )
{
        pid_t childpid;
        int fd[2];

        if ((pipe(fd) == -1) ||
          ((childpid = fork( )) == -1))
        {
                perror("Failed to set up pipeline...");
                return (1);
        }
}
```



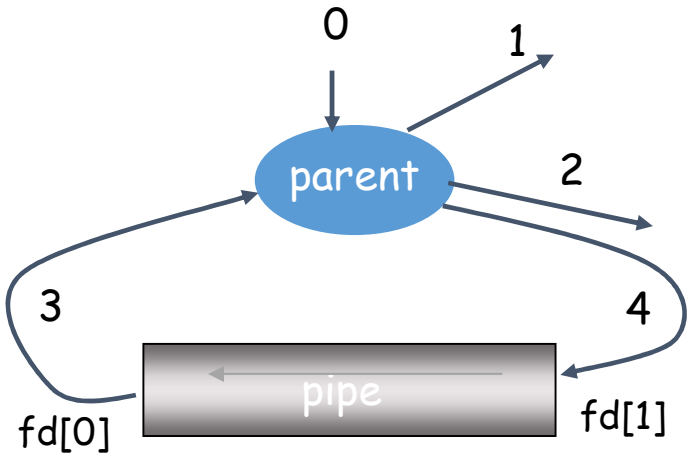
parent file descriptor table

0	standard in
1	standard out
2	standard err

In a program … with pipe execution

```
int main ( )
{
        pid_t childpid;
        int fd[2];

        if ((pipe(fd) == -1) ||
          ((childpid = fork( )) == -1))
        {
                perror("Failed to set up pipeline...");
                return (1);
        }
}
```

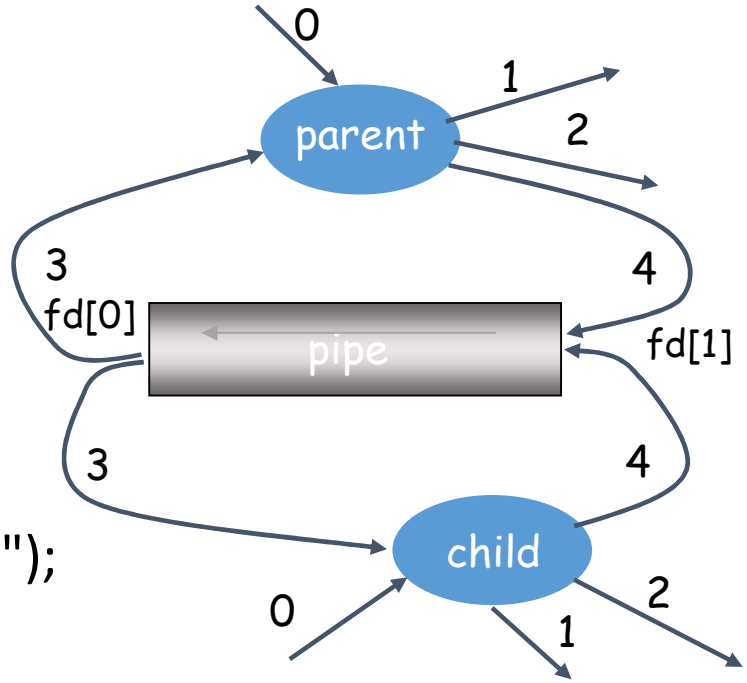


parent file descriptor table		
0	standard in	
1	standard out	
2	standard err	
3	pipe read	fd[0]
4	pipe write	fd[1]

In a program … with fork execution

```
int main ( )
{
        pid_t childpid;
        int fd[2];

        if ((pipe(fd) == -1) ||
          ((childpid = fork( )) == -1))
        {
                perror("Failed to set up pipeline...");
                return (1);
        }
```

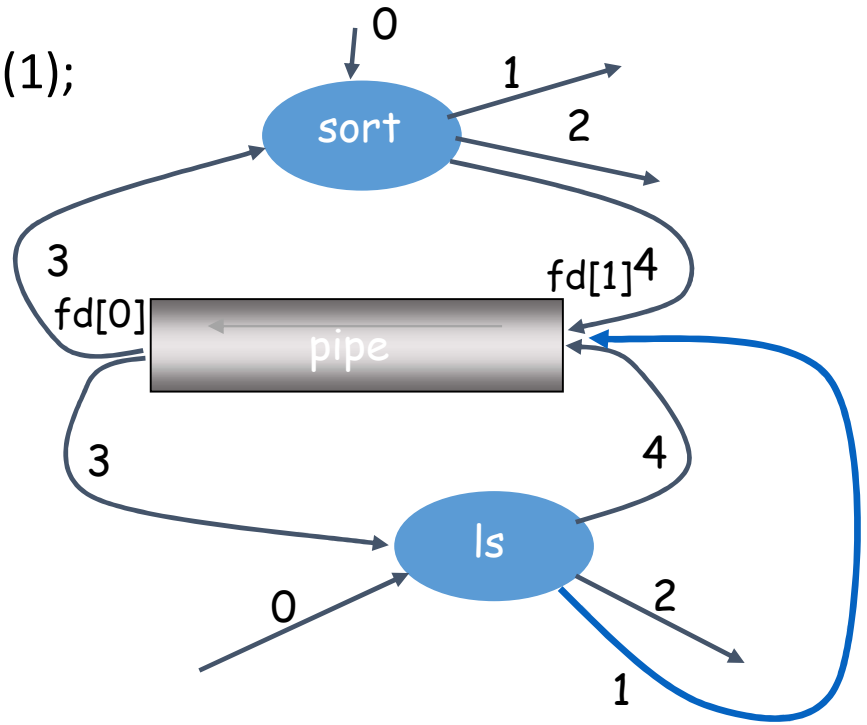


parent file descriptor table		
0	standard in	
1	standard out	
2	standard err	
3	pipe read	fd[0]
4	pipe write	fd[1]

child file descriptor table		
0	standard in	
1	standard out	
2	standard err	
3	pipe read	fd[0]
4	pipe write	fd[1]

Child calls dup2

```
if (childpid == 0)
{
      if (dup2(fd[1], STDOUT_FILENO) == -1)
           perror ("Failed to redirect stdout of ls");
      else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
          perror ("Failed to close extra file descriptors");
      else
      {
          execl("/bin/ls","ls", "-l", NULL);
          perror("Failed to exec ls ...");
      }
      return (1);
}
```

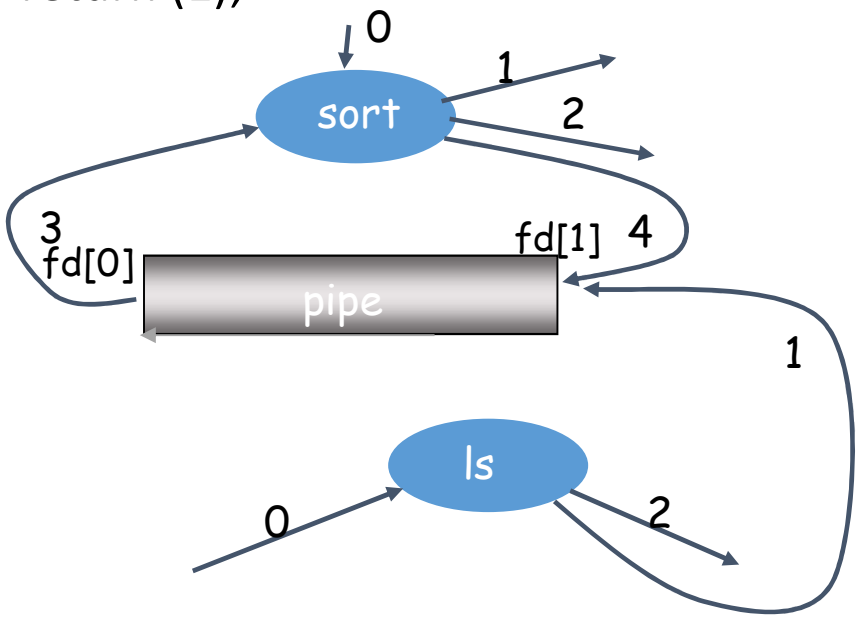


Standard out is first closed, then file descriptor fd[1] is duplicated on the file descriptor for stdout.

child file descriptor table after dup2 call		
0	standard in	
1	pipe write	
2	standard err	
3	pipe read	fd[0]
4	pipe write	fd[1]

Child closes file descriptors

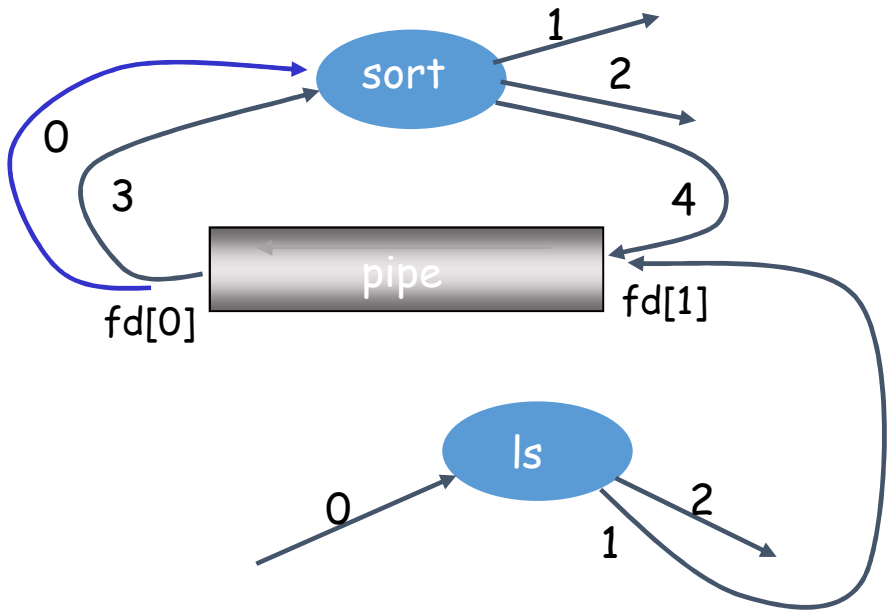
```
if (childpid == 0)
    {
        if (dup2(fd[1], STDOUT_FILENO) == -1)
            perror ("Failed to redirect stdout of ls");
        else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
            perror ("Failed to close extra file descriptors");
        else
        {
            execl("/bin/ls","ls", "-l", NULL);
            perror("Failed to exec ls ...");
        }
        return (1);
    }
```



child file descriptor table after dup2 and close(s) calls	
0	standard in
1	pipe write
2	standard err

Parent calls dup2

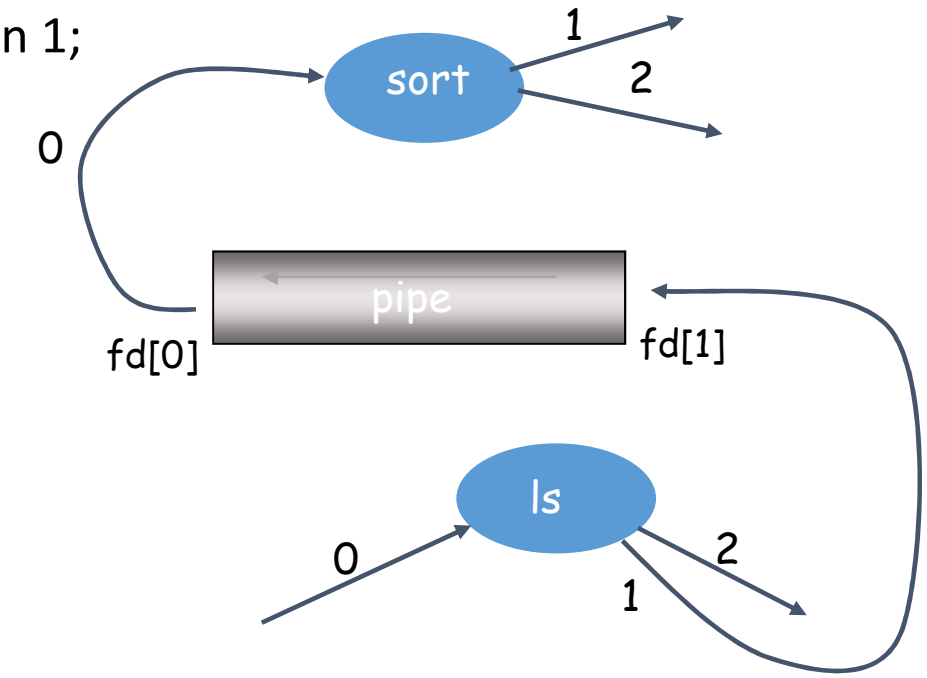
```
if(dup2(fd[0], STDIN_FILENO) == -1)     /* Parent executes sort */
        perror("Failed to redirect stdin of sort...");
else if ((close(fd[0] == -1) || close(fd[1]) == -1))
        perror("Failed to close extra file descriptors");
else
    {
        execl("/usr/bin/sort", "sort", "-k5", "-n", NULL);
        perror("Failed to exec sort");
    }
    return 1;
}
```



parent file descriptor table after dup2 call		
0	pipe read	
1	standard out	
2	standard err	
3	pipe read	fd[0]
4	pipe write	fd[1]

Parent closes file descriptors

```
if(dup2(fd[0], STDIN_FILENO) == -1)
        perror("Failed to redirect stdin of sort...");
else if ((close(fd[0] == -1) || close(fd[1]) == -1))
        perror("Failed to close extra file descriptors");
else
    {
        execl("/usr/bin/sort", "sort", "-k5", "-n", NULL);
        perror("Failed to exec sort");
    }
    return 1;
}
```

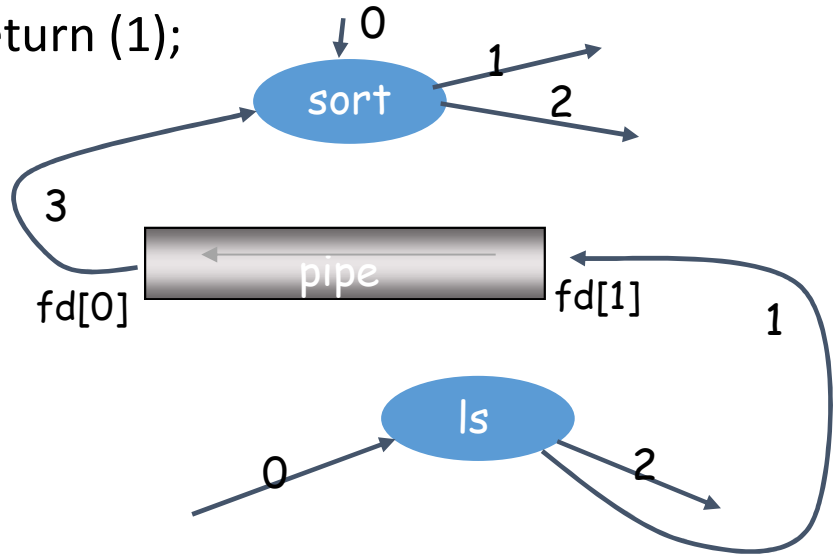


parent file descriptor table
After dup2 & close(s) calls

0	standard in
1	pipe write
2	standard err

Child executes ls

```
if (childpid == 0)
    {
        if (dup2(fd[1], STDOUT_FILENO) == -1)
            perror ("Failed to redirect stdout of ls");
        else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
            perror ("Failed to close extra file descriptors");
        else
        {
            execl("/bin/ls","ls", "-l", NULL);
            perror("Failed to exec ls ...");
        }
        return (1);
    }
```

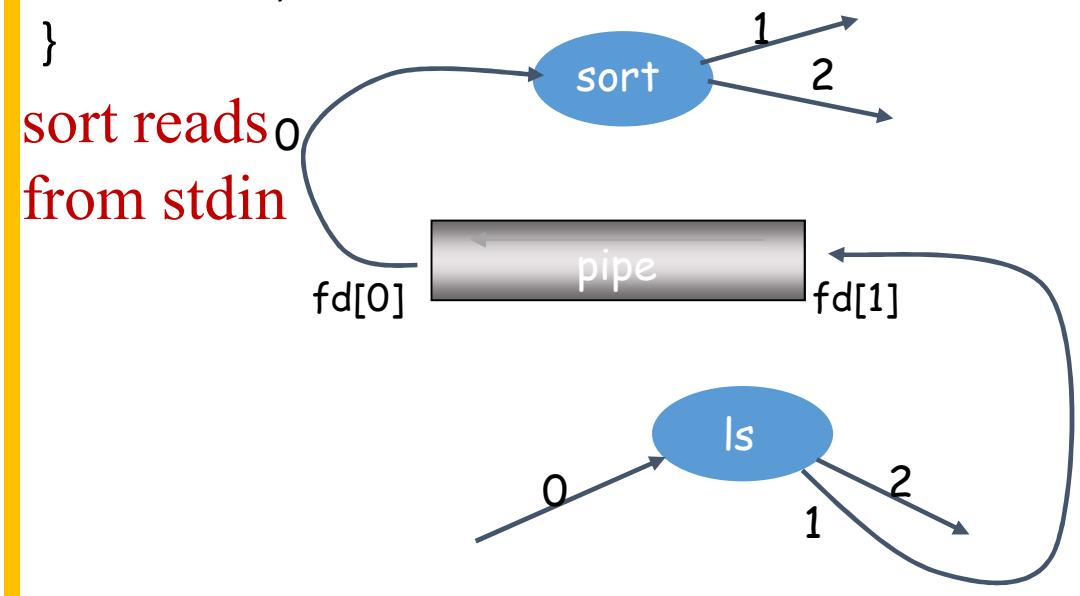


child file descriptor table After dup2 & close(s) calls	
0	standard in
1	pipe write
2	standard err

ls writes to stdout

Parent executes sort

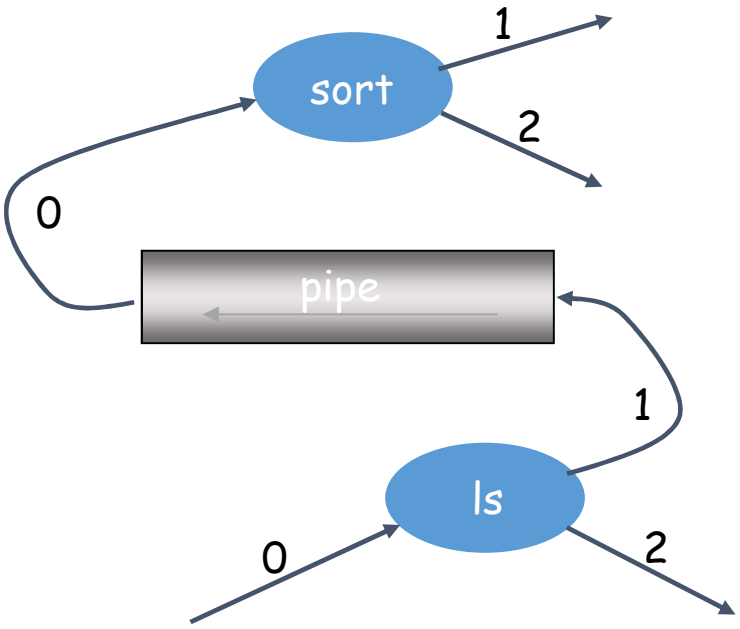
```
if(dup2(fd[0], STDIN_FILENO) == -1)
    perror("Failed to redirect stdin of sort...");
else if ((close(fd[0] == -1) || close(fd[1]) == -1))
    perror("Failed to close extra file descriptors");
else
{
    execl("/usr/bin/sort", "sort", "-k5", "-n", NULL);
    perror("Failed to exec sort");
}
return 1;
}
```



parent file descriptor table after dup2 & close(s) calls	
0	pipe read
1	standard out
2	standard err

Finally: At the end – we have

```
ls -l | sort -k5 -n
```



sort file descriptor table	
0	pipe read
1	standard out
2	standard err

ls file descriptor table	
0	standard in
1	pipe write
2	standard err

GDB debugger with *fork* *(1 of 2)*

GDB Commands using with fork	Description
(gdb) set follow-fork-mode (child or parent) Example: To follow the fork, type: follow-fork-child	Set debugger response to a program call of fork. follow-fork-mode can be: parent - the original process is debugged after a fork child - the new process is debugged after a fork The unfollowed process will continue to run. **By default**, the debugger will follow the parent process.
(gdb) set detach-on-fork (on or off)	Specifies whether GDB should debug both parent and child process after a call to **fork() -** Default is on: The child process (or parent process, depending on the value of follow-fork-mode) will be detached and allowed to run independently. This is the default.

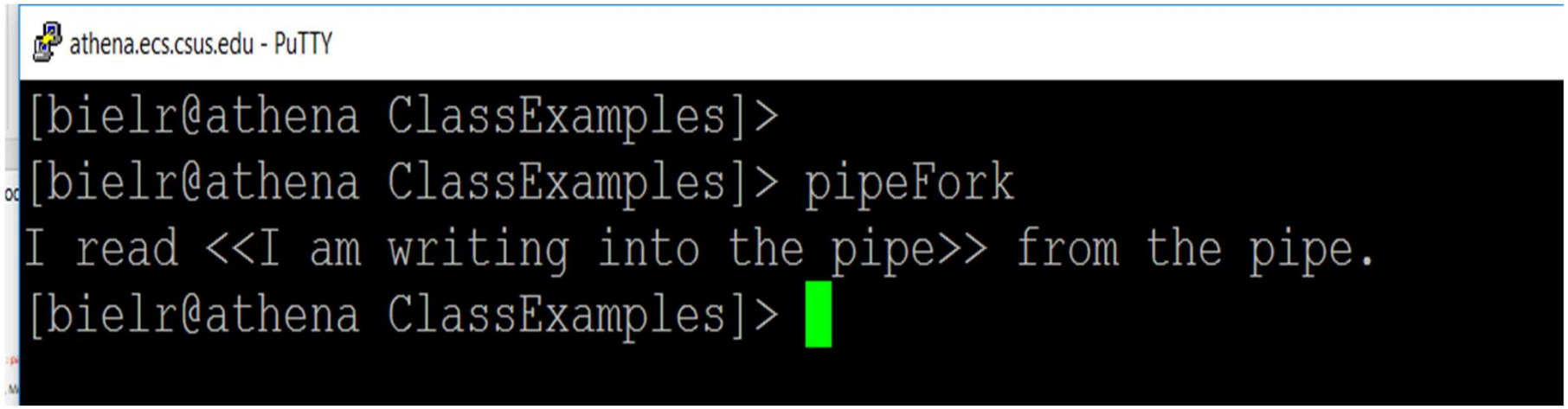
GDB debugger with *fork* *(2 of 2)*

GDB Commands using with fork	Description
(gdb) catch fork	Catch calls to fork.
(gdb) info inferiors	Display IDs of currently known inferiors.
(gdb) inferior N	Use this command to switch between inferiors. The new inferior ID must be currently known (See above command).

Demo # 1: set follow-fork-mode

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#define MAX_LENGTH 100
int main(int argc, char ** argv) {
   int fildes[2];
   char result[] = "";
   pipe(fildes);
   int pid = fork();  // Test: set follow-fork-mode : parent or child
   if (pid > 0) {
      write(fildes[1], "I am writing into the pipe", MAX_LENGTH); }
   else {
      read(fildes[0], result, MAX_LENGTH);
      printf("I read <<%s>> from the pipe.\n", result);
      _exit(0);
    }
   exit(EXIT_SUCCESS);
}
```

Demo # 1: set follow-fork-mode OUTPUT



Demo # 2: set detach-on-fork off (Process Synchronization)

Listing 44-3: Using a pipe to synchronize multiple processes

pipes/pipe_sync.c

```
#include "curr_time.h"                  /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];                         /* Process synchronization pipe */
    int j, dummy;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);               /* Make stdout unbuffered, since we
                                           terminate child with _exit() */
    printf("%s  Parent started\n", currTime("%T"));

①   if (pipe(pfd) == -1)
        errExit("pipe");

    for (j = 1; j < argc; j++) {
②       switch (fork()) {
        case -1:
            errExit("fork %d", j);

        case 0: /* Child */
            if (close(pfd[0]) == -1)    /* Read end is unused */
                errExit("close");

            /* Child does some work, and lets parent know it's done */

            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
                                        /* Simulate processing */
            printf("%s  Child %d (PID=%ld) closing pipe\n",
                    currTime("%T"), j, (long) getpid());
③           if (close(pfd[1]) == -1)
                errExit("close");

            /* Child now carries on to do other things... */

            _exit(EXIT_SUCCESS);
```

Enlarged version following

(1 of 3): set detach-on-fork off (Process Synchronization)

```
#include <sys/wait.h>
#include “tlpi_hdr.h”
#define BUF_SIZE 10

int main (int argc, char *argv[ ]) {
   int pfd[2];                                  /* Pipe file descriptors */
   char buf[BUF_SIZE];
   ssize_t numRead;

   if (argc != 2 || strcmp(argv[1], "--help") == 0)
         usageErr("%s string\n". argv[0]);
   if (pipe(pfd) == -1)                 /* Create the pipe */
         errExit("pipe");
   switch (fork()) {                            /* Call fork to create child */
         …continued….
```

(2 of 3): set detach-on-fork off (Process Synchronization)

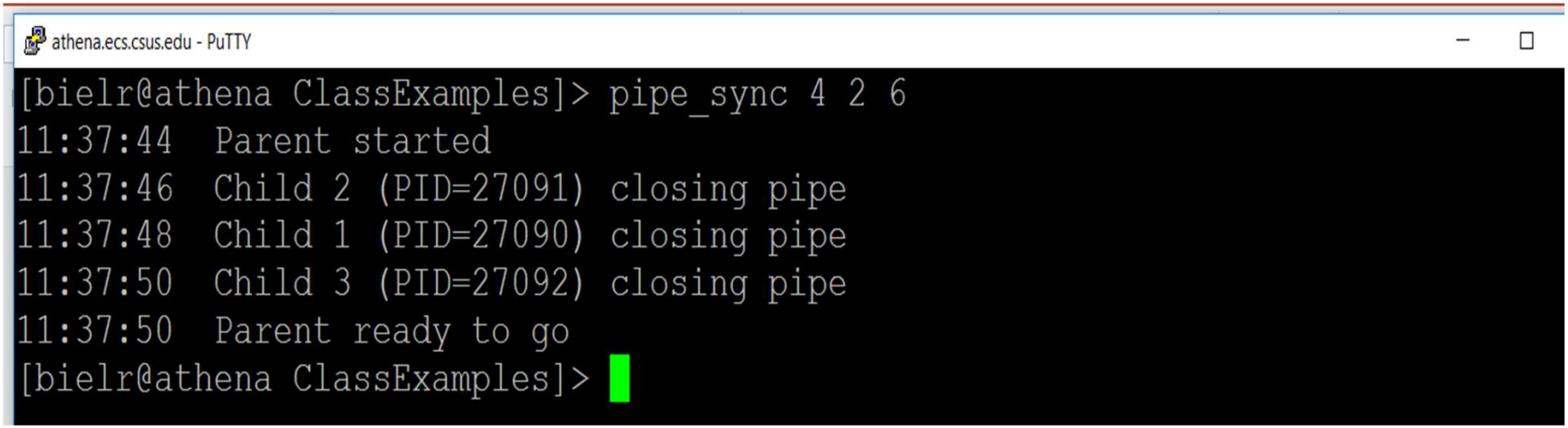
```
switch (fork()) {                       /* Call fork to create child */
        case -1:
           errExit("fork");
        case 0:                         /* Child -reads from pipe */
           if(close(pfd[1]) == -1)              /* Write end is unused */
                errExit("close - child");
           for (;;){                    /* Read data from pipe, echo on stdout */
                numRead = read(pfd[0], buf, BUF_SIZE);    /* read the data */
                if (numRead == -1)
                   errExit("read");
                if (numRead == 0)
                   break;                       /* encounters End-of-file */
                if (write(STDOUT_FILENO, BUF, numRead) != numRead)
                   fatal("child - partial/failed write");
           } /* end of for loop */
```

(3 of 3): set detach-on-fork off (Process Synchronization)

```
            write(STDOUT_FILENO, "\n", 1);     /* exit loop */
            if (close(pdf[0] == -1)
                  errExit("close");                  /*closed fd on read end  of pipe*/
            _exit(EXIT_SUCCESS);                     /* terminate */

         default:                                 /* Parent - writes to pipe */
            if (close(pdf[0]) == -1)                 /* Read end is unused */
                  errExit("close - parent");
            if (write(pdf[1], argv[1], strlen(argv[1])) != strlen(argv[1]))
                  fatal("parent - partial/failed write");
            if (close(pdf[1]) == -1)                 /* Child will see EOF */
                  errExit("close");
            wait(NULL);                              /* Wait for child to finish */
            exit(EXIT_SUCCESS);
      } /* end of switch */
   }
```

Program *pipe_sync* Output



12-UNIX

Inter-Process Communication - Pipe

The End