CSCE 3600
Principles of Systems Programming

Interprocess Communication
Part 1

University of North Texas

CSE

Interprocess Communication

Interprocess Communication

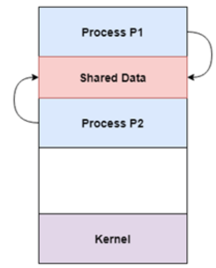
- Processes may be independent or cooperating
  - An independent process cannot affect or be affected by the execution of another process
    - A process that does not share data with another process is independent
  - A cooperating process can affect or be affected by the execution of another process
    - A process that shares data with another process is a cooperating process
    - Cooperating processes require interprocess communication (IPC)

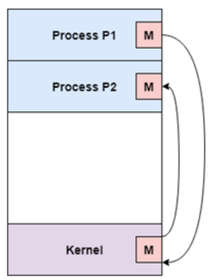
Cooperating Processes

- Why provide an environment for cooperating processes?
  - Information sharing
    - Cooperating processes can share information (such as access to the same files) among multiple processes, but a mechanism is required for parallel access
  - Modularity
    - Involves dividing complicated tasks into smaller subtasks, which can be completed by different *cooperating* processes
  - Computation speedup
    - Subtasks of a single task can be performed in parallel using cooperating processes
  - Convenience
    - Different tasks completed/managed by cooperating processes

Methods of Cooperation

- Cooperation by sharing
  - Use shared data such as memory, variables, files, databases, etc. mutually exclusive
  - Critical section used to provide data integrity with mutually exclusive writing to prevent inconsistent data





- Cooperation by communication
  - Cooperation using messages (message passing)
  - May lead to deadlock if each process waiting for a message from the other to perform an operation
  - Starvation also possible is process never receives a message

We will focus on message passing in this class

5

Message Passing

- A mechanism is needed for processes to *communicate* and *synchronize* their actions
- In a message system, processes communicate with each other without resorting to shared variables
- IPC facility provides two primitive operations for fixed or variable-sized message passing: send and receive
- If two processes wish to communicate, they need to
  - Establish a communication link between them
    - Logical (e.g., logical properties) or physical (e.g., shared memory, hardware bus)
  - Exchange messages via send/receive

6

Synchronizing Messages

- Message passing may be either blocking or non-blocking
  - Blocking is considered to be synchronous
    - Send sender blocked until message received
    - Receive receiver blocks until a message is available
  - Non-Blocking is considered to be asynchronous
    - Send sender resumes operation immediately after sending (i.e., send and continue)
    - Receive receiver returns immediately with either a valid or null message

7

Buffering

- A message queue can be implemented in one of three ways
  - Zero capacity
    - No messages may be queued within the link
      - Requires sender to wait until receiver retrieves message
  - Bounded capacity
    - Link has finite number of message buffers
      - If no buffers are available, then sender must wait if the link is full
  - Unbounded capacity
    - Link has unlimited buffer space, so sender never needs to wait

8

IPC Methods

- We will explore the following IPC methods
  - Signaling
    - As a limited form of IPC, a signal is essentially an asynchronous notification sent to a process in order to notify it of an event that occurred
  - Files
    - A file is a durable block of arbitrary information, or resource for storing information
  - Pipes
    - A pipe is a synchronized, interprocess byte stream
  - Sockets
    - Sockets provide point-to-point, two-way communication between two processes, even processes on different systems

9

CSE

Signal Handling

10

What Are Signals?

- A signal is a software interrupt
  - Notification of an event
    - A way to communicate information to a process about the state of other processes, the OS, and hardware so that the process can take appropriate action
  - Can change the flow of the program
    - When a signal is delivered to a process, process will stop what it's doing – and either handle or ignore signal
  - Signals can be delivered in an unpredictable manner
    - Originate outside of currently executing process
    - Asynchronous events due to external trigger at the hardware or OS level – causes a context switch!

11

What Are Signals?

- Every signal has a name that starts with `SIG`, a value, a default action, and a description
  - See man 7 signal

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from **abort**(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from **alarm**(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Defined in sys/signal.h

12

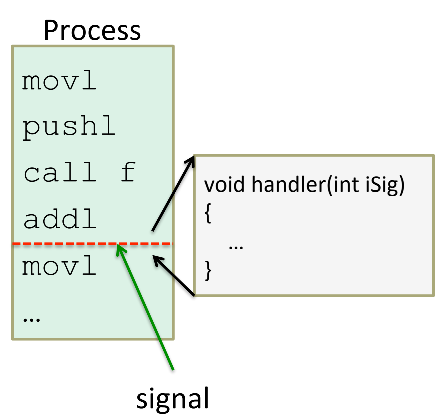
Default Action of Signals

- Each signal has a default action
  - `Term` the process will terminate
  - `Core` the process will terminate and produce a core dump file that traces the process state at the time of termination
  - `Ign` the process will ignore the signal
  - `Stop` the process will stop, like `Ctrl-Z`
  - `Cont` the process will continue from being stopped

13

Flow of Signals

1. Event gains attention of OS
2. OS stops process execution immediately
   - Sends it a signal
3. Signal Handler then executes to completion
4. Process execution resumes where it left off



14

Signal Events

- In the context of terminal signaling, programs can stop, start, and terminate
  - `Ctrl-C` is the same as sending `SIGINT` (2) signal
    - Default handler exits process
  - `Ctrl-Z` is the same as sending a `SIGSTOP` (20) signal
    - Default handler suspends process
  - `Ctrl-\` is the same as sending a `SIGQUIT` (3) signal
    - Default handler exits process
  - Typing `fg` or `bg` at the terminal is the same as sending a `SIGCONT` (18) signal to bring or send a process to the foreground or background, respectively

Signal Events

- Signals are notification of events
  - We can inject signals, such as `Ctrl-C`, to gain the attention of the OS and stop the process by sending a `SIGINT` signal
  - But some are done internally, such as when a process makes an illegal memory reference
    - Event gains attention of the OS
    - OS stops process execution immediately, sending it a `SIGSEGV` (11) signal
    - Signal handler for `SIGSEGV` signal executes to completion
      - Default signal handler for `SIGSEGV` signal prints “segmentation fault” and exits process

Signal Handling

- Each signal type has a default handler
  - Most default handlers exit the process
- A program can install its own handler for signals of almost any type
  - Cannot install its own handler for the following signals
    - `SIGKILL` (9)
      - Default handler exits the process
      - Catchable termination signal is `SIGTERM` (15)
    - `SIGSTOP` (19)
      - Default handler suspends the process
      - Can resume the process with signal `SIGCONT` (18)
      - Catchable suspension signal is `SIGTSTP` (20)

17

Installing a Signal Handler

- sighandler_t signal(int iSig, sighandler_t pfHandler);
  - Installs pfHandler as the handler of signals for type iSig
  - pfHandler is a function pointer

    typedef void (* sighandler_t) (int);
  - Returns the old handler on success; `SIGERR` on error
  - After call, pfHandler is invoked whenever process receives a signal of type iSig

18

Signal Handler Example

```
#include <stdio.h>
#include <assert.h>
#include <signal.h>

void myHandler(int iSig) {
        printf("In myHandler with argument %d\n", iSig);
}

int main() {
        void (*pfRet)(int) = signal(SIGINT, myHandler);
        assert(pfRet != SIG_ERR);

        printf("Entering an infinite loop\n");
        while (1) {
                printf(".");
        }
        return 0;
}
```

19

Predefined Signal Handlers

- Can install predefined signal handler SIG_IGN to ignore signals

  void (*pfRet)(int) = signal(SIGINT, SIG_IGN);

  – Subsequently, process will ignore `SIGINT` (2) signals
- Can install predefined signal handler SIG_DFL to restore default signal handler

  void (*pfRet)(int) = signal(SIGINT, myHandler);

  …

  pfRet = signal(SIGINT, SIG_DFL);

  – Subsequently, process will handle `SIGINT` (2) signals using the default handler for `SIGINT` (2) signals

20

Sending Signals via Command

- kill –signal pid
  - Send signal of type signal to process with ID pid
  - Specify signal type name (`-SIGINT`) or number (`-2`)
  - Examples:

    kill –2 1234
    kill –SIGINT 1234

    Same as typing `Ctrl-C` if process 1234 is running in foreground
- killall loop
  - Send a signal to all processes named loop to “terminate”
  - This will actually send the signal `SIGTERM`, whose purpose is to communicate a termination request to the given process, which does not necessarily have to terminate

21

Sending Signals via System Call

- int raise(int iSig);
  - Instructs OS to send signal of type iSig to current process
  - Returns 0 to indicate success; non-0 to indicate failure
  - Example:

```
int retVal = raise(SIGINT);  // process commits suicide
assert(retVal != 0);         // should not get here
```

- int kill(pid_t iPid, int iSig);
  - Sends iSig signal to process with ID iPid
  - Equivalent to raise(iSig) when iPid is ID of current process
  - Example:

```
pid_t iPid = getpid();   // process gets its pid
kill(iPid, SIGINT);      // sends itself a SIGINT signal, commits suicide
```

22

pause() Function

- int pause( );
  - Suspends the calling process, without wasting resources, until some kind of signal is caught
  - Signal action can be the execution of a signal handler function or process termination
  - Only returns –1 when a signal was caught and the signal-catching function returned

23

pause() Example

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void sig_usr( int signo );    /* handles two signals */

int main() {
   int i = 0;
   if( signal( SIGUSR1,sig_usr ) == SIG_ERR )
        printf( "Cannot catch SIGUSR1\n" );
   if( signal( SIGUSR2,sig_usr ) == SIG_ERR )
        printf( "Cannot catch SIGUSR2\n" );
            :
   while(1) {
        printf( "%2d\n", i );
        pause();
        /* pause until signal handler
         * has processed signal */
        i++;
   }

   return 0;
}
```

24

pause() Example

```
/* argument is signal number */
void sig_usr( int signo )
{
   if( signo == SIGUSR1 )
        printf( "Received SIGUSR1\n" );
   else if( signo == SIGUSR2 )
        printf( "Received SIGUSR2\n" );
   else
        printf( "Error: received signal %d\n", signo);

     return;
}
```

```
$ sig_examp &
[1]       4720
 0
$ kill -USR1 4720
Received SIGUSR1
 1
$ kill -USR2 4720
Received SIGUSR2
 2
$ kill 4720         /* send SIGTERM */
[1] + Terminated    sig_examp &
$
```

25

CSE

Files

26

Overview of Files

- Data communication with a C program and the outside world is performed through files
- Files are a non-volatile way to store data by means of such media as tape, CD-ROM, floppy disk, ZIP disk, hard drive, etc.
- C (just like the Linux operating system) considers all process communication media to be files
  - An ordinary file on a disk is considered to be a file
  - So is the keyboard, the screen, parallel ports, and serial ports

27

Linux Files

- A Linux file is a sequence of *m* bytes
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$
- All I/O devices are represented as files

  `/dev/sda2` (`/usr` disk partition)

  `/dev/tty2` (terminal)
- Even the kernel is represented as a file

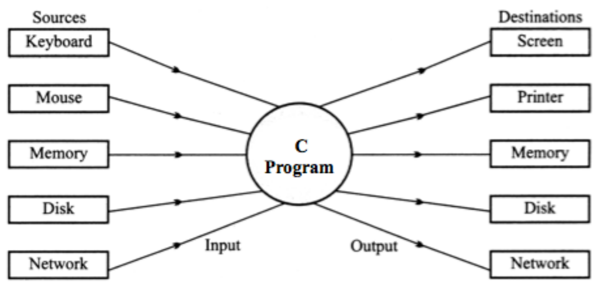
  `/dev/kmem` (kernel memory image)

  `/proc` (kernel data structures)

28

I/O and Data Movement

- Input and output share common property of unidirectional movement of data and support to sequential access to data
  - The flow of data into a program (input) may come from different devices such as keyboard, mouse, memory, disk, network, or another program
  - The flow of data out of a program (output) may go to the screen, printer, memory, disk, network, another program



Some Linux File Types (Review)

- Regular file
  - Binary or text file (Linux does not know the difference)
- Directory file
  - A file that contains the names and locations of other files
- Character special and block special files
  - Terminals (character special) and disks (block special)
- FIFO (named pipe)
  - A file type used for interprocess communication to exchange data between unrelated processes
- Socket
  - A file type used for network communication between processes

Communication through Files

- Storing and manipulating data using files is known as file processing
- Programs access files through basic file operations
  - Open a file
  - Read data from a file
  - Write data to a file
  - Close a file

  these "process" a file
- Text files store all data types as character bytes
  - The way a program reads the data (either text or binary mode) determines how the data is interpreted
  - Example: `12 z rti 456.79 room`

31

Standard I/O Functions

- Examples of standard I/O functions:
  - Opening and closing files (`fopen` and `fclose`)
  - Reading and writing bytes (`fread` and `fwrite`)
  - Reading and writing text lines (`fgets` and `fputs`)
  - Formatted reading and writing (`fscanf` and `fprintf`)

```
FILE* fp;
fp = fopen("In.file", "rw");
fscanf(fp, ……);
fprintf(fp, ……);
fread(………, fp);
fwrite(………, fp);
```

32

Standard I/O Streams

- Standard I/O models open files as *streams*
  - Abstraction for a file descriptor and a buffer in memory.
- Three files are automatically opened each time a C program is run and automatically closed when a C program ends:
  - `stdin` (standard input, default source = keyboard)
  - `stdout` (standard output, default destination = screen)
  - `stderr` (standard error, default destination = screen)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

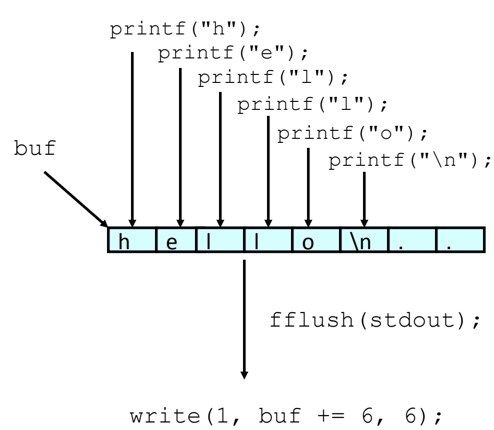
streams

file descriptors

33

Buffering in Standard I/O

- Standard I/O functions use buffered I/O



34

Standard I/O Buffering Example

- You can see this buffering using the Linux `strace` program:

```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    return 0;
}
```

```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6...)                = 6
...
_exit(0)                                 = ?
```

35

File Descriptors

- Each open file is associated with an open file description
  - An OS internal record of how a process or group of processes are *currently* accessing a file that includes
    - File offset indicates byte position where next I/O operation begins
    - File status indicates append mode/not, blocking/non-blocking, etc.
    - File access mode indicates whether file can be read or written
- Each process has a logical array of references to open file descriptions
  - Logical indices into this array are file descriptors used to identify the files for I/O operations
    - A file descriptor does not describe a file – it's just a number ephermerally associated with a particular open file description
  - List open files: lsof –p <pid>

36

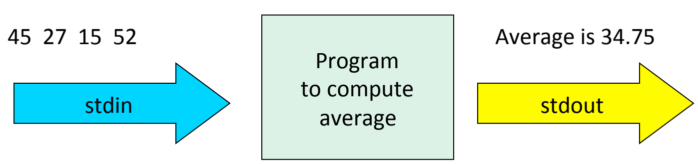
File Descriptors

- Other descriptors are assigned by system calls to open/create files, create pipes, or bind to devices or network sockets
  – E.g., pipe(), socket(), open(), creat()
- A common set of system calls operate on open file descriptors independent of their underlying types
  – E.g., read(), write(), dup(), dup2(), close()

37

Files as Pipes for Data

- Files act as *pipes* to bring a stream of data into the program and send a stream of data out of the program



- The program reads data from `stdin` and writes data to `stdout` as if they were *ordinary* files

38

One Byte at a Time

- Data is read or written one byte at a time from a file until the end of the file is reached or until the file is closed
- The file system uses a pointer to keep track of the next byte to read or to write

```
Smith   Jack    1045.76  Manager        15

Hanson  Susan     98.62  Operator        7

Jones   Nancy    790.25  Administrator  10

Doe     Carl     526.71  Technician     12
```

- In this file, the program has read the data as far as the letter 'J'. The next character to be read is 'a'

Functions Using stdin and stdout

- Some standard C functions implicitly use `stdin` and `stdout`

```
scanf("%d", &aNumber);
aSymbol = getchar();
gets(theBuffer); // High security risk
printf("Average: %5.2f", theAverage);
putchar(aCharacter);
puts(aPhrase);
```

Functions Using stdin and stdout

- Other functions need to have `stdin` and `stdout` specified as the file descriptor

```
fscanf(stdin, "%d", &aNumber);
aSymbol = fgetc(stdin);
fgets(theBuffer, sizeof(theBuffer), stdin);
fprintf(stdout, "Average: %f", theAverage);
fputc(aCharacter, stdout);
puts(aPhrase, stdout);
```

41

CSE

Pipes

42

What is a Pipe?

- A pipe is a simple, synchronized way of passing information between processes
  - A pipe is a special file/buffer that stores a limited amount of data in a FIFO (i.e., sequential) manner
  - Pipes are commonly used from within shells to connect `stdout` of one utility to `stdin` of another
- Pipes provide a basic level of synchronization
  - If a process tries to write to a full pipe, it is blocked until the reader process consumes some data
  - If a process tries to read from an empty pipe, it is blocked until the writer produces some data
- Data is written to and read from the pipe using the unbuffered system calls write() and read()

43

Two Types of Pipes

- Unnamed pipes
  - Not associated with any file
  - Can only be used with related processes, such as a parent and child process
  - Exist only as long as using processes are not terminated and support unidirectional communication
  - Created using the pipe() system call
- Named pipes, or FIFOs
  - Associated with a file and can be used with unrelated processes
  - Has a directory entry with file access permissions
  - Can support bidirectional communication
  - Created using the mknod() or mkfifo() system calls

44

Unnamed Pipes

- The pipe() system call creates an unnamed pipe and opens it for reading and writing
- General syntax

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

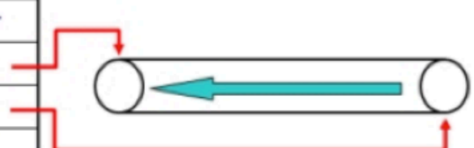
  – If successful, it will return two integer file descriptors in fd[0] and fd[1]
  – fd must be an integer array of size 2
  – The file descriptor in fd[0] is associated with the read end of the pipe and fd[1] is associated with the write end of the pipe
  – Returns -1 if an error occurred

45

Pipes

File Descriptor Table

0	stdin
1	stdout
2	stderr
3	fd[0]
4	fd[1]
5	...



46

Writing to a Pipe

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

- Appends up to count bytes from the end of the pipe referenced by the file descriptor from the buffer
- Atomicity guaranteed for requests with size typically around 4096 bytes or less
  - See PIPE_BUF in /usr/include/linux/limits.h for block size
- If write() done for pipe not open for reading by any process
  - SIGPIPE signal generated to signify a broken pipe with errno set to EPIPE (broken pipe)
- See man 2 write for more information on this system call

Reading from a Pipe

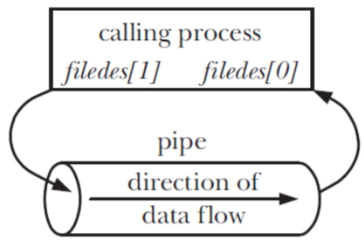
```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

- Attempts to read up to count bytes from end of the pipe referenced by the file descriptor from the buffer in a FIFO manner
  - Returns number of bytes read from the buffer
- If read() done for pipe not open for writing by any process, 0 is returned
- If read() done for empty pipe open for writing by another process, the process sleeps until the input becomes available
- See man 2 read for more information on this system call

Creating and Using Pipes

- Created using pipe():

```
int filedes[2];
pipe(filedes);
.
.
.
write(filedes[1], buf, count);
read(filedes[0], buf, count);
```



- Pipes are anonymous
  - There is no name in the file system
- But then how do two processes share a pipe?

49

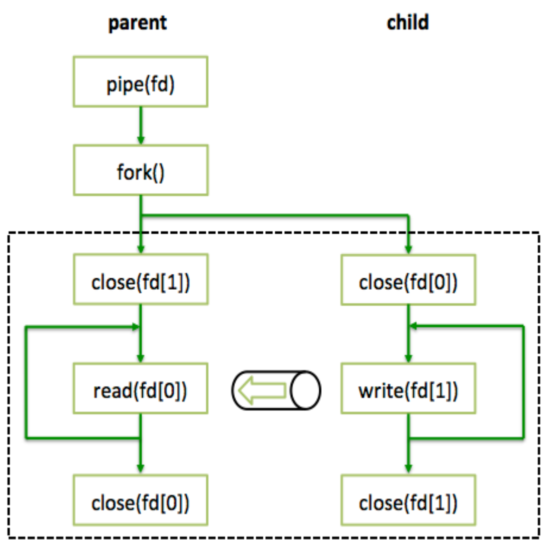
Unnamed Pipes

- Since access to an unnamed pipe is through the file descriptor mechanism, only the process that created the pipe and its descendants may use the pipe
- The typical sequence of opening unnamed pipes
  - The parent process creates an unnamed pipe
    - It is crucial that this be done before forking
  - The parent process forks
  - The writer process closes the read end of the pipe
  - The reader process closes the write end of the pipe

    May be reversed
  - The processes communicate by using write() and read()
  - Each process closes its active pipe-end when finished

50

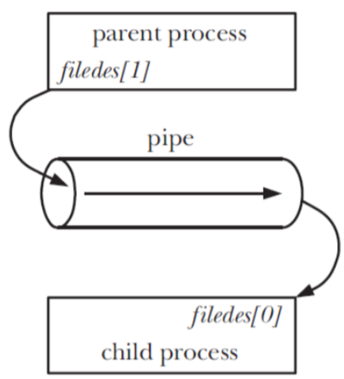
Pipe Communication Scheme



51

Sharing a Pipe

```
int filedes[2];
pipe(filedes);
pid = fork();
if (pid == 0) {
   close(filedes[1]);
   // child now reads
} else {
   close(filedes[0]);
   // parent now writes
}
```



- The fork() system call duplicates parent's file descriptors
  - Parent and child must close unused descriptors – necessary for correct use of pipes

52

Pipe Example (1)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define READ 0  /* The index of the read end of the pipe */
#define WRITE 1 /* The index of the write end of the pipe */

char* phrase = "This goes in the pipe";

int main () {
        int fd[2], bytesRead;
        char message[100]; /* Parent process' message buffer */

        pipe(fd); /* Create unnamed pipe */
        if (fork() == 0) /* Child, writer */
        {
           close(fd[READ]); /* Close unused end */
           write(fd[WRITE], phrase, strlen(phrase) + 1); /* Include NULL */
           close(fd[WRITE]); /* Close used end */
        }
        else /* Parent, reader */
        {
           close(fd[WRITE]); /* Close unused end */
           bytesRead = read(fd[READ], message, 100);
           printf("Parent just read %i bytes: %s\n", bytesRead, message);
           close(fd[READ]); /* Close used end */
        }
}
```

53

Pipe Example (2)

```
/* This program will demonstrate what happens if a read takes
   place with a pipe whose write end is closed, and vice versa */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#define READ 0 /* The index of the read end of the pipe */
#define WRITE 1 /* The index of the write end of the pipe */
char* phrase = "Another pipe example end closed";
void signal_catcher(int);
int main() {
     int fd[2], bytesWritten = 0, bytesRead;
     char message[100]; /* Parent process' message buffer */
     signal(SIGPIPE, signal_catcher);
     pipe(fd); /* Create pipe */
     close(fd[WRITE]); /* Close used end */
     printf("About to read from pipe\n");
     bytesRead = read(fd[READ], message, 100);
     printf("%i bytes were read with write closed\n", bytesRead);
     close(fd[READ]); /* Close used end */
     pipe(fd); /* Recreate unnamed pipe */
     close(fd[READ]); /* Close unused end */
     printf("About to write to pipe\n");
     bytesWritten = write(fd[WRITE], phrase, strlen(phrase) + 1);
     printf("%i bytes were written with read end closed\n", bytesWritten);
     close(fd[WRITE]);
}
void signal_catcher(int theSig) {
     printf("A SIGPIPE (%i) has been caught\n", theSig);
}
```

54

dup() System Call

```
#include <unistd.h>
int dup(int oldfd);
```

- Creates a copy of the oldfd file descriptor with the same open pipe, file pointer, and access mode in common with the original file descriptor that is set to remain open across the exec family of system calls
- Returns the lowest unused file descriptor, -1 if error
  - Problems if file descriptor returned is not one you are expecting!
- A useful system call to convert a stream to a file descriptor

```
int fileno(FILE *fp);
```

Note that dup() refers to a file descriptor, not a stream!

55

dup() Example

```
int fd[2];
pipe(fd);
close(fileno(stdout));
dup(fd[1]);
```

Note that no error checking is done in this example

- Since 1 is the lowest fd available, the write end of the pipe is duplicated at fd 1 (`stdout`)
  - Now any data written to `stdout` will be written to the pipe
- But you are taking a chance that the file descriptor that will be returned by dup() is what you want
  - The process may be interrupted between the close() and the dup()

56

dup2() System Call

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

- A better alternative to dup() in that it makes the newfd file descriptor as the copy of the old file descriptor atomically, closing newfd first if necessary
  - There is no time lapse between closing newfd and duplicating oldfd into its spot
- Both oldfd and newfd now refer to the same file

57

dup() and dup2() Example

```
/* This program demonstrates the dup and dup2 system calls.
   You must have a file present in the directory called "test.txt".
   It may be empty or have stuff in it doesn't matter. */
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/file.h>
int main() {
    int fd1, fd2, fd3;
    fd1 = open("test.txt", O_RDWR | O_TRUNC);
    printf("fd1 = %i\n", fd1);
    write(fd1, "what's", 6);
    fd2 = dup(fd1); /* make a copy of fd1 */
    printf("fd2 = %i\n", fd2);
    write(fd2, " up", 3);
    close(0); /* close standard input */
    fd3 = dup(fd1); /* make another copy of fd1 */
    printf("fd3 = %i\n", fd3);
    write(0, " doc", 4); /* because 0 was the smallest file descriptor */
                         /* and now belongs to fd3                    */
    dup2(3,2); /* duplicate channel 3 to channel 2 */
    write(2, "?\n", 2);
}
```

58

dup() and dup2() Example

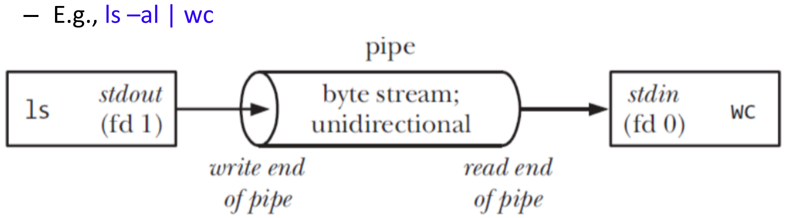
The file descriptor table

fd	in the beginning	after first open	after 1st dup()	after close	after 2nd dup()	after dup2()
0	stdin	stdin	stdin		test.txt (fd3)	test.txt (fd3)
1	stdout	stdout	stdout	stdout	stdout	stdout
2	stderr	stderr	stderr	stderr	stderr	test.txt (fd1)
3		test.txt (fd1)	test.txt (fd1)	test.txt (fd1)	test.txt (fd1)	test.txt (fd1)
4			test.txt (fd2)	test.txt (fd2)	test.txt (fd2)	test.txt (fd2)
...						

59

Implement Command-Line Pipe

- Can we implement a command-line pipe with pipe()?
  - E.g., ls –al | wc



- How do we attach the `stdout` of ls to the `stdin` of wc?

stdin ⟶ ls –al ⟶ wc ⟶ stdout

60

Implement Command-Line Pipe

```
/* Modeling the command-line command:  ls -al | wc  using pipes */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
enum { READ, WRITE };
int main() {
     int fd[2];
     if (pipe(fd) == -1) { /* generate the pipe */
          perror("Pipe");
          exit(1);       }
     switch (fork()) {
          case -1: perror("Fork");
                  exit(2);
          case 0: /* in child */
               dup2(fd[WRITE], fileno(stdout));
               close(fd[READ]);
               close(fd[WRITE]);
               execl("/bin/ls", "ls", "-al", (char *)0 );
               exit(3);
          default: /* in parent */
               dup2(fd[READ], fileno(stdin));
               close(fd[READ]);
               close(fd[WRITE]);
               execl("/usr/bin/wc", "wc", (char *)0 );
               exit(4);
     }
     return 0;
}
```


Implement Redirection

- When a process forks, the child inherits a copy of its parent's file descriptors
- When a process execs, all non-close-on-exec file descriptors remain unaffected
  - This includes `stdin`, `stdout`, and `stderr`
- To implement redirection, the shell simply does the following
  - The parent shell forks then waits for the child shell to terminate

Implement Redirection

```
/* The program demonstrates implementing redirection.
   To run: ./a.out <output filename> <command>
 */
#include <stdio.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char* argv[])
{
    int fd;
    /* Open file for redirection */
    fd = open(argv[1], O_CREAT | O_TRUNC | O_WRONLY, 0600);
    dup2(fd, 1); /* Duplicate descriptor to standard output */
    close(fd); /* Close original descriptor to save descriptor space */
    execvp(argv[2], &argv[2]); /* Invoke program; will inherit stdout */
    perror("main"); /* Should never execute */
}
```


Implement Redirection

- The child shell opens the file, say `ls.out`, creating it or truncating it as necessary
- The child shell then
  - Duplicates the file descriptor of `ls.out` to the standard output file descriptor (fd 1)
  - Closes the original file descriptor of `ls.out`
    - All standard output is therefore directed to `ls.out`
  - The child shell then execs the `ls` utility
  - Since the file descriptors are inherited during an exec, all `stdout` of `ls` goes to `ls.out`
- When the child process terminates, the parent resumes
- The parent's file descriptors are unaffected by the child's action as each process maintains its own descriptor table

Implement Pipe and Redirection

```
/* Equivalent to "sort < file2 | uniq */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
    int fd[2];
    FILE *fp = fopen("file2", "r");
    dup2(fileno(fp), fileno(stdin));
    fclose(fp);
    pipe(fd);
    if (fork() == 0) {
        dup2(fd[1], fileno(stdout));
        close(fd[0]);
        close(fd[1]);
        execl("/usr/bin/sort", "sort", (char*) 0);
        exit(2);
    } else {
        dup2(fd[0], fileno(stdin));
        close(fd[0]);
        close(fd[1]);
        execl("/usr/bin/uniq", "uniq", (char*)0);
        exit(3);
    }
    return 0;
}
```