Objectives

- Ideas and Skills
  - I/O redirection: What and Why?
  - Definitions of standard input, output, and error
  - Redirecting standard I/O to files
  - Using fork to redirect I/O for other programs
  - Pipes
  - Using for with pipes
- System calls and functions
  - dup, dup2
  - pipe

Pipes and I/O redirection used in Unix Shell

```
[wch@cwu-xp ~]$ ls
bin              Document   project                          sfdnls.m  workspace
DEPENDTUIT.doc   grade.txt  Projects                         software
Desktop          optim      runtime-EclipseApplication       test
[wch@cwu-xp ~]$ ls > ls.txt
[wch@cwu-xp ~]$ more ls.txt
bin
DEPENDTUIT.doc
Desktop
Document
grade.txt
ls.txt
optim
project
Projects
runtime-EclipseApplication
sfdnls.m
software
test
workspace
[wch@cwu-xp ~]$ ls -l |more
```

An example from the book

A script listing the login and logout records

```
get list of users (call it prev)
while true
 sleep
 get list of users (call it curr)
 compare lists
  in prev, not in curr ->logout
  in curr, not in prev -> login
 make prev = curr
repeat
```

```
who | sort > prev
while true; do
 sleep 60
 who | sort > curr
 echo "logged out:"
 comm -23 prev curr
 echo "logged in"
 comm -13 prev curr
 mv curr prev
done
```

Facts about standard I/O

- Each program has three standard input and output stream, which it does not to open explicitly.
  - standard input, FILE *stdin, has file descriptor 0
  - standard output, FILE *stdout, has file descriptor 1
  - standard error, FILE *stderr, has file descriptor 2
- By default, the three standard streams connect to the terminal.

Question: Can programs not launched within a terminal, read and write from the standard stream? You can try the following code:

```
#include <stdio.h>

int main(int argc,  char *argv[ ]){

   int i = 3, j = 4;            /*local variables to function main*/

   if(printf("3 + 4 is %d\n", 3+4)!=0)
   {
      FILE *fp = fopen("temp.txt", "w");
```

```
        if(fp){
            fprintf(fp, "3 + 4 is %d\n", 3+4);
            fclose(fp);
        }
    }
    return 0;
}
```

Facts about standard I/O

It is the shells not the program that redirects the I/O. Let's compile and run the following C program.

```
/* listargs.c
 *              print the number of command line args, list the args,
 *              then print a message to stderr
 */
#include        <stdio.h>

main( int ac, char *av[] )
{
        int     i;

        printf("Number of args: %d, Args are:\n", ac);
        for(i=0;i<ac;i++)
                printf("args[%d] %s\n", i, av[i]);

        fprintf(stderr,"This message is sent to stderr.\n");
}
```

How does the Shell redirect I/O?

When we open a file, we get a file descriptor, which is simply an index of an array containing all the open files. The kernel assigns the least available file descriptor to the process when getting an *open()* request.

Since stdin always points to the file attached with file descriptor 0, we can redirect stdin to a file by using close-then-open method.

```
/* stdinredir1.c
 *      purpose: show how to redirect standard input by replacing file
 *               descriptor 0 with a connection to a file.
 *       action: reads three lines from standard input, then
 *               closes fd 0, opens a disk file, then reads in
 *               three more lines from standard input
 */
#include        <stdio.h>
#include        <fcntl.h>

main()
{
        int     fd ;
        char    line[100];
```

```
        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );

        /* redirect input */

        close(0);
        fd = open("/etc/passwd", O_RDONLY);
        if ( fd != 0 ){
                fprintf(stderr,"Could not open data as fd 0\n");
                exit(1);
        }

        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
}
```

How does the Shell redirect I/O?

Method two: open..close..dup..close

- Step 1 *open(file)* Open the file to which stdin should be attached. This returns a file descriptor fd.
- Step 2 *close(0)* Close file descriptor 0. File descriptor 0 is now unused.
- Step 3 *dup(fd)* The dup(fd) system call makes a duplicate of fd. The duplicate uses the lowest number unused file descriptor. Therefore file descriptor 0 will be used.
- Step 4 *close(fd)*

How does the Shell redirect I/O?

```
/* stdinredir2.c
 *      shows two more methods for redirecting standard input
 *      use #define to set one or the other
 */
#include        <stdio.h>
#include        <fcntl.h>

/* #define      CLOSE_DUP               /* open, close, dup, close */
/* #define      USE_DUP2        /* open, dup2, close */

main()
{
        int     fd ;
        int     newfd;
        char    line[100];

        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
```

```
        /* redirect input */
        fd = open("data", O_RDONLY);    /* open the disk file   */
#ifdef CLOSE_DUP
        close(0);
        newfd = dup(fd);                /* copy open fd to 0    */
#else
        newfd = dup2(fd,0);             /* close 0, dup fd to 0 */
#endif
        if ( newfd != 0 ){
                fprintf(stderr,"Could not duplicate fd to 0\n");
                exit(1);
        }
        close(fd);                      /* close original fd    */

        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
}
```

Manpage of dup and dup2

```
NAME
       dup, dup2 - duplicate a file descriptor

SYNOPSIS
       #include <unistd.h>

       int dup(int oldfd);
       int dup2(int oldfd, int newfd);

DESCRIPTION
       dup and dup2 create a copy of the file descriptor oldfd.

       After successful return of dup or dup2, the old and new descriptors may
       be used interchangeably. They share locks, file position  pointers  and
       flags;  for example, if the file position is modified by using lseek on
       one of the descriptors, the position is also changed for the other.

       The two descriptors do not share the close-on-exec flag, however.

       dup uses the lowest-numbered unused descriptor for the new  descriptor.

       dup2  makes  newfd  be the copy of oldfd, closing newfd first if neces-
       sary.
```

how the Shell redirect I/O of other process

```
/* whotofile.c
 *      purpose: show how to redirect output for another program
 *         idea: fork, then in the child, redirect output, then exec
 */
#include        <stdio.h>

main()
{
        int     pid ;
        int     fd;

        printf("About to run who into a file\n");

        /* create a new process or quit */
        if( (pid = fork() ) == -1 ){
                perror("fork"); exit(1);
        }
        /* child does the work */
        if ( pid == 0 ){
                close(1);                               /* close, */
                fd = creat( "userlist", 0644 );         /* then open */
                execlp( "who", "who", NULL );           /* and run      */
```

```
                perror("execlp");
                exit(1);
        }
        /* parent waits then reports */
        if ( pid != 0 ){
                wait(NULL);
                printf("Done running who.  results in userlist\n");
        }
}
```

Summary of Redirection to Files

- Standard input, output, and error are file descriptors 0, 1, and 2.
- The kernel always uses the lowest numbered unused file descriptor.
- The set of file descriptors is passed unchanged across exec calls.

Programming PIPES

A pipe is a data queue in the kernel with each end attached to a file descriptor. Pipes are created by *pipe* system call

```
NAME
       pipe - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int filedes[2]);

DESCRIPTION
       pipe  creates a pair of file descriptors, pointing to a pipe inode, and
       places them in the array pointed to  by  filedes.   filedes[0]  is  for
       reading, filedes[1] is for writing.

RETURN VALUE
       On  success,  zero is returned.  On error, -1 is returned, and errno is
       set appropriately.
```

A demonstration of PIPE in a process

```
/*  pipedemo.c  * Demonstrates: how to create and use a pipe
 *             * Effect: creates a pipe, writes into writing
 *               end, then runs around and reads from reading
 *               end.  A little weird, but demonstrates the idea.
 */
#include        <stdio.h>
#include        <unistd.h>

main()
{
        int     len, i, apipe[2];       /* two file descriptors */
        char    buf[BUFSIZ];            /* for reading end      */

        /* get a pipe */
        if ( pipe ( apipe ) == -1 ){
                perror("could not make pipe");
                exit(1);
        }
        printf("Got a pipe! It is file descriptors: { %d %d }\n",
                                                apipe[0], apipe[1]);

        /* read from stdin, write into pipe, read from pipe, print */
```

```
        while ( fgets(buf, BUFSIZ, stdin) ){
                len = strlen( buf );
                if (  write( apipe[1], buf, len) != len ){      /* send */
                        perror("writing to pipe");              /* down */
                        break;                                  /* pipe */
                }
                for ( i = 0 ; i<len ; i++ )                     /* wipe */
                        buf[i] = 'X' ;
                len = read( apipe[0], buf, BUFSIZ ) ;           /* read */
                if ( len == -1 ){                               /* from */
                        perror("reading from pipe");            /* pipe */
                        break;
                }
                if ( write( 1 , buf, len ) != len ){            /* send  */
                        perror("writing to stdout");            /* to    */
                        break;                                  /* stdout */
                }
        }
}
```

Using fork to share a PIPE

```
/* pipedemo2.c  * Demonstrates how pipe is duplicated in fork()
 *              * Parent continues to write and read pipe,
 *                but child also writes to the pipe
 */
#include        <stdio.h>

#define CHILD_MESS      "I want a cookie\n"
#define PAR_MESS        "testing..\n"
#define oops(m,x)       { perror(m); exit(x); }

main()
{
        int     pipefd[2];              /* the pipe     */
        int     len;                    /* for write    */
        char    buf[BUFSIZ];            /* for read     */
        int     read_len;

        if ( pipe( pipefd ) == -1 )
                oops("cannot get a pipe", 1);

        switch( fork() ){
                case -1:
                        oops("cannot fork", 2);
```

```
                /* child writes to pipe every 5 seconds */
                case 0:
                        len = strlen(CHILD_MESS);
                        while ( 1 ){
                                if (write( pipefd[1], CHILD_MESS, len) != len )
                                        oops("write", 3);
                                sleep(5);
                        }

                /* parent reads from pipe and also writes to pipe */
                default:
                        len = strlen( PAR_MESS );
                        while ( 1 ){
                                if ( write( pipefd[1], PAR_MESS, len)!=len )
                                        oops("write", 4);
                                sleep(1);
                                read_len = read( pipefd[0], buf, BUFSIZ );
                                if ( read_len <= 0 )
                                        break;
                                write( 1 , buf, read_len );
                        }
        }
}
```

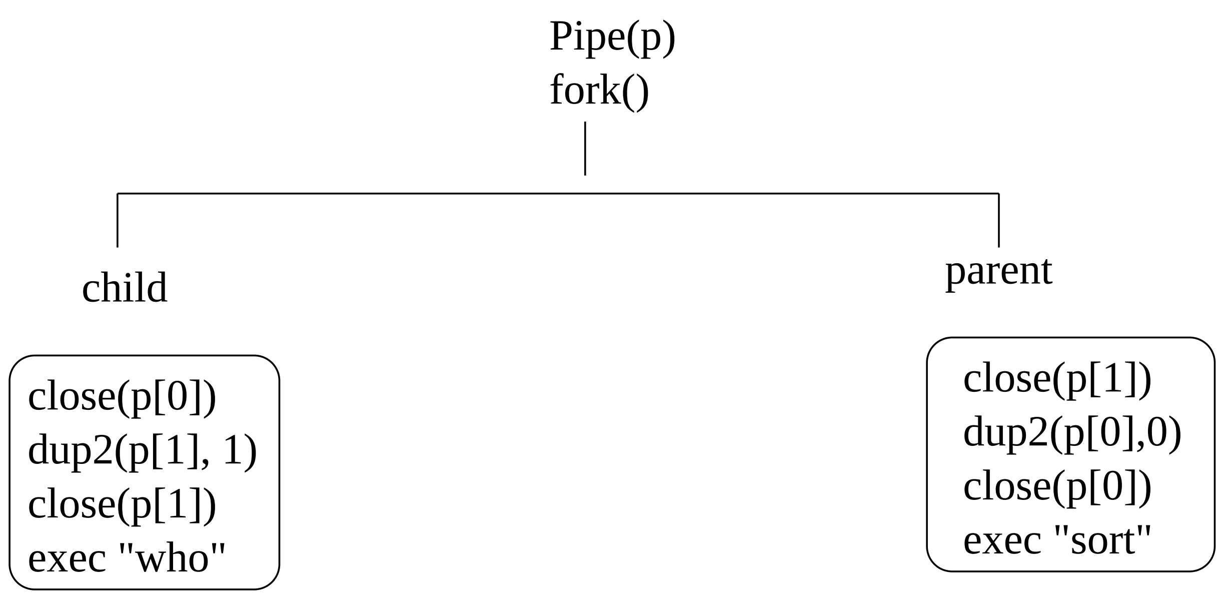
How pipes can connect the standard I/O of two process

If we want to write a program to create a pipe between two applications, for example

`pine who sort`

`pipe ls head`

We can use the following logic.



How pipes can connect the standard I/O of two process

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

#define oops(m,x)       { perror(m); exit(x); }

main(int ac, char **av)
{
        int     thepipe[2],                     /* two file descriptors */
                newfd,                          /* useful for pipes     */
                pid;                            /* and the pid          */

        if ( ac != 3 ){
                fprintf(stderr, "usage: pipe cmd1 cmd2\n");
                exit(1);
        }
        if ( pipe( thepipe ) == -1 )            /* get a pipe           */
                oops("Cannot get a pipe", 1);

        /* ------------------------------------------------------------ */
        /*      now we have a pipe, now let's get two processes         */
```

```
if ( (pid = fork()) == -1 )                          /* get a proc   */
        oops("Cannot fork", 2);

/* ------------------------------------------------------------ */
/*      Right Here, there are two processes                      */
/*              parent will read from pipe                       */

if ( pid > 0 ){                       /* parent will exec av[2]       */
        close(thepipe[1]);            /* parent doesn't write to pipe */

        if ( dup2(thepipe[0], 0) == -1 )
                oops("could not redirect stdin",3);

        close(thepipe[0]);            /* stdin is duped, close pipe   */
        execlp( av[2], av[2], NULL);
        oops(av[2], 4);
}

/*       child execs av[1] and writes into pipe                  */

close(thepipe[0]);                    /* child doesn't read from pipe */

if ( dup2(thepipe[1], 1) == -1 )
        oops("could not redirect stdout", 4);
```

```
        close(thepipe[1]);              /* stdout is duped, close pipe  */
        execlp( av[1], av[1], NULL);
        oops(av[1], 5);
}
```

Some properties of pipes

Reading from pipes

- read on a pipe blocks until data is available.
- When no writer to the pipe exists, read return EOF.
- Once data is read, it is removed from the pipe. So multiple readers can cause trouble.

Writing to pipes

- write will block until there is space
- write guarantees a minimum chunk size
- write fails if no readers.

The data in a pipe can only flow from one end to the other end. So bi-direction communication with a pipe is not possible.