```
:::::::::::::: watch.sh ::::::::::::::
#!/bin/sh
#
#  watch.sh  - a simple version of the watch utility, written in sh
#
        who | sort > prev               # get initial user list
        while true                      # true is a program: exit(0);
        do
                sleep 30                # wait a while
                who | sort > current    # get current user list
                echo "Logged out:"      # print header
                comm -23 prev current   # and results
                echo "Logged in:"       # header
                comm -13 prev current   # and results
                mv current prev
        done

:::::::::::::: whotofile.c ::::::::::::::

#include        <stdio.h>

/*
 * whotofile.c
 *      purpose show how to redirect output for another program
 *         idea fork, then in the child, redirect output, then exec
 */

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");
        }
}
```

```
:::::::::::::: stdinredir1.c ::::::::::::::
#include       <stdio.h>
#include       <fcntl.h>

/* stdinreader1.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 stdandard input
 */
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("data", 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 );
}

:::::::::::::: stdinredir2.c ::::::::::::::
#include       <stdio.h>
#include       <fcntl.h>
/*
 * stdinreader2.c
 *      shows two more methods for redirecting standard input
 *      use #define to set one or the other
 */
#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 );
}
```

```
:::::::::::::: redirect.c ::::::::::::::
#include       <stdio.h>
/**
 **     redirect.c     * Demonstrates how a program (like the shell)
 **                      redirects input from a file for another program
 **
 **                    * first arg is name of file to use for stdin
 **                      rest of args is command line
 **
 **                    * usage:  redirect filename cmd [arg  ..]
 **                      equivalent to   cmd [arg ..] < filename
 **
 **                    * exercise: convert this to redirect output, too
 **/
#define TRUE    1
#define FALSE   0

main( ac , av )
char **av;
{
       if ( ac < 3 ){
              fprintf( stderr, "usage: redirect filename cmd [arg..]\n");
              exit(1);
       }
       if ( setstdin( av[1] ) == FALSE )
              exit(1);

       run_command( ac-2 , av+2 );          /* this will exec */
}

run_command(ac, av)
char **av;
/**
 **     run the command and args in av.  convert to null-terminated array
 **     no return.  should just exec.
 **/
{
       char    **newav, *malloc();
       int     i;

       if ( ( newav=(char **) malloc((ac+1) * sizeof ( char * )) ) == NULL ){
              fprintf(stderr, "redirect: out of memory\n");
              exit(1);
       }
       for (i=0;i<ac;i++)                   /* copy current args   */
              newav[i] = av[i];
       newav[i] = NULL ;                    /* and NULL terminate  */

       execvp( av[0] , newav );             /* run new program     */
       perror( "Cannot exec command" );     /* or tell why         */
       exit(2);                             /* and exit            */
}

setstdin( fname )
char *fname ;
/*
 *     set standard input (fd 0) to named file
 *     returns TRUE if it worked, else returns FALSE
 */
{
       int     newfd;               /* result of open, should be 0        */
       int     retval = FALSE ;     /* pessimism                          */

       close( 0 ) ;                 /* ignore error if not open   */
       newfd = open( fname , 0 );   /* open the file. should be 0 */
       if ( newfd == -1 )           /* oops */
              perror( fname );      /* system tells what's wrong  */
       else if ( newfd != 0 )       /* didn't work.. hmm   */
              fprintf( stderr, "New file was not 0..\n" );
       else
              retval = TRUE ;               /* it worked!                 */
       return retval ;
}
```

```
:::::::::::::: pipedemo.c ::::::::::::::
#include       <stdio.h>

/**
 **    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.
 **
 **/

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

        /*
         *      first, get a pipe from the operating system
         */

        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]);

        /*
         *      then, write each arg down the pipe's writing end ( [1] )
         *      and read words back from the reading end
         */

        while ( gets(buf) ){                         /* get next line */
                len = strlen( buf );
                if (  write( apipe[1], buf, len) != len ){   /* send */
                        perror("writing to pipe");           /* down */
                        break;                               /* pipe */
                }
                for ( i = 0 ; i<len ; i++ )
                        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 */
                }
                write( 1, "\n", 1 );
        }
}
```

```
:::::::::::::: pipedemo2.c ::::::::::::::
#include        <stdio.h>

/**
 **     pipedemo2.c     * Demonstrates how pipe is duplicated in fork()
 **                     * Parent continues to write and read pipe,
 **                       but child also writes to the pipe
 **/

#define CHILD_MESS      "I want a cookie"
#define PAR_MESS        "testing.."

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

        if ( pipe( pipefd ) == -1 ){
                perror("cannot get a pipe");
                exit(1);
        }

        switch( fork() ){

                case -1:
                        fprintf(stderr,"cannot fork");
                        exit(1);

                case 0:                 /* child                */
                                        /* write to pipe        */
                                        /* every 5 seconds      */
                        len = strlen(CHILD_MESS);
                        while ( 1 ){
                                if (write( pipefd[1], CHILD_MESS, len) != len )
                                        exit(2);
                                sleep(5);
                        }

                default:                /* parent               */
                                        /* read and write pipe */
                        len = strlen( PAR_MESS );
                        while ( 1 ){
                                if ( write( pipefd[1], PAR_MESS, len)!=len )
                                        exit(3);
                                sleep(1);
                                read_len = read ( pipefd[0], buf, BUFSIZ );
                                if ( read_len <= 0 )
                                        break;
                                write( 1 , buf, read_len );
                                write( 1, "\n", 1 );
                        }
        }
}
```

```
:::::::::::::: pipedemo3.c ::::::::::::::
#include       <stdio.h>

/**
 **    pipedemo3.c    * Demonstrates how pipe is duplicated in fork()
 **                   * shows more detailed error handling and
 **                     diagnosis of eof file conditions.
 **/

#define CHILD_MESS     "I want a cookie"
#define PAR_MESS       "testing.."

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

        if ( pipe( pipefd ) == -1 ){
                perror("cannot get a pipe");
                exit(1);
        }

        switch( fork() ){

                case -1:
                        fprintf(stderr,"cannot fork");
                        exit(1);

                case 0:                 /* child                */
                                        /* write to pipe        */
                                        /* every 5 seconds      */
                        times = 5 ;
                        len = strlen(CHILD_MESS);
                        close(pipefd[0]);
                        while ( times-- ){
                                if (write( pipefd[1], CHILD_MESS, len) != len )
                                {
                                        perror("write on pipe failed");
                                        exit(2);
                                }
                                sleep(2);
                        }
                        write(pipefd[1], "bye\n", 4);
                        close(pipefd[1]);
                        exit(0);

                default:                /* parent               */
                                        /* read and write pipe */
                        close( pipefd[1] );
                        len = strlen( PAR_MESS );
                        times = 3 ;     /* all i'll take from that pipe */
                        while ( times-- ){
                                /*if ( write( pipefd[1], PAR_MESS, len)!=len )
                                        exit(3); */
                                read_len = read ( pipefd[0], buf, BUFSIZ );
                                if ( read_len == 0 ){
                                        printf("say EOF on pipe.bye\n");
                                        exit(0);
                                }
                                if ( read_len < 0 ){
                                        perror("read on pipe");
                                        exit(1);
                                }
                                write( 1 , buf, read_len );
                                write( 1, "\n", 1 );
                        }
                        close(pipefd[0]);
                        wait(&child_death_status);
                        printf("child died with %d\n", child_death_status);
                        exit(0);
        }
}
```

```
::::::::::::: pipe.c :::::::::::::
#include        <stdio.h>

/**
 *      pipe.c          * Demonstrates how to create a pipeline from
 *                        one process to another
 *
 *                      * Takes two args, each a command, and connects
 *                        av[1]'s output to input of av[2]
 *                      * usage: pipe command1 command2
 *                        effect: command1 | command2
 *                      * Limitations: commands do not take arguments
 *                      * uses execlp() since known number of args
 *                      * Note: exchange child and parent and watch fun
 **/

main(ac, av)
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          */
                perror( "cannot create pipe" );
                exit(1);                        /* or exit             */
        }

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

        if ( (pid = fork()) == -1 ){                    /* get a proc  */
                fprintf(stderr,"cannot fork\n");
                exit(1);                                /* or exit     */
        }

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

        if ( pid > 0 ){                         /* the child will be av[2]    */
                close(thepipe[1]);      /* close writing end          */
                close(0);               /* will read from pipe        */
                newfd=dup(thepipe[0]); /* so duplicate the reading end      */
                if ( newfd != 0 ){      /* if not the new stdin..     */
                        fprintf(stderr,"Dupe failed on reading end\n");
                        exit(1);
                }
                close(thepipe[0]);      /* stdin is duped, close pipe */
                execlp( av[2], av[2], NULL);
                exit(1);                /* oops                       */
        }

        /*
         *      child will write into writing end of pipe
         */
        close(thepipe[0]);      /* close reading end          */
        close(1);               /* will write into pipe               */
        newfd=dup(thepipe[1]); /* so duplicate writing end   */
        if ( newfd != 1 ){      /* if not the new stdout..    */
                fprintf(stderr,"Dupe failed on writing end\n");
                exit(1);
        }
        close(thepipe[1]);      /* stdout is duped, close pipe        */
        execlp( av[1], av[1], NULL);
        exit(1);                /* oops                       */
}
```