Preview

- Interprocess Communication with Pipe
  - Pipe from the Parent to the child
  - Pipe from the child to the parent
  - popen() with "r"
  - Popen() with "w"

Interprocess Communication with Pipe

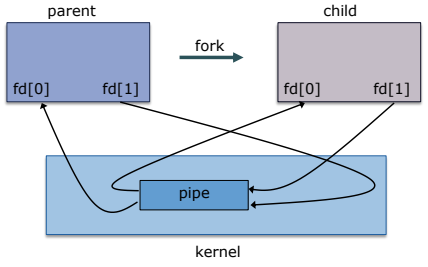
- Pipes are the oldest form of UNIX system Interprocess Communication (IPC).
- Pipes have two limitations
  - Historically, pipes are half duplex – data flows in only one direction. (Some system provide full duplex pipes.)
  - Pipes can be used only between processes that have a common ancestor.
    - Usually, a pipe is used between the parent and the child.

Interprocess Communication with Pipe

```
#include <unistd.h>
int pipe (int filedes[2]);
                    Return 0 if OK, -1 on error
```

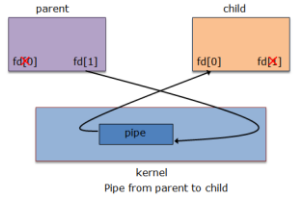
- Two file descriptors are returned through the filedes[] arguments.
  - **filedes[0]** is open for reading
  - **filedes[1]** is open for writing
- In typical use, a process creates a pipe just before it forks one or more child processes.
- The pipe is then used for communication either between the parent or child processes, or between two sibling processes

Half-Duplex Pipe After a Fork



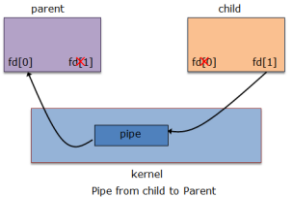
Interprocess Communication with Pipe

- For sending a data through a pipe from the parent to a child
  - parent closes fd[0] (the read end of the pipe),
  - child close fd[1] (the write end of the pipe)



Interprocess Communication with Pipe

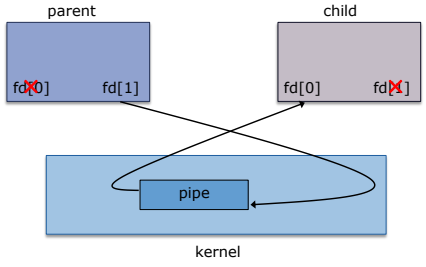
- For sending a data through a pipe from the child to the parent
  - parent closes fd[1] (the write end of the pipe),
  - child close fd[0] (the read end of the pipe)



Interprocess Communication with Pipe

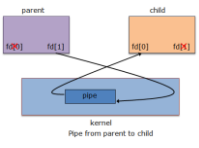
- When one end of a pipe is closed, two rules apply.
  - If we read from a pipe whose end has been closed, read returns 0 to indicate an end of file after all data has been read.
  - If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. It can be ignored or catch and return from the signal handler.

Pipe from the Parent to the Child



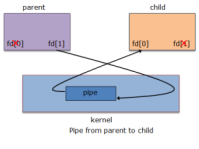
Pipe from parent to child


```c
/* pipe1.c demonstrate a pipe from the parent to the child */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256
void err_sys(char *);

int main()
{
        int n, fd[2];
        pid_t pid;
        char line[MAXLINE];

        if (pipe(fd) < 0) /* creat a pipe */
                err_sys("pipe error");

        if ( (pid = fork()) < 0) /*creat a child */
                err_sys("fork error");

        else if (pid > 0) /* parent */
        {
                close(fd[0]);
                sleep (5);
                write(fd[1], "hello world\n", 12); /*write to a pipe */
        }
        else /* child */
        {
                close(fd[1]);
                n = read(fd[0], line, MAXLINE); /*read from the pipe */
                write(STDOUT_FILENO, "my mom said ", 12);
                write(STDOUT_FILENO, line, n);
        }

        exit(0);
}

void err_sys(char *str)
{
   printf ("%s \n",str);
   exit (1);
}
```



Pipe from parent to child

```c
/* pipe2.c */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
/* function prototypes */
void read_from_pipe (int);
void write_to_pipe (int);
void err_sys(char *);

int main (void)
{
        pid_t pid;
        int mypipe[2];

        if (pipe (mypipe) < 0) /* Create the pipe. */
                err_sys("pipe failed");
        if ( (pid = fork()) < 0) /*creat a child */
                err_sys("fork error");
        else if (pid > 0) /* parent */
        {
                close (mypipe[0]);
                write_to_pipe (mypipe[1]);
        }
        else /*the child process */
        {
                close (mypipe[1]);
                read_from_pipe (mypipe[0]);
        }
        exit (0);
}
```

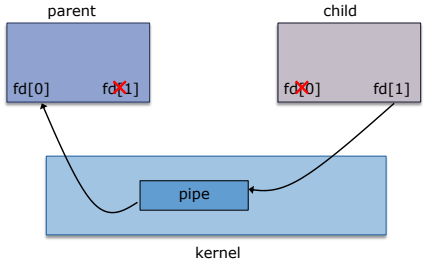


Pipe from parent to child

Pipe from the Parent to the Child

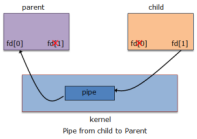
```c
/* Read characters from the pipe and echo them to stdout. */
void read_from_pipe (int fd)
{
        FILE *stream;
        int c;
        /* The fdopen() associates a stream with the existing file descriptor, fd */
        stream = fdopen (fd, "r");
        while ((c = fgetc (stream)) != EOF) /* Read a character from the stream untilEOF */
                putchar (c);                /* Write on standard output */
        fclose (stream);
}
/* Write some random text to the pipe. */
void write_to_pipe (int fd)
{
        FILE *stream;

        stream = fdopen (fd, "w"); /* open a pipe for writing */
        fprintf (stream, "hello, world!\n");
        fprintf (stream, "goodbye, world!\n");
        fclose (stream);
}

void err_sys(char *str)
{
   printf ("%s \n",str);
   exit (1);
}
```


Pipe from the Child to the Parent



Pipe from child to Parent

```c
/* pipe3.c: demonstrate pipe from the child to the parent */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXLINE 256
void err_sys(char *);

int main()
{
        int         n, fd[2];
        pid_t pid;
        char line[MAXLINE];
        if (pipe(fd) < 0) /* creat a pipe */
                err_sys("pipe error");
        if ( (pid = fork()) < 0) /*creat a child */
                err_sys("fork error");
        else if (pid > 0) /* parent */
        {
                close(fd[1]);
                n = read(fd[0], line, MAXLINE); /*read from the pipe */
                write(STDOUT_FILENO, "My baby said ", 13);
                write(STDOUT_FILENO, line, n);
        }
        else /* child */
        {
                close(fd[0]);
                write(fd[1], "hello world\n", 12); /*write to a pipe */
        }
        exit(0);
}

void err_sys(char *str)
{
   printf ("%s \n",str);
   exit (1);
}
```




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

#define MAXLINE 256
void err_sys(char *);

int main()
{
        int         n, fd[2], size, int1, int2;
        pid_t pid;
        char sline[MAXLINE], rline[MAXLINE];

        if (pipe(fd) < 0) /* creat a pipe */
                err_sys("pipe error");

        if ( (pid = fork()) < 0) /*creat a child */
                err_sys("fork error");

        else if (pid > 0) /* parent use the pipe to send message to child*/
        {
            close(fd[0]); /* fd[1] is used for sending data */
            printf("submit two integers\n");
            while ( (size = read(STDIN_FILENO, sline, MAXLINE)) > 0)
            {
                write(fd[1], sline, size); /*write to a pipe */
                printf("submit two integers\n");
            }
        }
        else /* child use pipe to receive message from the parent */
        {
            close(fd[1]);
            while ((n = read(fd[0], rline, MAXLINE)>0) /*read from the pipe */
            {
                /* chose first two string as two integer */
                if (sscanf(rline, "%d%d", &int1, &int2) == 2)
                {
                    sprintf(rline, "%d\n", int1 + int2);
                    n = strlen(rline);
                    if (write(STDOUT_FILENO, rline, n) != n)
                        err_sys("write error");
                }
                else /* if first two string is not integer */
                {
                    if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                        err_sys("write error");
                }
            }
        }
        return 0;
}

void err_sys(char *str)
{
  printf ("%s \n",str);
  exit (1);
}
```

popen(), pclose() function

- These two function handle all work that we've been doing ourselves for using a pipe.
  - Creating a pipe
  - Forking a child,
  - Closing the unused ends of the pipe,
  - Executing a shell to run the command, and
  - Waiting for the command to terminate

popen(), pclose() function

```
#include <stdio.h>
FILE *popen(const char *cmdstring, const char *mode);
                        Return a file pointer if OK, NULL on error
int pclose(FILE *fp);
                        Return termination status, -1 on error
```

- The popen() function does fork and ***exec*** to execute the ***cmdstring*** and returns a standard I/O file pointer.
- If mode = "r", the file pointer is connected to the standard output of the ***cmdstring***.
- If mode = "w", the file pointer is connected to the standard inputs of the ***cmdstring***.

exec System Call

- By using exec system call, a child process can execute another program.
- Once a process call a exec system call, that process is completely replaced by the new program.
- The new program starts executing at its main function. The main function might need arguments.
- The process ID does not change across an exec system call, since it is not created.
- The content of text, data, heap and stack segment will be replaced by new program.

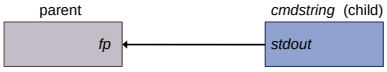
exec System Call

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ... /*, (char *)0, char *const envp[]*/);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);

Return -1 on error, no return on success
```

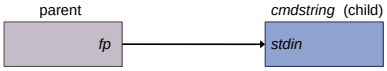
Six system call can be recognized by
- Argument list or Argument vector
- File name or path name
- With or without environment

popen(), pclose() function

fp = popen (cmdstring, "r");



fp = popen (cmdstring, "w");

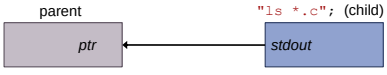


popen(), pclose() function

```c
/* popen.c demonstrate the popen function. */
#include <stdio.h>
#include <stdlib.h>

int main( )
{
        char *cmd = "ls *.c";
        char buf[BUFSIZ]; /* BUFSIZ =1024 defined in stdio.h */
        FILE *ptr;

        if ((ptr = popen(cmd, "r")) != NULL)
                while (fgets(buf, BUFSIZ, ptr) != NULL)
                        (void) printf("%s", buf);
        pclose(ptr);

        return 0;
}
```


popen(), pclose() function

```c
char *cmd = "ls *.c";
ptr = popen(cmd, "r");
```

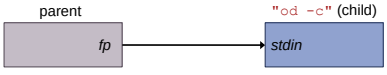


popen(), pclose() function

```c
/* popen1.c: demonstrate popen with "w" */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
        FILE *wfp;
        char buffer[BUFSIZ + 1];

        sprintf(buffer,"This is testing popen with w \n");
        /* octal dump:displays contents as octal numbers */
        wfp = popen("od -c ", "w");
        if (wfp != NULL)
        {
                fwrite(buffer, sizeof(char), strlen(buffer), wfp);
                pclose(wfp);
                exit (0);
        }
        exit (1);
}
```

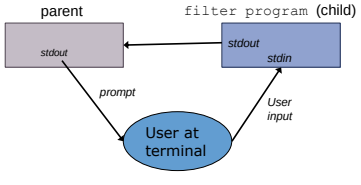

popen(), pclose() function

```c
wfp = popen("od -c ", "w");
```



popen(), pclose() function

- An application that write a prompt to standard output and read a line from standard input.
- We can interpose a program between the application and its input to transform the input.



```c
/* popen2.c function demonstrate an popen function */
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

#define MAXLINE 256
void err_sys(char *);

int main(void)
{
        char    line[MAXLINE];
        FILE    *fpin; /* use pointer to read from stdin */

        if ( (fpin = popen("./some", "r")) == NULL)
                err_sys("popen error");
        for ( ; ; )
        {
                fputs("prompt> ", stdout);
                fflush(stdout);
                if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
                        break;
                if (fputs(line, stdout) == EOF) /* write to stdout */
                        err_sys("fputs error to pipe");
        }
        if (pclose(fpin) == -1)
                err_sys("pclose error");
        putchar('\n');
        exit(0);
}
void err_sys(char *s)
{
        printf ("%s \n", s);
        exit (1);
}
```

popen(), pclose() function

```c
/* some.c function change upper case character string to lower case */
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

void err_sys(char *);
int main()
{
        int c;
        while ( (c = getchar()) != EOF)
        {
                if (isupper(c))
                        c = tolower(c);
                if (putchar(c) == EOF)
                        err_sys("output error");
                if (c == '\n')
                        fflush(stdout);
        }
        exit(0);
}

void err_sys(char *s)
{
        printf ("%s \n", s);
        exit (1);
}
```