Preview

- The popen() ,pclose()
- Corprocess
- FIFOs
- XSI IPC
  - Message Queue
  - Semaphore
  - Shared Memory

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

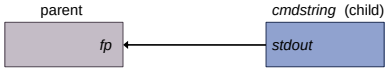
popen(), pclose() function

```
#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
```

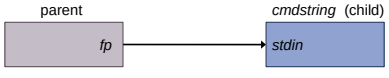
- The popen() function does fork and ***exec*** to execute the ***cmdstring*** and returns a standard I/O file pointer.
- If mode = "r", the file pointer is connected to the standard output of the ***cmdstring***.
- If mode = "w", the file pointer is connected to the standard inputs of the ***cmdstring***.

popen(), pclose() function

fp = popen (cmdstring, "r");
OS create a pipe and child process
Child process runs cmdstring and send output through the pipe



fp = popen (cmdstring, "w");
OS create a pipe and child process
The parent process send input through the pipe and child process runs cmdstring with input through the pipe



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];
        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");
```

OS create a pipe and child process
Child process runs ls *.c shell command and send output through the pipe

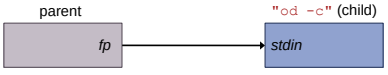
popen(), pclose() function

```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

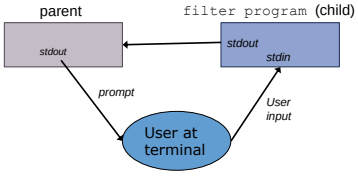
```c
wfp = popen("od -c ", "w");
```

OS create a pipe and child process
The parent process send input through the pipe and child process runs shell command "od –c" with input through the pipe



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 pipe */
        /* OS create a pipe for read. Create a child and let it run program some */
        /* the parent is ready to read from the child */
        if ( (fpin = popen("./some", "r")) == NULL)
                err_sys("popen error");
        for ( ; ; )
        {
                fputs("prompt> ", stdout); /* display prompt> on screen */
                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);
}
```

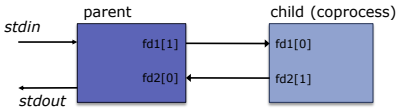
popen(), pclose() function

```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) /* read input from keyboard */
        {
                if (isupper(c)) /* change to capital to lower case */
                        c = tolower(c);
                if (putchar(c) == EOF) /* write on standard output until ctr-D */
                        err_sys("output error");
                if (c == '\n')
                        fflush(stdout);
        }
        exit(0);
}

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


Coprocess

- A **filter** is a program that reads from standard input and write to standard output.
- A filter becomes a **coprocess** when the same program generates the filter's input and read the filter's output.
- With a **coprocess**, we have two one-way pipes to the other process: one to its standard input and one from its standard output.

Coprocess




```c
/* add2.c : get two integer from std input and calcuate sum
   and write to standard output*/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define MAXLINE 256
void err_sys(char *);
int main(void)
{
        int  n, int1, int2;
        char line[MAXLINE];

        /* read a string from stdin */
        while ( (n = read(STDIN_FILENO, line, MAXLINE)) > 0)
        {
                /* chose first two string as two integer and save as integers*/
                if (sscanf(line, "%d%d", &int1, &int2) == 2)
                {
                        /* write the result of sum to buffer*/
                         sprintf(line, "%d\n", int1 + int2);
                         n = strlen(line);
                        /* write content of buffer to standard output*/
                         if (write(STDOUT_FILENO, line, n) != n)
                                err_sys("write error");
                }
                else /* if first two string are not integer */
                {
                         if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                                err_sys("write error");
                }
        }
        exit(0);
}
void err_sys(char *str)
{
   printf ("%s \n",str);
   exit (1);
}
```

```c
/* parent.c demonstrate coprocessor */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define MAXLINE 256
void err_sys(char *);

int main(void)
{
        int n, fd1[2], fd2[2];
        pid_t pid;
        char line[MAXLINE];
        if (pipe(fd1) < 0 || pipe(fd2) < 0) /* create two pipe */
                err_sys("pipe error");

        if ( (pid = fork()) < 0) /* create a child */
                err_sys("fork error");
        else if (pid > 0) /* for parent */
        {
                close(fd1[0]); /* since the first pipe is used for std output */
                close(fd2[1]); /* since the second pipe is used for std input */
                printf("two integers");
                while (fgets(line, MAXLINE, stdin) != NULL) /* read from standard input */
                {
                        n = strlen(line);
                        if (write(fd1[1], line, n) != n) /* write to the first pipe */
                                err_sys("write error to pipe");
                        if ( (n = read(fd2[0], line, MAXLINE)) < 0) /* read from the second pipe */
                                err_sys("read error from pipe");
                        if (n == 0) {
                                err_sys("child closed pipe");
                                break;
                        }
                        line[n] = '\0';         /* null terminate */
                        if (fputs(line, stdout) == EOF) /* write to standard output*/
                                err_sys("fputs error");
                }
                if (ferror(stdin))
                        err_sys("fgets error on stdin");
                exit(0);
        }
        else
        }
```

Coprocess

```c
        else
        {
                /* child */
                close(fd1[1]); /*since first pipe is used for std input */
                close(fd2[0]); /*since second pipe is used for std output */
                if (execl("./add2", "add2", (char *) 0) < 0) /* execute addr2 */
                        err_sys("execl error");
        }
}
```

FIFOs

- **Pipe**s can be used only between related processes when a common ancestor has created the pipe.
- **FIFO**s allow two unrelated processes to communicate with each other.
- Since **FIFO** is a type of file, creating a FIFO is similar to creating a file.
- Two unrelated processes can open a FIFO and begin communication.

FIFOs

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
                                    Return 0 if Ok, -1 error
```

- The specification of mode argument is the same as for the open system call.
- The rule for user and group ownership of a **FIFO** are the same as in a file.
- Once we create a **FIFO** by using mkfifo, we can open it by using open().

FIFOs

- A FIFO supports blocked read and write operations by default: if a process opens the FIFO for reading, it is blocked until another process opens the FIFO for writing, and vice versa.
- However, it is possible to make FIFOs support non-blocking operations by specifying the O_NONBLOCK flag while opening them.
- A FIFO must be opened either read-only or write-only. It must not be opened for read-write because it is half-duplex, that is, a one-way channel.

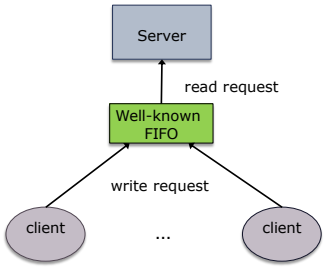
FIFOs

- In the normal case (O_NONBLOCK not specified)
  - An FIFO open for read-only blocks until some other process opens the FIFO for writing.
  - An FIFO open for write-only blocks until some other process open the FIFO for reading.
- If O_NONBLOCK is specified
  - An open for read-only returns immediately.
  - An open for write-only returns –l with errno set to ENXIO if no process has the FIFO open for reading.

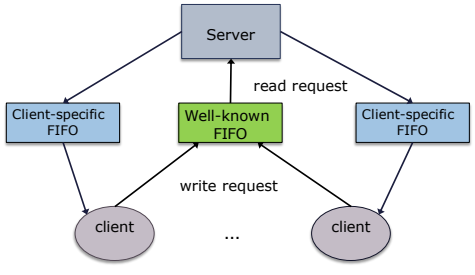
FIFOs

- **FIFO**s are used by shell commands to pass data from one shell pipleline to another without creating intermediate temporary files.
- **FIFO**s are used as rendezvous point in client-server applications to pass data between the clients and the servers.

FIFOs



FIFOs




```c
/* server.c create a FIFO to communicate with client*/
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#define HALF_DUPLEX  "halfduplex"
#define MAX_BUF_SIZE 255

int main(int argc, char *argv[])
{
    int fd, ret_val, count, numread;
    char buf[MAX_BUF_SIZE];
    /* Create the FIFO(named - pipe) */
    ret_val = mkfifo(HALF_DUPLEX, 0666);
    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);
    }
    /* Open the FIFO for reading */
    fd = open(HALF_DUPLEX, O_RDONLY);
    /* Read from the FIFO */
    numread = read(fd, buf, MAX_BUF_SIZE);
    buf[numread] = '\0';
    printf("Half Duplex Server : Read From the pipe : %s\n", buf);
    /* Convert to the string to upper case */
    count = 0;
    while (count < numread) {
        buf[count] = toupper(buf[count]);
        count++;
    }
    printf("Half Duplex Server : Converted String : %s\n", buf);
    return 0;
}
```

FIFOs

```c
/* client.c write a string to FIFO */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#define HALF_DUPLEX "halfduplex"
#define MAX_BUF_SIZE 255

int main(int argc, char *argv[])
{
    int fd;
    /* Check if an argument was specified. */
    if (argc != 2) {
        printf("Usage : %s <string to be sent to the server>n", argv[0]);
        exit (1);
    }

    /* Open the FIFO for writing */
    fd = open(HALF_DUPLEX, O_WRONLY);

    /* Write to the pipe */
    write(fd, argv[1], strlen(argv[1]));
    return 0;
}
```

```c
/*server1.c which receive two integer through FIFO and calculate it's sum.
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#define HALF_DUPLEX        "halfduplex"
#define BFFERSIZE          20

int main(int argc, char *argv[])
{
    int fd, ret_val, count, numread;
    char line[BFFERSIZE];

    /* Create the named - pipe */
    ret_val = mkfifo(HALF_DUPLEX, 0666);

    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);
    }

    /* Open the FIFO for reading */
    fd = open(HALF_DUPLEX, O_RDONLY);
    int int1, int2, n, sum;
     while((n=read(fd, line, BFFERSIZE))>0)
     {
       line[n]='\0';
       if (sscanf(line, "%d %d", &int1, &int2)== 2)
       {
                sprintf(line, "The sum is %d\n", int1+int2);
                write(1, line, strlen(line));
       }
       else
                write(1, "invalid arguments\nEnter two integers\n", 37);
     }
    return 0;
}
```


FIFOs

```c
/*client1.c which send two integer through FIFO to server1.
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#define HALF_DUPLEX                "halfduplex"
#define BFFERSIZE      20

int main(int argc, char *argv[])
{
    int fd, n;
    char line[BFFERSIZE];


    /* Open the FIFO for writing */
    fd = open(HALF_DUPLEX, O_WRONLY);

     printf("Enter to integers\n Press Ctrl+D to exit\n");
      while(fgets(line, BFFERSIZE, stdin)!=NULL)
      {
         n=strlen(line);
         write(fd,line,n); //pass information from  to child through FIFO
      }
```


XSI Interprocess Communication

- There are three types of XSI IPC
  - Message queue
  - Semaphore
  - Shared memory
- Each IPC structure in the kernel is referred to by a non-negative identifier.
- When a given IPC structure is created and then removed, the identifier associated with that structure continually increase up to the maximum positive integer, and then wraps around to 0.

XSI Interprocess Communication

- When an XSI IPC structure is created (by calling msgget(), semget() or shmget()), a key must be specified.
- The data type **key_t** for a key is specified in the header file <sys/types.h>.
- XSI IPC associated with ipc_perm structure. This structure defines the permissions and owner and so on.

XSI Interprocess Communication

```
struct ipc_perm {
  uid_t uid; /* owner's effective user ID*/
  gid_t gid; /* owner's effective group ID */
  uid_t cuid; /* creator effective user ID */
  gid_t cgid; /* creator effective group ID */
  mode_t mode /* access mode */
};
```

XSI IPC (Message Queue)

- A message queue is a linked list of message stored within the kernel and identified by a message queue ID.
- A new message queue is created or opened by msgget().
- A new messages are added to the end of a queue by msgsnd()