PROGRAMMING LAB 2 COMMUNICATION BETWEEN PROCESSES

BEFORE YOU START...

Download pLab2.zip from course Moodle page to your local directory

Unzip pLab2.zip with any compression tool in your system.

Goto the pLab2 directory

For WSL, shift + right click

Assume you are using WSL or Docker, right click on pLab2 directory to open a Linux shell/Mac terminal

Then, execute the following command to compile all programs

- make

When you see TRY!

- Open a terminal and try these command(s)

CONTENTS

REFERENCES

Man pages

The GNU C Library

- http://www.gnu.org/software/libc/manual/html_node/index.html

Programming in C – UNIX System Calls and Subroutines using C by A. D. Marshall

- http://www.cs.cf.ac.uk/Dave/C/CE.html

IPC – SIGNALS (RECAP)

Signal handling

- When a process gets a signal, OS determines how the process will respond to that signal by examines that process's PCB
- Each signal is represented by a value/symbolic name
- Processes may catch, ignore or mask a signal
  - Catching a signal involves
    - specifying a routine (signal handler) that the OS will invoke when the process receives that signal , e.g., the signal() or sigaction() system call
  - Using OS's default action to handle the signal – could be the default handler or could be just ignore
  - Masking a signal is to instruct the OS not to deliver signals of that type **until** the process clears the signal mask
- SIGKILL and SIGSTP cannot be caught, blocked (masked) or ignored

SIGNAL( ) - SET NEW SIGNAL HANDLER

```
sighandler_t signal(int signum, sighandler_t handler);
```

Use for **setting up** a signal handler to **handle** a **specific signal** (signum)

- signum: the target signal number
- handler: can be one of the following values
  1. A signal handler written by you for handling the signal
     - the prototype of the new handler function *must be as follows*:

       ```
       void handler_name(int signum);
       ```
  2. SIG_IGN: the signal is ignored
  3. SIG_DFL: the default action is performed
- returns the signal handler that was previously in effect for that specific signum
- If failure, it returns SIG_ERR instead

TRY! ./lab2-signal

LAB2-SIGNAL.C

```
void sigint_handler(int signum) {
    printf("\nA signal SIGINT is caught\n\n");
}

int main() {
    printf("SIGINT can be CAUGHT once in the coming 10 seconds.\n");
    printf("Press Ctrl-c to try\n");
    signal(SIGINT, sigint_handler);

    sleep(10);

    printf("SIGINT is ignored in the coming 10 seconds.\n");
    printf("Press Ctrl-c to try\n");
    signal(SIGINT, SIG_IGN);

    sleep(10);

    printf("\n\nSIGINT is set to default action.\n");
    printf("Press Ctrl-c to kill me\n");
    signal(SIGINT, SIG_DFL);

    while (1) {
        sleep(10);
    }
}
```

sleep(*i*) makes the calling process sleep until ***i*** seconds have elapsed **or** a signal arrives which is not ignored.

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

SIGACTION( ) - SET NEW SIGNAL HANDLER

Another way to set a signal handler to handle a specific signal

- A little bit complicated, but more flexible and powerful

The basic steps in setting the handler:

1. retrieve the old sigaction structure for that target signal (e.g. SIGINT)
   ```
   sigaction(SIGINT, NULL, &act);
   ```
   - The current sigaction structure for SIGINT is saved in act

2. Modify the sigaction structure act with the new value/setting
   - Just need to modify the corresponding fields

3. Set the new sigaction structure for this signal
   ```
   sigaction(SIGINT, &act, NULL);
   ```

SIGACTION( )

struct sigaction

- Use for telling the system what to do for a specific signal
- This structure is defined in the <signal.h> header to include at least the following members:

```
struct sigaction {
    void (*sa_handler) (int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    ...
} act;
```

Method 1:
Install a handler similar to the one use in signal() function call

Method 2:
Install a handler that can have more info retrieved from the system

Specifies a set of signals to be masked while the handler is running

Specifies a set of flags which modify the behavior of the signal

SIGACTION( )

We have 2 ways in using sigaction() to install a new signal handling function:

- **Method 1**
  - We define a handling function, which has the same function signature as in the case of using signal()

```
void handler1 (int signum) {
    ...
}
```

  - Simply assign this function to the sa_handler field in the struct sigaction

TRY! ./lab2-sigact1

LAB2-SIGACT1.C

```
int n=0;  // a global variable

void sigint_handler1(int signum) {
    printf("signal %d is caught for %d times\n", signum, ++n);

    if (n == 3) {
        exit(0);
    }
}

int main() {
    struct sigaction sa;

    /* use sigaction to install a signal handler named sigint_handler1 */
    sigaction(SIGINT, NULL, &sa);
    sa.sa_handler = sigint_handler1;
    sigaction(SIGINT, &sa, NULL);

    printf("My process ID is %d\n", (int)getpid());
    printf("Press Ctrl-c 3 times to kill me\n");;
    while (1) {
        sleep(1);
    }
}
```

```
struct sigaction {
        void (*sa_handler) (int);
       void (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t sa_mask;
        int sa_flags;
        ...
} act;
```

SIGACTION( )

- **Method 2**
  - Set **sa_flags** = SA_SIGINFO
  - **sa_sigaction** is assigned with the new signal handling function pointer with signature

    ```
    void (*)(int, siginfo_t *, void *)
    ```

  - The new handling function is implemented as follows:

    ```
    void handler2(int signum, siginfo_t *sig, void *context) {
        ...
    }
    ```

  - With this method, more information could be passed to the signal handler by the system via the 2nd argument
    - **siginfo_t** is a struct consists of a number of fields
      - e.g. who sent the signal? sig.si_pid gives you the process ID of the sender
    - the 3rd argument is a pointer to a ucontext_t, but it is of **no real use** by the handler function

SIGACTION( )

TRY! **Method 2**

- ./lab2-sigact2

```
int n=0;  // a global variable

void sigint_handler2(int signum, siginfo_t *sig, void *v) {
    printf("signal %d (from %d) is caught for %d times\n", signum, sig->si_pid, ++n);
    if (sig->si_code == SI_USER)
        printf("It is sent by a user (kill command)\n");
    else if (sig->si_code == SI_KERNEL)
        printf("It is sent by the kernel\n");

    if (n == 3) {
        exit(0);
    }
}

int main() {
    struct sigaction sa;

    /* use sigaction to install a signal handler named sigint_handler1 */
    sigaction(SIGINT, NULL, &sa);
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = sigint_handler2;
    sigaction(SIGINT, &sa, NULL);

    printf("My process ID is %d\n", (int)getpid());
    printf("Press Ctrl-c 3 times to kill me\n");;
    while (1) {
        sleep(1);
    }
}
```

SIGNALS CLOSE TOGETHER MERGE TO ONE

If multiple signals of the same type are delivered to a process before invoking the signal handler, the handler may only be invoked once.

We cannot reliably use a signal handler to count how many times a signal been delivered.

```
int sigemptyset(sigset_t *set)

int sigaddset(sigset_t *set, int signum)
```

BLOCKING SIGNALS

Blocking a signal means telling the operating system to **hold it** and **deliver later**

Basic idea:

- Mask the signal for a process (usually before critical activity)
- The process works on that critical activity
- Unmask the signal to resume handling the signal

Use the following functions to mask a specific signal

- sigemptyset() - initializes a signal set variable to an empty set
- sigaddset() - add the target signal to the specified signal set variable
- sigpromask() - to mask the signal(s)

SIGPROCMASK()

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
```

sigprocmask() is used to fetch and/or change the signal mask of the calling process

The behavior of the call is dependent on the value of how, as follows:

- how = **SIG_BLOCK**
  - Add the signals in the input signal set to the current mask (i.e., union the existing set with the input set)
- how = **SIG_UNBLOCK**
  - Remove the signals in the input signal set from the current mask
- how = SIG_SETMASK
  - Replace the current set by the input set

If oldset is non-NULL, the previous value of the signal mask is stored in oldset

sigprocmask() returns 0 on success and -1 on error

BLOCKING SIGNAL

TRY! ./lab2-mask

```
void sigint_handler(int signum) {
    printf("\nA signal SIGINT is caught\n\n");
}

int main() {
  sigset_t blkset;

  printf("SIGINT can be CAUGHT once in the coming 10 seconds.\n");
  printf("Press Ctrl-c to try\n");
  signal(SIGINT, sigint_handler);

  sleep(10);

  printf("SIGINT is masked in the coming 10 seconds.\n");
  printf("Press Ctrl-c to try\n");
  sigemptyset(&blkset);
  sigaddset(&blkset, SIGINT);
  sigprocmask(SIG_BLOCK, &blkset, NULL);

  sleep(10);

  sigprocmask(SIG_UNBLOCK, &blkset, NULL);
  printf("\n\nSIGINT is set to default action.\n");
  printf("Press Ctrl-c to kill me\n");
  signal(SIGINT, SIG_DFL);

  while (1) {
      sleep(10);
  }
}
```

SIGNAL MESSAGES

```
char *strsignal(int signum)
```

strsignal() returns a pointer to a statically-allocated string containing a (system-specific) message describing the signal signum

- e.g. SIGKILL - Killed, SIGTERM- Terminated, SIGINT - Interrupt

```
void psignal(int signum, const char *s)
```

psignal() displays a message on stderr consisting of the string s, a colon, a space, a string describing the signal number signum, and a trailing newline.

IPC – PIPE

A pipe is a mechanism for interprocess communication

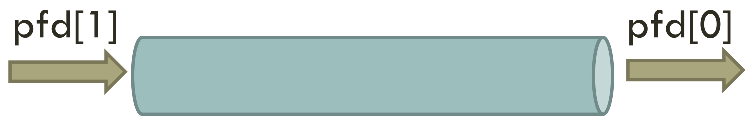
- Between parent ←→ child processes and between child processes
- Data written to the pipe by one process can be read by another process

A pipe is a unidirectional FIFO channel with one end as input and the other end as output

Parent process creates pipe(s) and child processes inherit both ends of the pipes

PIPE( ) - CREATING A PIPE

```
#include <unistd.h>
int pipe(int pfd[2]);
```



This function creates both the reading and writing ends of the pipe

stdin

Puts the file descriptors for the **reading** and **writing** ends of the pipe into **pfd[0]** and **pfd[1]** respectively

stdout

Returns

- 0 if successful
- -1 on failure

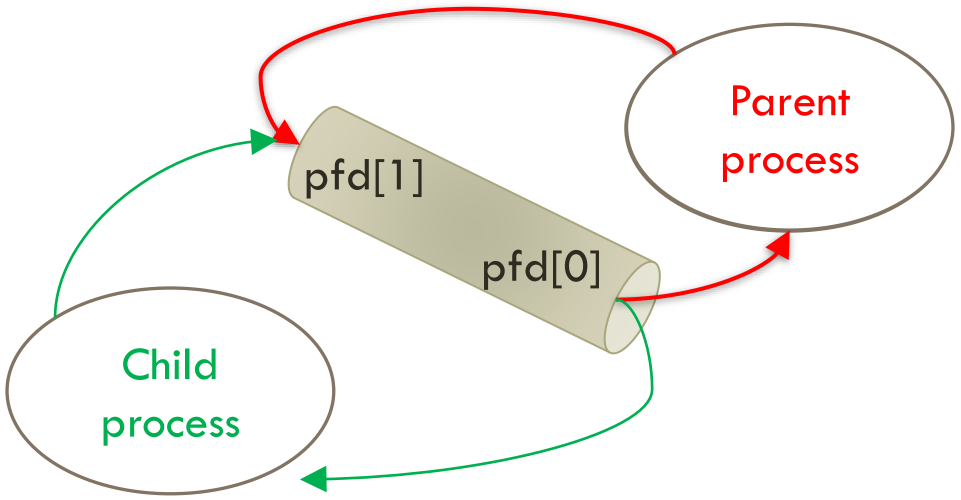
```
int main() {
    int pfd[2];
    pipe(pfd);
    printf("pfd[0]=%d, pfd[1]=%d\n", pfd[0], pfd[1]);
    return 0;
}
```

PIPE & FORK

Typically, parent process creates a pipe just **before it forks** one or more child processes

When we execute fork()

- **All opened files and memory variables are copied**
  - So as the file descriptors created by pipe()



PIPE PROPERTIES

If a process attempts to write to a full pipe, then **write blocks** until sufficient data has been read from the pipe to allow the write to complete.

- We can also set the behavior of write() towards full pipe to fail

TRY!

```
./lab2-pipefull
```

LAB2-PIPEFULL.C

```
int main() {
    char pattern[] = "0123456789abcdef";
    int pfd[2];
    int num = 0, i = 0;

    pipe(pfd);

    while (1) {
        num = write(pfd[1], pattern, 16);
         i += num;
        printf("%d\n", i);
    }

    return 0;
}
```

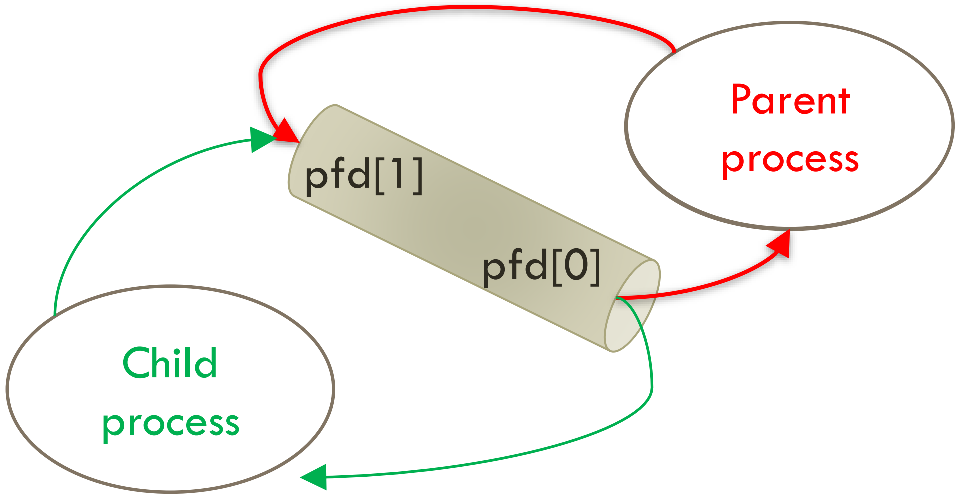
PIPE PROPERTIES

If a process attempts to **read from an empty pipe**, then **read() will be blocked** until data is available

If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read from the pipe will see the **end-of-file**

- **i.e., read() would return 0**

TRY! ./lab2-pipeerr



LAB2-PIPEERR.C

```
int main() {
  int pfd[2];

  pipe(pfd);

  if (fork() == 0) {  //in child
    close(pfd[1]);  //close the write end
    char buff[100];
    //printf("Child: Read from pipe ...\n");
    int res = read(pfd[0], buff, 100);
    if (res == 0)
        printf("Child: EOF is reached -- No one holds the write end!!\n");
  } else {  //in parent
    close(pfd[1]);  //close the write end
    printf("Press enter to close the program\n");
    getchar();
  }

  return 0;
}
```

HAVE SOME PRACTICE AT HOME

Test 1

- Comment out the close(pfd[1]); statement in the child block, compile and test run the program
- Observe and explain the behavior of the program

Test 2

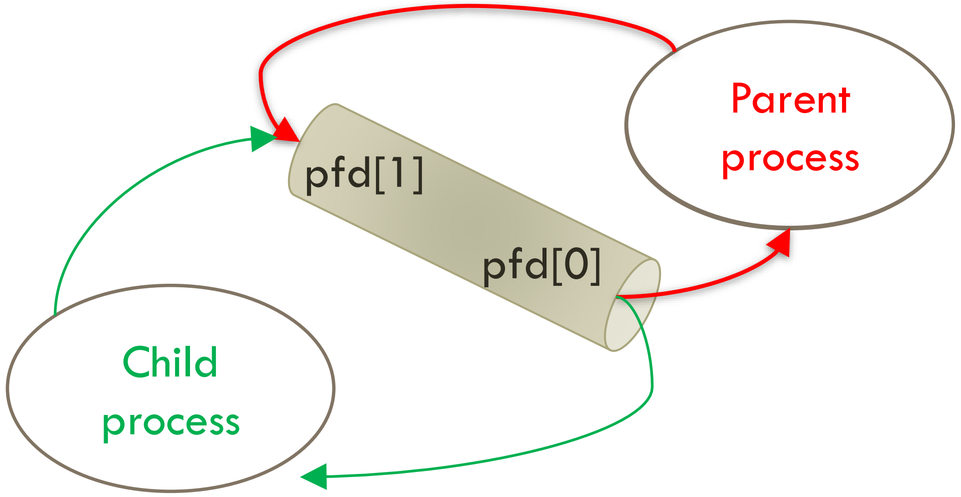
- Comment out the close(pfd[1]); statement in the parent block, compile and test run the program
- Observe and explain the behavior of the program

PIPE PROPERTIES

If all file descriptors referring to the read end of a pipe have been closed, then a write will cause a SIGPIPE signal to be generated for the calling process

- i.e., write() would also fails and return error

TRY! ./lab2-pipesig



LAB2-PIPESIG.C

```c
int main() {
  int pfd[2];

  pipe(pfd);

  if (fork() == 0) {    //in child
    sleep(2); //force child to run later
    close(pfd[0]);  //close the read end
    char buff[100];
    if (write(pfd[1], buff, 100) == -1)
      printf("Child: Encountered write error\n");
  } else {  //in parent
    close(pfd[0]);  //close the read end
    printf("Press enter to close the program\n");
    getchar();
/*  int status;
    wait(&status);
    if (WIFSIGNALED(status))
      printf("Child process was terminated by signal %s\n", strsignal(WTERMSIG(status)));
*/
  }

  return 0;
}
```

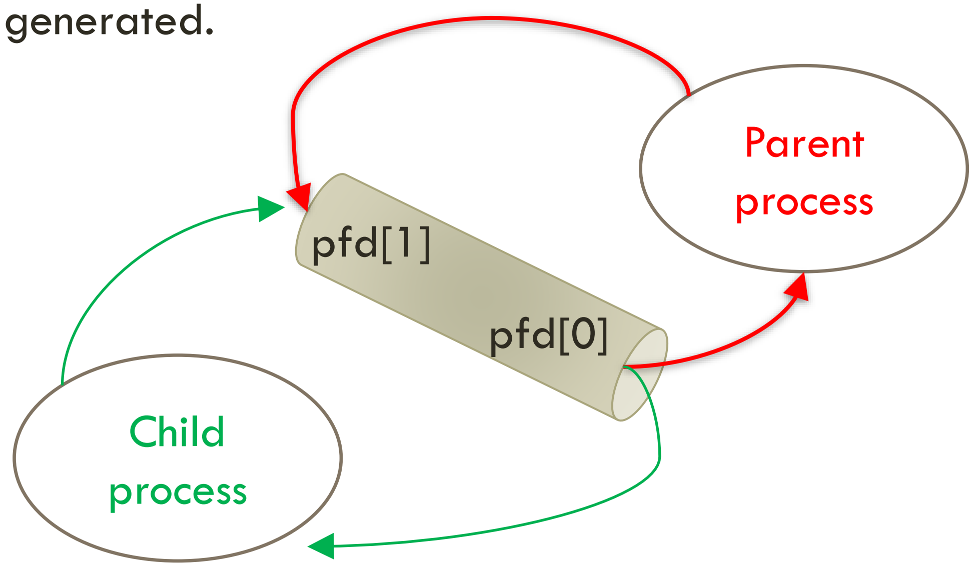
HAVE SOME PRACTICE AT HOME

Modify lab2-pipesig.c by uncomment the lines from line# 26 to 30, compile and test run the program

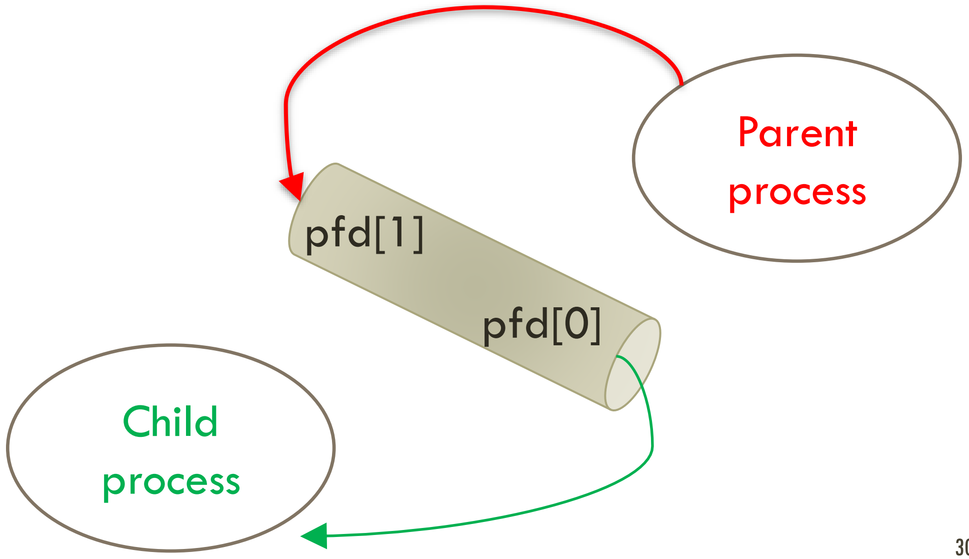
PIPE PROPERTIES

Situation1: Child is the reader and parent is the writer; however, after parent left, the child does not know and keep on waiting as read() cannot detect end-of-file.

Situation2: Child is the reader and parent is the writer; after the child left, the parent does not know and keep on writing until pipe becomes full as no SIGPIPE signal is generated.

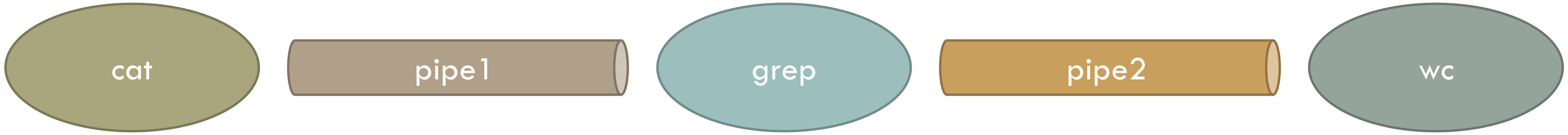


You should call close() to close unnecessary file descriptors; this ensures that end-of-file and SIGPIPE/EPIPE are delivered to correctly reflect the situation.



USING PIPES

cat demo-txt2.txt | grep process | wc -l



When user typed in these commands with pipes in the shell, it creates three processes and two pipes

Note: These three programs were developed independently. They don't know there are pipes used in the communication.

The Cruz: How can these three programs know that they have to output or get input from the pipes?

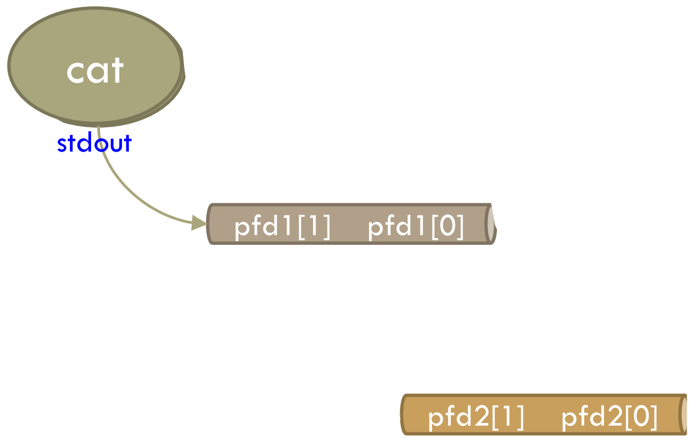
HERE IS THE MAGIC

Shell first creates two pipes

```
pipe(pfd1);
pipe(pfd2);
```

Then creates 1st child

The child then closes all pfds except pfd1[1] and associates its stdout to pdf[1] before switching to run the 1st command



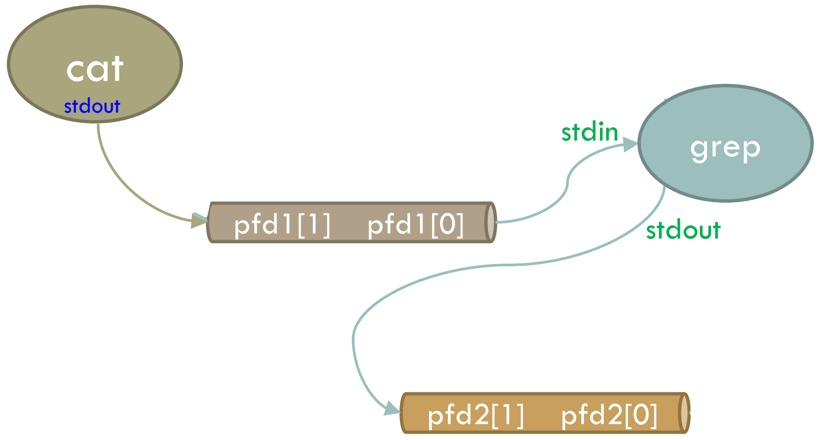
```
if (fork() == 0) { //first child
  close(pfd2[0]); //close pipe2
  close(pfd2[1]);
  close(pfd1[0]); //close pipe1 read end
  dup2(pfd1[1], 1);  //set pipe1 write end to stdout
  if (execvp(cmd1[0], cmd1) == -1) {
    printf("execvp: error no = %s\n", strerror(errno));
    exit(-1);
  }
}
```

HERE IS THE MAGIC

Parent creates 2nd child; this child can also access pdf1 & pdf2.

However, it closes pfd1[1] and pfd2[0] and associates pfd1[0] to its stdin & pdf2[1] to its stdout before switching to run the 2nd command

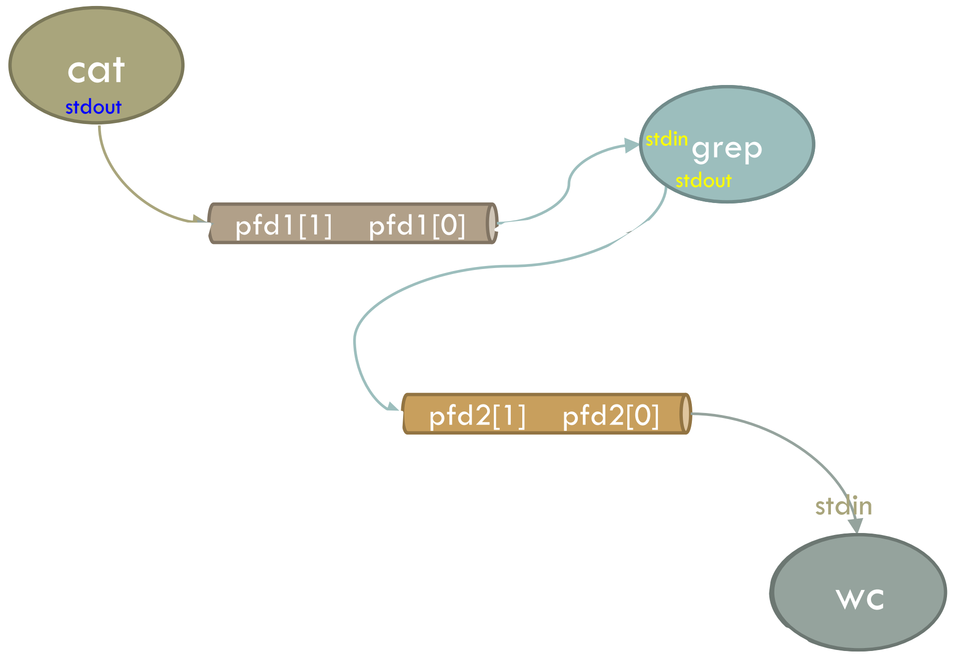
```
if (fork() == 0) { //second child
  close(pfd1[1]); //close pipe1 write end
  close(pfd2[0]); //close pipe2 read end
  dup2(pfd1[0], 0);  //set pipe1 read end to stdin
  dup2(pfd2[1], 1);  //set pipe2 write end to stdout
  if (execvp(cmd2[0], cmd2) == -1) {
    printf("execvp: error no = %s\n", strerror(errno));
    exit(-1);
  }
}
```



HERE IS THE MAGIC

Finally, creates 3rd child, and this child closes all pfds except pfd2[0] and associates it stdin to pfd2[0] before switching to run the 3rd command

```
if (fork() == 0) { //third child
  close(pfd1[0]); //close pipe1
  close(pfd1[1]);
  close(pfd2[1]); //close pipe2 write end
  dup2(pfd2[0], 0);  //set pipe2 read end to stdin
  if (execvp(cmd3[0], cmd3) == -1) {
    printf("execvp: error no = %s\n", strerror(errno));
    exit(-1);
  }
}
```



dup2( )

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

The dup2() system call creates a copy of the file descriptor ***oldfd*** and associate it to the file descriptor specified in ***newfd***.

- The system will first close the original file associated with newfd and set up the new association
- If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed
- If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd

Return

- On success, the new file descriptor newfd.
- On error, -1 is returned

BACKUP SLIDES

KILL( ) - SEND SIGNAL TO ANOTHER PROCESS

```
int kill(pid_t pid, int sig);
```

Send the signal *sig* to the process with PID *pid*

- e.g., kill(10234, SIGINT);
  - Send the signal SIGINT to process 10234

In command prompt, signal can be sent using kill command

- e.g.,

```
    [tmchan@localhost ~] kill -s SIGINT 12366
    [tmchan@localhost ~] kill -2 12366
```

SIGNAL RELATED FUNCTIONS

alarm( )

- unsigned int alarm(unsigned int seconds)
- Usage : send a SIGALRM signal after appointed time

raise( )

- int raise(int sig)
- Usage: created an signal with the signal number appointed by sig

pause( )

- int pause(void)
- Usage: suspend the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process

sigsuspend( )

- int sigsuspend(const sigset_t *mask)
- Usage: replace the signal with the given mask, and then suspend the process until delivery of a signal whose action is to invoke a signal handler or to terminate a process