UNIX Process Control

Chapter 8

Process Identifiers

- Every process has a unique process ID
  - Non-negative integer
- Special processes
  - 0: the swapper (long-term scheduler process - kernel)
  - 1: init process (runs as superuser)

Process Identifiers

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);  // PID of the caller
pid_t getppid(void); // PID of parent
pid_t getuid(void);  // real user-ID of process
pid_t geteuid(void); // effective user-ID of process
pid_t getgid(void);  // real group-ID of process
pid_t getegid(void); // effective group-ID of process
```

- These system calls return the various identifiers associated with the process

Process Creation

- All processes (except the first process created when the system is booted) are created by another process (**parent process**)
  - They are said to be **children** of the process that created them
- UNIX creates processes through the **fork()** system call (called forking a process)
- When a process forks
  - OS creates an **identical copy** of the forking process with a **new address space** and a **new PCB**
  - the **only** resources shared by the parent and the child processes are the opened files

fork() System Call

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

- Creates a new process
- This “function” is called once, but returns twice
- fork() returns a PID as follows:
  - Parent process gets the pid of the child process
  - Child process gets 0
- Both child and parent continue execution at the statement following the call to fork()

Process Creation Example

```
int pid; // to store child's pid
 ...
pid = fork();
if (pid == 0) {
   // code executed by child
     ...
   _exit(1); //terminates child
}
// code executed by parent
...
```

Process Creation

- One often finds:

```
if ((pid = fork()) == 0) { ... }
```

instead of:

```
pid = fork();
if (pid == 0) { ... }
```

File Sharing

- All file descriptors to open files in the parent are duplicated in the child process
- Thus, both the parent and the child share all files open at time of fork()
- If both parent and child write to same descriptor without synchronization, the outputs will be intermixed

Inherited Properties

- Properties inherited by child
  - Real user-ID, real group-ID
  - Effective user-ID, and effective group-ID
  - Supplementary group-IDs
  - Session ID
  - Process group ID
  - Controlling terminal
  - Set-user-ID flag and set-group-ID flag
  - Current working directory
  - Root directory
  - File mode creation mask (umask)
  - Signal mask and dispositions
  - Close-on-exec flag for any open file descriptors
  - Environment
  - Attached shared memory segments
  - Resource limits

fork() Failure

- Too many processes in the system
- Total number of processes for the real user-ID exceeds system's limit

Example of fork()

```
#include    <sys/types.h>
#include    "ourhdr.h"

int glob = 6; /*external variable initialized data */
char  buf[] = "a write to stdout\n";

int main(void) {
   int var;  /* automatic variable on the stack */
   pid_t     pid;

   var = 88;
   if (write(STDOUT_FILENO, buf, sizeof(buf)-1) !=
             sizeof(buf)-1)
      err_sys("write error");
   printf("before fork\n"); /*we don't flush stdout */
```

Example of fork()

```
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {        /* child */
        glob++;                 /* modify variables */
        var++;
    } else
        sleep(2);               /* parent */

    printf("pid = %d, glob = %d, var = %d\n",
    getpid(), glob, var); /*both processes*/
    exit(0);
}
```

Example of fork()

- If we run the example, we get:

```
> a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89
pid = 429, glob = 6, var = 88
```

childs variables were changed
parents copies not changed

```
> a.out > temp.out
cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

2nd before fork due to buffers not being flushed prior to fork

vfork() System Call

```
#include <sys/types.h>
#include <unistd.h>

pid_t vfork(void);
```

- Works like fork(), except:
  - Both parent and child share the same address space
  - Both run in the address space of the parent
- New address space created when the exec() system call is made

Example of vfork()

```
#include    <sys/types.h>
#include    "ourhdr.h"

int glob = 6; /*external variable initialized data */
char  buf[] = "a write to stdout\n";

int main(void) {
   int var;  /* automatic variable on the stack */
   pid_t     pid;

   var = 88;
   if (write(STDOUT_FILENO, buf, sizeof(buf)-1) !=
             sizeof(buf)-1)
       err_sys("write error");
   printf("before fork\n"); /*we don't flush stdout */
```

Example of vfork()

```
if ( (pid = vfork()) < 0)
    err_sys("fork error");
else if (pid == 0) {            /* child */
    glob++;                     /* modify variables */
    var++;
    _exit(0);                   /* child terminates */
}

/* parent */
sleep(4);
printf("pid = %d, glob = %d, var = %d\n",
getpid(), glob, var);
exit(0);
}
```

Example of vfork()

- If we run the example, we get:

```
>a.out
a write to stdout
before fork
pid = 607, glob = 7, var = 89
```
parents variables were changed

Race Conditions

- A race condition occurs when two or more processes are trying to do something with shared data, and the final outcome depends on the order in which the processes run

Waiting for Child Process Termination

- A parent process can **wait** for the termination of one of its children by doing:

  pid=wait(0);

  which returns the process ID of the child whose termination was caught.
- To wait for the completion of a **specific** child, say the one with process ID **this_child**, use:

```
while (wait(0) != this_child) /* empty */;
```

Executing a New Program

- To load a different program in the child's address space:

```
execve(full_pathname, arg_vector, envp);
```

  - where **full_pathname** is the pathname of the executable to be fetched
  - **arg_vector** is an array of pointers to the individual argument strings
    - **arg_vector[0]** contains the name of the program as it appears in the command line and the end of the array is indicated by a **NULL** pointer
  - **envp** is an array of pointers pointing to the environment strings: it is also terminated by a **NULL**

Process Deletion

- If the deletion is initiated by the process itself:
  - Use the **exit(code)** library call, or
  - Use the **_exit(code)** system call
- If it is initiated by another process, the other process can send a **signal** using the **kill()** system call.
  - The process receiving the signal can catch it by using the **signal()** system call;
    - the process catching the signal will not terminate.
- Two signals cannot be caught:
  - the 9^{th} signal, **SIGKILL**, and
  - the 23^{rd} signal, **SIGTSTOP**

wait() & waitpid() System Calls

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

- *statloc* is a pointer to an integer used to store the termination status of the process handled by the wait
  - Macros can be used to determine this status
- *pid* specifies the process ID number to wait for
- *options* allows the caller to have further control over how the wait occurs

wait() & waitpid() System Calls

- A process executing wait() or waitpid() can:
  - Block
    - If all of its children are still running
  - Return immediately with the termination status of a child
    - If a child has terminated and is waiting for its termination status to be fetched
  - Return immediately with an error
    - If it does not have any child processes

Macros for wait

WIFEXITED(status)	True if status was returned for a child that terminated normally.
WEXITSTATUS(status)	Fetches the low order 8 bits of the argument to exit() or _exit()

Macros for wait

`WIFSIGNALED(status)`	True if status was returned for a child that terminated abnormally (by an uncaught signal)
`WTERMSIG(status)`	Fetches the signal number which caused termination
`WCOREDUMP(status)`	True if a core file was generated for the terminated process

Macros for wait

`WIFSTOPPED(status)`	True if status was returned for a child that is currently stopped
`WSTOPSIG(status)`	Fetches the signal number that cased the child to stop

Pid values for waitpid()

- The interpretation of pid depends upon its value:
  - `-1`: waitpid() waits for any process [same as wait()]
  - `>0`: waits for child with specified pid #
  - `0`: waits for any child whose process group ID number equals that of the calling process
  - `<-1`: waits for any child whose process group ID number equals the absolute value of the specified pid #

wait() & waitpid() System Calls

- wait() always blocks until a child terminates
- waitpid() has an option to allow it to continue
- waitpid() can also check on the status of stopped child processes

Options for waitpid()

WNOHANG	waitpid() will not block if the child specified by **pid** is not immediately available (i.e., already terminated)
WUNTRACED	Causes waitpid() to return the status for any *stopped* child specified by **pid** which has not had its status reported since it was stopped

Examples using waitpid()

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int main(void) {
  pid_t    pid;
  int status;

  if ( (pid = fork()) < 0) err_sys("fork error");
  else if (pid == 0)    /* child */
      exit(7);

  if (wait(&status) != pid)  err_sys("wait error");
           /* wait for child */
  pr_exit(status);       /* and print its status */
```

Examples using waitpid()

```
if ( (pid = fork()) < 0)  err_sys("fork error");
else if (pid == 0)    /* child */
    abort();    /* generates SIGABRT */

if (wait(&status) != pid)  err_sys("wait error");
          /* wait for child */
pr_exit(status);       /* and print its status */

if ( (pid = fork()) < 0)  err_sys("fork error");
else if (pid == 0)    /* child */
    status /= 0;  /*divide by 0 generates SIGFPE*/

if (wait(&status) != pid)  err_sys("wait error");
          /* wait for child */
pr_exit(status);       /* and print its status */

exit(0);
}
```

wait3 and wait4

- BSD varaints of wait and waitpid

```
pid_t wait3(int *statloc, int options,
  struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int
  options, struct rusage *rusage);
```

These two are similar but allow kernel to return information about resource usage by the terminated process

- See man pages for more on the struct rusage type

exec System Calls

- When a process makes an *exec* call, the process is completely replaced by the new program, which starts execution of its main program
- However, the process ID does not change

exec System Calls

- Six forms of *exec*:

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h“

int execl(const char * pathname, const char * arg0, …,
   (char *) 0);

int execv(const char *pathname, char * const argv[]);

int execle(const char *pathname, const char * arg0, …,
   (char *) 0, char * const envp[]);

int execve(const char *pathname, char * const argv[],
   char *const envp[]);

int execlp(const char *filename, const char * arg0, …,
   (char *) 0);

int execvp(const char *filename, char * const argv[]);
```

exec System Calls

- The first four take a pathname argument
  - Assumed to be an executable program
- The last two take a filename argument
  - This filename can itself be a pathname
  - The filename is either:
    - An executable generated by the link editor
      - Load the program and execute it
    - A shell script
      - Load a shell and have the shell execute the script

exec System Calls

- Argument Passing
  - Two categories of exec calls :
    - The execl* set of calls (l stands for list):
      - The execl(), execlp(), and execle() calls
      - The command line arguments to be passed to the new program must be specified as separate arguments to the call to *exec*
      - This looks like:

```
char *arg0, char *arg1, …, char *argn,
(char *) 0
```

exec System Calls

- Argument Passing
  - Two categories of exec calls :
    - The execv* set of calls (v stands for vector):
      - The execv(), execvp(), and execve() calls
      - The command line arguments to be passed to the new program must be specified in an array of C style strings (char *)
        - Similar to command line argument processing in the call to main( )

exec System Calls

- Environment List Passing
  - Two categories of exec calls :
    - The set of calls which use the environment variable:
      - The execl(), execlp(), execv(), and execvp() calls
      - The environment strings to be passed to the new program are obtained from the environ variable in the calling process
      - Allows the environment to be inherited from the parent

exec System Calls

- Environment List Passing
  - Two categories of exec calls :
    - Those which use an array of pointers:
      - The execle() and execve() calls
      - The environment strings to be passed to the new program must be specified in an array of C style strings (char *)
        - The **envp[]** array
        - Also similar to command line argument processing in the call to main( )
      - Allows the parent to specify a specific environment for the child

Example of execle()

```
#include     <sys/types.h>
#include     <sys/wait.h>
#include     "ourhdr.h"

char *env_init[] = {"USER=unknown","PATH=/tmp",NULL};

int main(void) {
  pid_t     pid;

  if ( (pid = fork()) < 0)
      err_sys("fork error");
  else if (pid == 0) {  /* specify pathname, specify
                                environment */
      if (execle("/home/stevens/bin/echoall",
                      "echoall", "myarg1", "MY ARG2",
                            (char *) 0, env_init) < 0)
            err_sys("execle error");
  }
```

Example of execle()

```
    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {  /* specify filename, inherit
                             environment */
        if (execlp("echoall", "echoall", "only 1 arg",
               (char *) 0) < 0)
               err_sys("execlp error");
    }
    exit(0);
}
```

The echoall program

```
#include      "ourhdr.h"

int main(int argc, char *argv[], char *environ[]) {
int main(int argc, char *argv[]){
   int                 i;
   char                **ptr;
   extern char         **environ;

   for (i = 0; i < argc; i++) /*echo all command-line args*/
       printf("argv[%d]: %s\n", i, argv[i]);

   for (ptr = environ; *ptr != 0; ptr++) /* and all env  */
                                          /*     strings */
       printf("%s\n", *ptr);

   exit(0);
}
```

Example Output

```
$ a.out myarg1 “MY ARG2”
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unkown
PATH=/tmp
argv[0]: echoall
$ arv[1]: only 1 arg
USER=stevens
HOME=/home/stevens
LOGNAME=stevens
    ⋮
EDITOR=/usr/ucb/vi
```

the parent doesn’t wait for the child to finish

Summary of Process Control Primitives

- The ***fork*** calls create new processes
- The ***exec*** calls initiate new programs
- The ***exit*** calls and the ***wait*** calls handle process termination

setuid() & setgid() System Calls

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

- Changes user-IDs as follows:
  - Superuser:
    - setuid changes the real user-ID, effective user-ID, and saved set-user-ID to the given *uid*
  - Others:
    - If uid equals either real user-ID or effective user-ID, or the saved set-user-ID, then change the effective user-ID to the uid specified (real user-ID and saved set-user-ID not changed)
    - Otherwise: error

UNIX Pipes

- In a UNIX shell, the pipe symbol is: | (the vertical bar)
- In a shell, UNIX pipes look like:

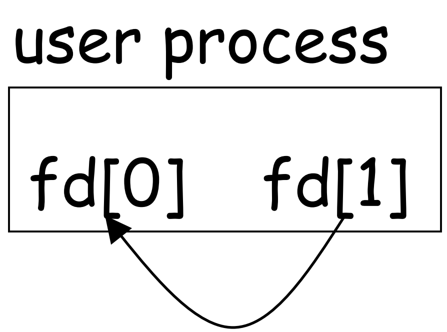
  ls -alg | more

  where the standard output of the program at the left (i.e., the *producer*) becomes the standard input of the program at the right (i.e., the *consumer*).
- We can have longer pipes:

  pic paper.ms | tbl | eqn | ditroff -ms
- Pipes have one *major limitation*:
  - processes **cannot pass** pipes and must *inherit them* from their parent
  - if a process creates a pipe, all its children will inherit it

Pipes

- Pipes are established using the **`pipe()`** system call
  - Establishes two "connected" streams
  - The streams are half duplex
    - Data flows in one direction
  - Data written to "write" end of the stream can be read from the "read" end of the stream



Pipes

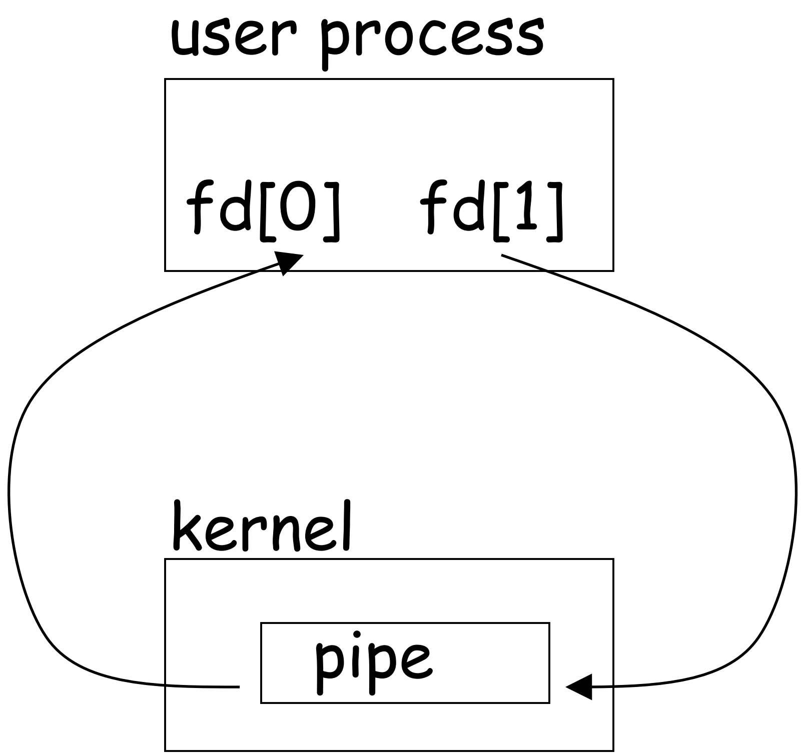
```
#include <unistd.h>

int pipe(int filedes[2]);
```

- The two slots of the array are file descriptors referencing the two ends of a stream which can cross process address space boundaries

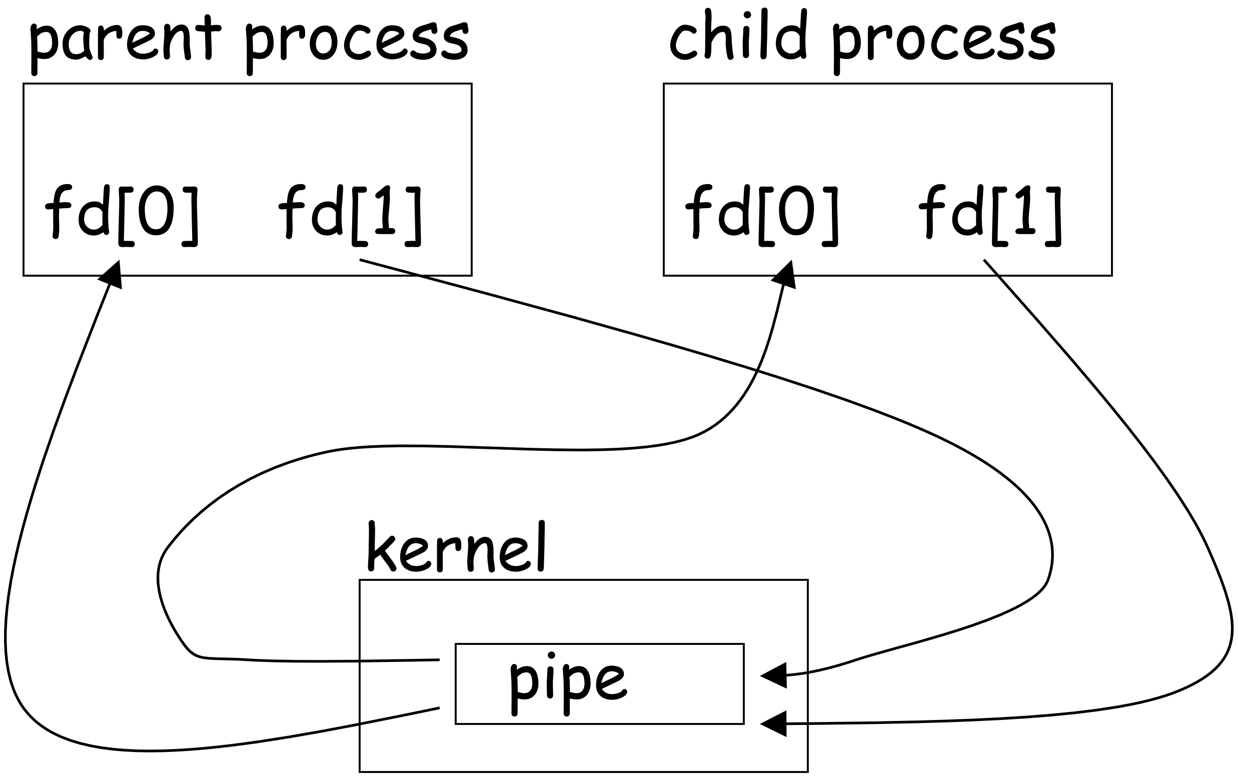
Pipes

- Result after the call to **`pipe( )`**



Pipes

- Result after a fork



Pipes

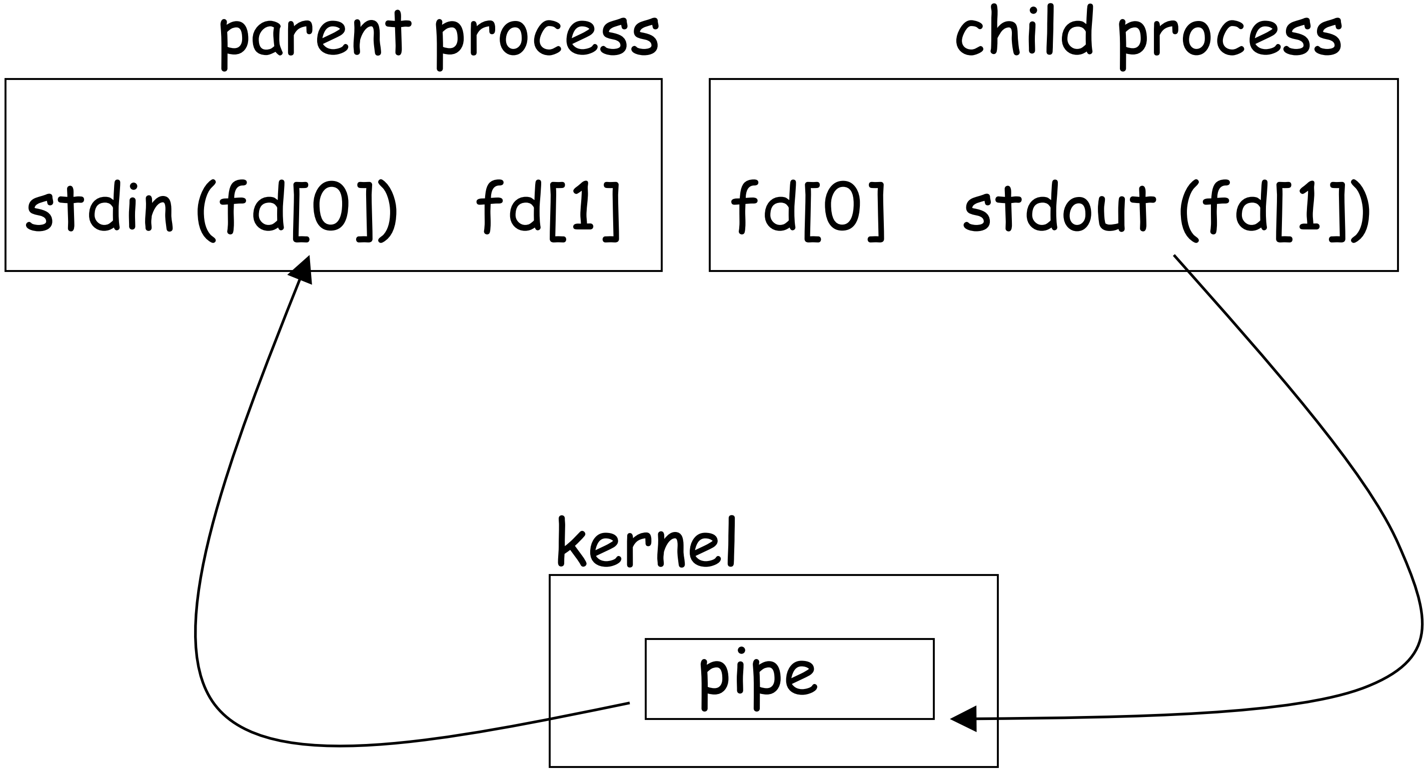
- Example

```
int filedes[2];
pipe(filedes);

if ((pid=fork())==0) {
    close(1);
    dup(filedes[1]);
    close(filedes[0]);
    close(filedes[1]);
    exec(......);
}
else {
    close(0);
    dup(filedes[0]);
    close(filedes[0]);
    close(filedes[1]);
    exec(......);
}
```

Pipes

- Final result



I/O Redirection

- Note that **dup()** can also be used to redirect the standard output of a process to a file:

```
// open target file and create it if needed
fd = open("log", O_WRONLY | O_CREAT, 0644);
close(1); // close stdout/
dup(fd); // dup into stdout
close(fd); // good practice
```

- or read its standard input from another file:

```
fd = open("data", O_RDONLY);
close(0); // close stdin/
dup(fd); // dup into stdin
close(fd); // good practice
```

I/O Redirection

```
char * filename1, filename2;
filename1 = get_name();
filename2 = get_name();
fd0=open (filename1, O_RDONLY);
fd1=open(filename2, O_WRONLY);

if ((pid=fork())==0) {  // child with I/O Redirection
    close(1);
    dup(fd1);     // redirect STDOUT to filename2
    close(fd1);
    close(0);
    dup(fd0);     // redirect STDIN from filename1
    close(fd0);
    exec(......);
}
else {
    // parent does this
}
```