Processes and Basic IPC [1]

Alex Delis
alex.delis -at+ nyu.edu

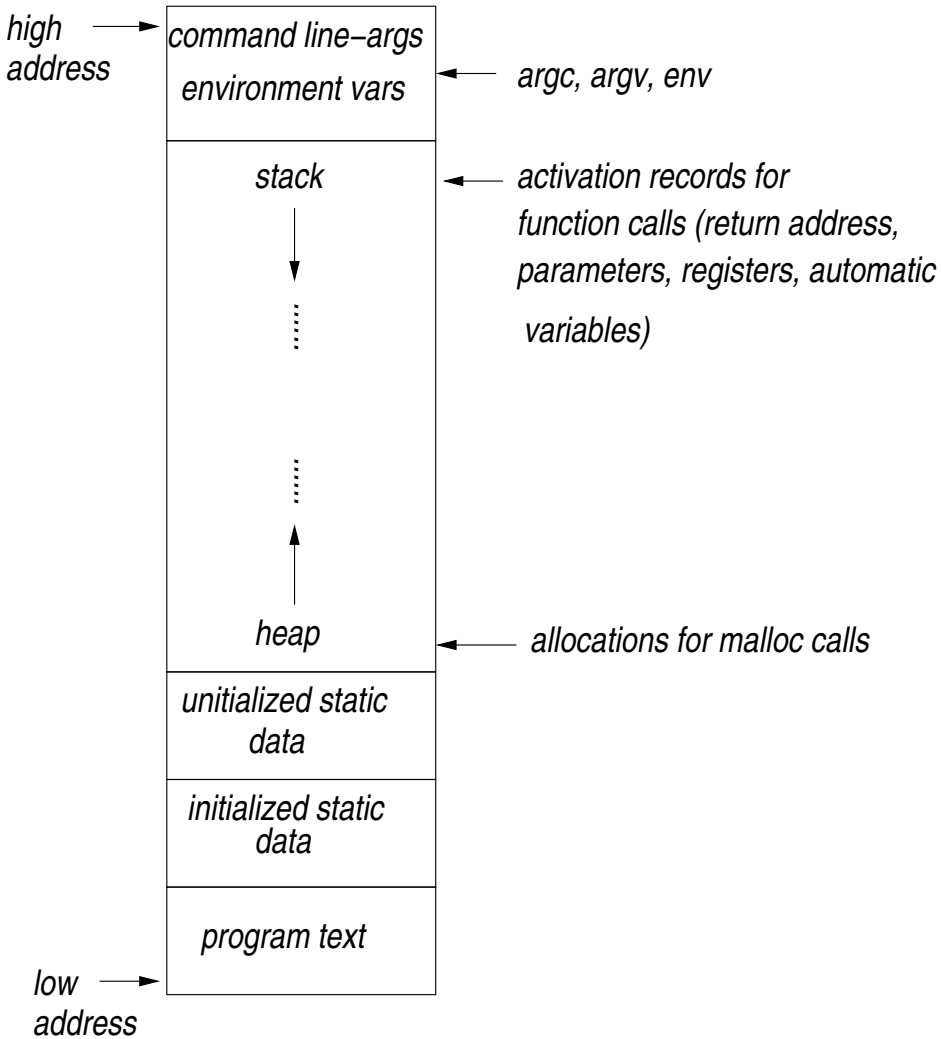
September 2017

[1] Acknowledgements to Prof. T. Stamatopoulos, M. Avidor, Prof. A. Deligiannakis, S. Evangelatos, Dr. V. Kanitkar and Dr. K. Tsakalozos.

Processes in Unix

- Each Unix process has its own identifier (PID), its code (text), data, stack and a few other features (that enable it to "import" argc, argv, env variable, memory maps, etc).
- The *very first* process is called *init* and has PID=1.
- The **only way** to create a process is that another process *clones itself*. The new process has a child-to-parent relationship with the original process.
- The id of the child is different from the id of the parent.
- All processes in the system are descedants of *init*.
- A child process can eventually *replace* its own code (text-data), its data and its stack with those of another executable file. In this manner, the child process may differentiate itself from its parent.

Process Instance



Process IDs

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

  pid_t getpid(void);
  pid_t getppid(void);
  ```

- *getpid()*: obtain my own ID,
  getppid(): get the ID of my parent.

- ```
  #include <stdio.h>
  #include <unistd.h>

  int main(){
     printf("Process has as ID the number: %ld \n",(long)getpid());
     printf("Parent of the Process has as ID: %ld \n",(long)getppid());
     return 0;
  }
  ```

- Running the program...

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
Process has as ID the number: 14617
Parent of the Process has as ID: 3256
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

The *exit()* call

- ```
  #include <stdlib.h>

  void exit(int status);
  ```
- Terminates the running of a process and returns a *status* which is available in the parent process.
- When *status* is 0, it shows successful exit; otherwise, the value of *status* is available (often) at the shell variable $?

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

#define EXITCODE 157

main(){
   printf("Going to terminate with status code 157 \n");
   exit(EXITCODE);
}
```

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
Going to terminate with status code 157
ad@ad-desktop:~/SysProMaterial/Set005/src$ echo $?
157
ad@ad-desktop:~/SysProMaterial/Set005/src$
```
```

Creating a new process – *fork()*

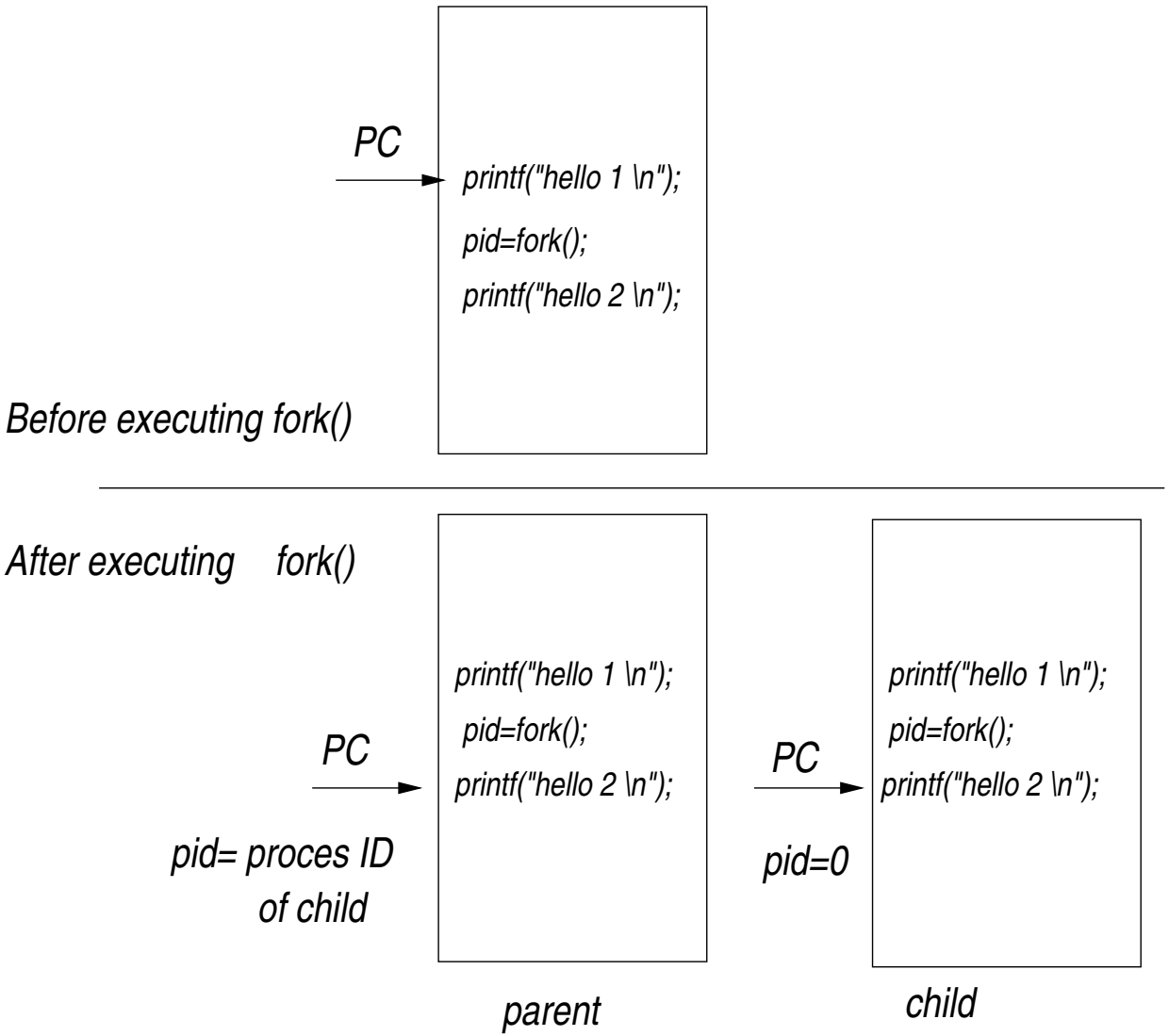
- The system call:

```
#include <unistd.h>

pid_t fork(void);
```

- creates a new process by duplicating the calling process.
- *fork()* returns the value 0 in the child-process, while returns the processID of the child process to the parent.
- *fork()* returns -1 in the parent process if it is not feasible to create a new child-process.

Where the PCs are after *fork()*



fork() example

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main(){
  pid_t  childpid;

  childpid = fork();
  if (childpid == -1){
     perror("Failed to fork");
     exit(1);
     }
  if (childpid == 0)
     printf("I am the child process with ID: %lu \n",
        (long)getpid());
  else
     printf("I am the parent process with ID: %lu \n",
        (long)getpid());
  return 0;
}
```

```
d@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
I am the parent process with ID: 15373
I am the child process with ID: 15374
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
I am the parent process with ID: 15375
I am the child process with ID: 15376
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

Another example

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

int main(){
  pid_t  childpid;
  pid_t  mypid;

  mypid = getpid();
  childpid = fork();
  if (childpid == -1){
     perror("Failed to fork");
     exit(1);
     }
  if (childpid == 0)
     printf("I am the child process with ID: %lu -- %lu\n",
             (long)getpid(), (long)mypid);
  else { sleep(2);
         printf("I am the parent process with ID: %lu -- %lu\n",
             (long)getpid(), (long)mypid); }
  return 0;
}
```

→ Running the executable:

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
I am the child process with ID: 15704 -- 15703
I am the parent process with ID: 15703 -- 15703
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

Creating a chain of processes

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

int main(int argc, char *argv[]){
pid_t childpid = 0;
int i,n;

if (argc!=2){
  fprintf(stderr,"Usage: %s processes \n",argv[0]); return 1;
}

n=atoi(argv[1]);
for(i=1;i<n;i++)
  if ( (childpid = fork()) > 0 )  // only the child carries on
    break;

fprintf(stderr,"i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
    i,(long)getpid(),(long)getppid(),(long)childpid );
return 0;
}
```

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out 2
i:1 process ID:7654 parent ID:3420 child ID:7655
i:2 process ID:7655 parent ID:7654 child ID:0
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out 4
i:1 process ID:7656 parent ID:3420 child ID:7657
i:3 process ID:7658 parent ID:7657 child ID:7659
i:4 process ID:7659 parent ID:7658 child ID:0
i:2 process ID:7657 parent ID:1 child ID:7658
```

The *wait()* call

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

  pid_t wait(int *status);
  ```
- Waits for state changes in a child of the calling process, and obtains information about the child whose state has changed.
- Returns the ID of the child that terminated, or -1 if the calling process had no children.
- Good idea for the parent to wait for *every* child it has spawnned.
- If *status* information is not NULL, it stores information that can be inspected.
  1. *status* has two bytes: in the left we have the exit code of the child and in the right byte just 0.
  2. if the child was terminated due to a signal, then the last 7 bits of the *status* represent the code for this signal.
  ```

Use of *wait*

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(){
  pid_t  pid;
  int  status, exit_status;

  if ( (pid = fork()) < 0 ) perror("fork failed");

  if (pid==0){ sleep(4); exit(5); /* exit with non-zero value */ }
  else { printf("Hello I am in parent process %d with child %d\n",
        getpid(), pid); }

  if ((pid= wait(&status)) == -1 ){
     perror("wait failed"); exit(2);
     }
  if ( (exit_status = WIFEXITED(status)) ) {
     printf("exit status from %d was %d\n",pid, exit_status);
     }
exit(0);
}
```

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
Hello I am in parent process 17022 with child 17023
exit status from 17023 was 1
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

The *waitpid* call

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

pid_t waitpid(pid_t pid, int *status, int options);
```

- *pid* may take various values:
  1. $<$ -1: wait for any child whose groupID $=$ |pid|
  2. -1: wait for any child
  3. 0: wait for any child process whose process groupID is equal to that of the calling process.
  4. $>$ 0 : wait for the child whose process ID is equal to the value of pid.
- *options* is an OR of zero or more of the following constants:
  1. WNOHANG: return immediately if no child has exited.
  2. WUNTRACED: return if a child has stopped.
  3. WCONTINUED: return if a stopped child has been resumed (by delivery of SIGCONT).

getpid() example

```c
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main(){
   pid_t  pid;
   int status, exit_status,i ;

   if ( (pid = fork()) < 0 )
      perror("fork failed");
   if ( pid == 0 ){
    printf("Still child %lu is sleeping... \n",(long)getpid());
        sleep(5); exit(57);
        }
   printf("reaching the father %lu process \n",(long)getpid());
   printf("PID is %lu \n", (long)pid);
   while( (waitpid(pid, &status, WNOHANG)) == 0 ){
        printf("Still waiting for child to return\n");
        sleep(1);
    }
   printf("reaching the father %lu process \n",(long)getpid());
   if (WIFEXITED(status)){
       exit_status = WEXITSTATUS(status);
       printf("Exit status from %lu was %d\n", (long)pid, exit_status);
       }
   exit(0);
}
```

Example with *waitpid()*

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
reaching the father 17321 process
PID is 17322
Still waiting for child to return
Still child 17322 is sleeping...
Still waiting for child to return
Still waiting for child to return
Still waiting for child to return
Still waiting for child to return
reaching the father 17321 process
Exit status from 17322 was 57
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

Using *wait*

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

int main(){
  pid_t pid;
  int status;

  printf("Original Process: PID = %d\n",getpid());
  pid = fork();
  if (pid == -1 ) {
      perror("fork failed");
      exit(1);
      }

   if ( pid!=0 ) {
      printf("Parent process: PID = %d \n",getpid());
      if ( (wait(&status) != pid ) ) {
           perror("wait");
           exit(1);
           }
      printf("Child terminated: PID = %d, exit code = %d\n",pid, status >> 8);
      }
    else {
       printf("Child process: PID = %d, PPID = %d \n", getpid(), getppid());
       exit(62);
       }
    printf("Process with PID = %d terminates",getpid());
    sleep(1);
}
```

Running the Example with *waitpid()*

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
Original Process: PID = 17392
Parent process: PID = 17392
Child process: PID = 17393, PPID = 17392
Child terminated: PID = 17393, exit code = 62
Process with PID = 17392 terminatesad@ad-desktop:~/SysProMaterial/Set005/src$
```

The *execve()* call

- *execve* executes the program pointed by *filename*

```
#include <unistd.h>

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

- *argv*: is an array of argument strings passed to the new program.
- *envp*: is an array of strings the designated the "environment" variables seen by the new program.
- Both *argv* and *envp* must be NULL-terminated.
- *execve* does not return on success, and the text, data, bss (un-initialized data), and stack of the calling process are overwritten by that of the program loaded.
- On success, *execve()* does **not** return, on error -1 is returned, and *errno* is set appropriately.

Related system calls: *execl*, *execlp*, *execle*, *execv*, *execvp*

- ```
  #include <unistd.h>
  extern char **environ;
  int execl(const char *path, const char *arg, ...);
  int execlp(const char *file, const char *arg, ...);
  int execle(const char *path, const char *arg, ..., char * const envp[]);
  int execv(const char *path, char *const argv[]);
  int execvp(const char *file, char *const argv[]);
  ```

- These calls, collectively known as the *exec* calls, are a front-end to *execve*.

- They all replace the calling process (including text, data, bss, stack) with the executable designated by either the *path* or *file*.
  ```

Features of *exec* calls

- *execl*, *execle* and *execv* require either absolute or relative paths to executable(s).
- *execlp* and *execvp* use the environment variable PATH to "locate" the executable to replace the invokign process with.
- *execv* and *execvp* require the name of the executable and its arguments in *argv[0], argv[1], argv[2],..,argv[n]* and NULL as delimeter in *argv[n+1]*.
- *execl*, *execlp* and *execle* require the names of executable and parameters in *arg0, arg1, arg2, .., argn* with NULL following.
- *execle* requires the the passing of environment variables in *envp[0], envp[1], envp[2],..,envp[n]* and NULL as delimeter in *envp[n+1]*.

Using *execl()*

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

main(){
  int retval=0;

  printf("I am process %lu and I will execute an 'ls -l .; \n",(long)getpid);

  retval=execl("/bin/ls", "ls", "-l", ".", NULL);

  if (retval==-1)      // do we ever get here?
    perror("execl");
}
```

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
I am process 134513516 and I will execute an 'ls -l .;
total 64
-rwxr-xr-x 1 ad ad 8413 2010-04-19 23:56 a.out
-rw-r--r-- 1 ad ad  233 2010-04-19 23:56 exec-demo.c
-rwx------ 1 ad ad  402 2010-04-19 00:42 fork1.c
-rwx------ 1 ad ad  529 2010-04-19 00:59 fork2.c
-rwx------ 1 ad ad  669 2010-04-19 13:08 wait_use.c
-rwx------ 1 ad ad  273 2010-04-19 13:16 zombies.c
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

Example with *execvp()*

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

int main(void){
   int pid, status;
   char *buff[2];

   if ( (pid=fork()) == -1){ perror("fork"); exit(1); }
   if ( pid!=0 ) { // parent
      printf("I am the parent process %d\n",getpid());
      if (wait(&status) != pid){  //check if child returns
          perror("wait"); exit(1); }
      printf("Child terminated with exit code %d\n", status >> 8);
      }
   else {
      buff[0]=(char *)malloc(12); strcpy(buff[0],"date");
      printf("%s\n",buff[0]); buff[1]=NULL;

      printf("I am the child process %d ",getpid());
      printf("and will be replaced with 'date'\n");
      execvp("date",buff);
      exit(1);
      }
}
```

Running the program...

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
I am the parent process 3792
date
I am the child process 3793 and will be replaced with 'date'
Tue Apr 20 00:23:45 EEST 2010
Child terminated with exit code 0
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

Pipes

- Sharing files is a way for various processes to communicate among themselves (but this entails a number of problems).
- **pipes** are one means that Unix addresses one-way communication between two process (often parent and child).
- Simply stated: in a pipe, a process sends "down" the pipe data using a *write* and another (perhaps the same?) process receives data at the other end through with the help of a *read* call.

Pipes

- ```
  #include <unistd.h>

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

- *pipe* creates a unidirectional data channel that can be used for interprocess communication in which *pipefd[0]* refers to the **read end** of the pipe and *pipefd[1]* to the **write end**.

- A pipe's real value appears when it is used in conjunction with a *fork()* and the fact that the file descriptors remain open across a *fork()*.
  ```

Somewhat of a useless example...

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGSIZE 16

char *msg1="Buenos Dias! #1";
char *msg2="Buenos Dias! #2";
char *msg3="Buenos Dias! #3";

main(){
  char inbuf[MSGSIZE];
  int p[2], i=0, rsize=0;
  pid_t pid;

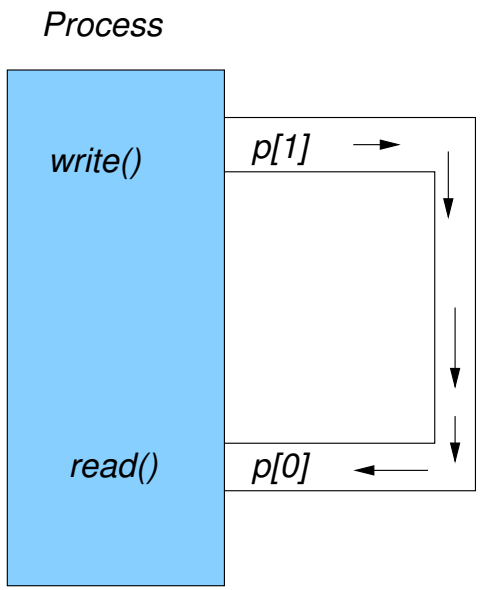
  if (pipe(p)==-1) { perror("pipe call"); exit(1);}

  write(p[1],msg1,MSGSIZE);
  write(p[1],msg2,MSGSIZE);
  write(p[1],msg3,MSGSIZE);

  for (i=0;i<3;i++){
        rsize=read(p[0],inbuf,MSGSIZE);
        printf("%.*s\n",rsize,inbuf);
        }
  exit(0);
}
```

Here is what happens...

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
Buenos Dias! #1
Buenos Dias! #2
Buenos Dias! #3
ad@ad-desktop:~/SysProMaterial/Set005/src$
```



The output and the input are part of the **same** process - not useful!

Here is a *somewhat* more useful example...

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGSIZE 16

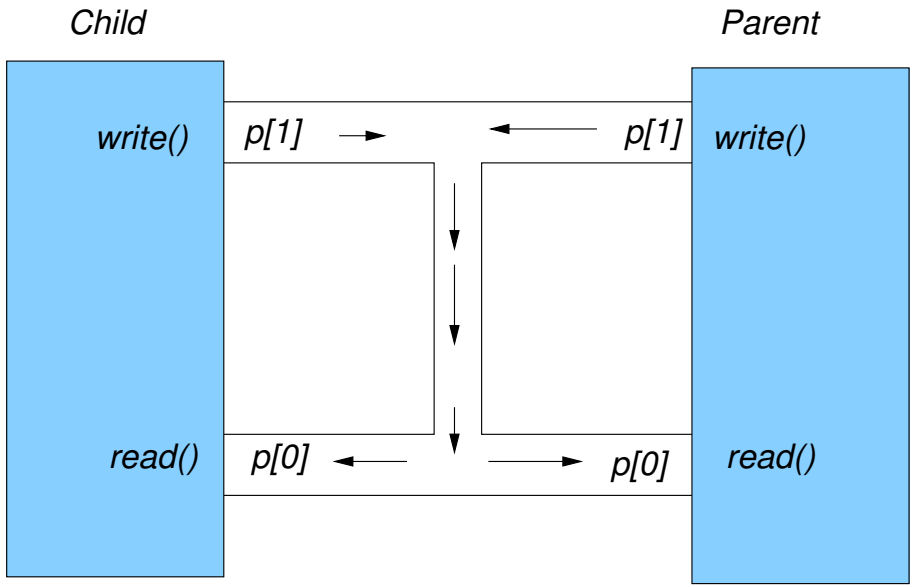
char *msg1="Buenos Dias! #1";
char *msg2="Buenos Dias! #2";
char *msg3="Buenos Dias! #3";

main(){
  char inbuf[MSGSIZE];
  int p[2], i=0, rsize=0;
  pid_t pid;

  if (pipe(p)==-1) { perror("pipe call"); exit(1);}

  switch(pid=fork()){
  case -1: perror("fork call"); exit(2);
  case  0: write(p[1],msg1,MSGSIZE);   // if child then write!
           write(p[1],msg2,MSGSIZE);
           write(p[1],msg3,MSGSIZE);
           break;
  default: for (i=0;i<3;i++){          // if parent then read!
                rsize=read(p[0],inbuf,MSGSIZE);
                printf("%.*s\n",rsize,inbuf);
                }
           wait(NULL);
  }
  exit(0);
}
```

Here is what happens now:



- Either process could write down the file descriptor *p[1]*.
- Either process could read from the file descriptor *p[0]*.
- Problem: pipes are intended to be unidirectional; if both processes start reading and writing indiscriminately, **chaos may ensue**.

A much cleaner version

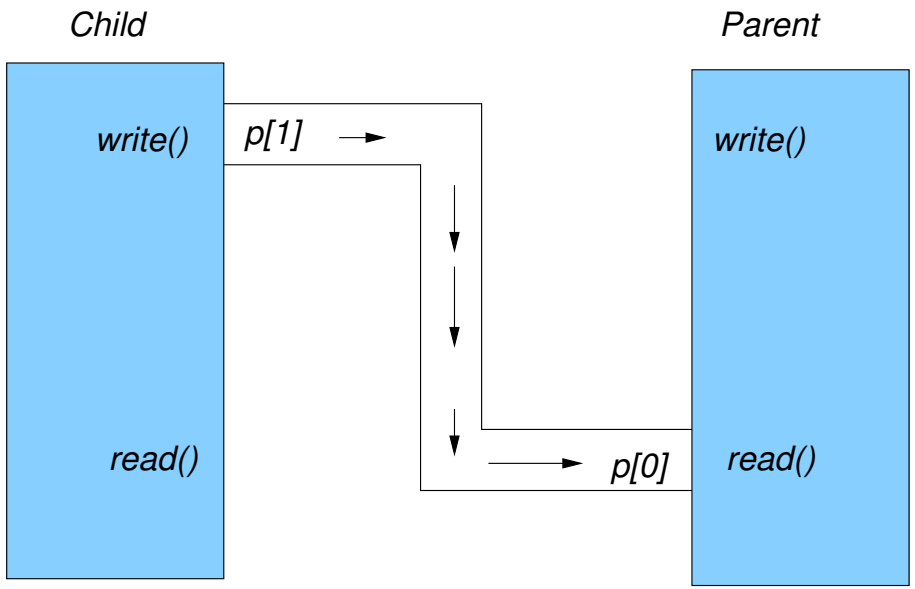
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGSIZE 16

char *msg1="Buenos Dias! #1";
char *msg2="Buenos Dias! #2";
char *msg3="Buenos Dias! #3";

main(){
  char inbuf[MSGSIZE];
  int p[2], i=0, rsize=0;
  pid_t pid;
  if (pipe(p)==-1) { perror("pipe call"); exit(1);}

  switch(pid=fork()){
  case -1: perror("fork call"); exit(2);
  case  0: close(p[0]);                  // child is writing
           write(p[1],msg1,MSGSIZE);
           write(p[1],msg2,MSGSIZE);
           write(p[1],msg3,MSGSIZE);
        break;
  default: close(p[1]);                  // parent is reading
           for (i=0;i<3;i++){
        rsize=read(p[0],inbuf,MSGSIZE);
        printf("%.*s\n",rsize,inbuf);
        }
        wait(NULL);
  }
  exit(0);
}
```

.. and pictorially:



Example using Pipe

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#define BUFSIZE 10

int main(void){
   char bufin[BUFSIZE]="empty";
   char bufout[BUFSIZE]="hello";
   int  bytesin;
   pid_t  childpid;
   int fd[2];

   if (pipe(fd) == -1 ){
      perror("failed to create pipe"); exit(23);
      }

   bytesin=strlen(bufin);
   childpid = fork();
   if (childpid == -1) { perror("failed to fork"); exit (23);}
   if (childpid)  // parent code
       write(fd[1],bufout,strlen(bufout)+1);
   else           // child code
       bytesin=read(fd[0],bufin,strlen(bufin)+1);

   printf("[%ld]: my bufin is {%.*s}, my bufout is {%s} (parent process %ld)\n",
       (long)getpid(), bytesin, bufin, bufout, (long)getppid());
   return 0;
}
```

Outcome:

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
[6679]: my bufin is {empty}, my bufout is {hello} (parent process 3420)
[6680]: my bufin is {hello}, my bufout is {hello} (parent process 6679)
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

Another Example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#define  READ 0
#define  WRITE 1
#define  BUFSIZE 100

char *mystring = "This is a test only";

int main(void){
  pid_t  pid;
  int fd[2], bytes;
  char message[BUFSIZE];

  if (pipe(fd) == -1){ perror("pipe"); exit(1); }

  if ( (pid = fork()) == -1 ){ perror("fork"); exit(1); }
  if ( pid == 0 ){    //child
     close(fd[READ]);
     write(fd[WRITE], mystring, strlen(mystring)+1);
     close(fd[WRITE]);
     }
  else{                 // parent
     close(fd[WRITE]);
     bytes=read(fd[READ], message, sizeof(message));
     printf("Read %d bytes: %s \n",bytes, message);
     close(fd[READ]);
     }
}
```

Outcome:

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
Read 20 bytes: This is a test only
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

- Anytime, *read/write-ends* are not any more needed, make sure they are closed off.

read() call and pipes

- If a process has opened up a pipe for *write* but has not written anything yet, a potential *read* blocks.
- if a pipe is empty and no process has the pipe open for *write*, a *read* returns 0.

Yet another example

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

#define  READ 0
#define  WRITE 1

int main(int argc, char *argv[]){
  pid_t  pid;
  int fd[2], bytes;

  if (pipe(fd) == -1){ perror("pipe"); exit(1); }
  if ( (pid = fork()) == -1 ){ perror("fork"); exit(1); }

  if ( pid != 0 ){            // parent and writer
     close(fd[READ]);
     dup2(fd[WRITE],1);
     close(fd[WRITE]);
     execlp(argv[1], argv[1], NULL); // Anything that argv[1] writes,
     perror("execlp");               // goes to the pipe.
     }
  else{                       // child and reader
     close(fd[WRITE]);
     dup2(fd[READ],0);
     close(fd[READ]);
     execlp(argv[2],argv[2],NULL);  // Anything that argv[2] reads,
     }                              // is obtained from the pipe.
}
```

Some outcomes:

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out ls wc
ad@ad-desktop:~/SysProMaterial/Set005/src$       22       22
        244

ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out ps sort
 3420 pts/4    00:00:00 bash
 6849 pts/4    00:00:00 ps
 6850 pts/4    00:00:00 sort
  PID TTY          TIME CMD
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out ls head
a.out
exec-demo.c
execvp-1.c
execvp-2.c
fork1.c
fork2.c
mychain.c
mychild2.c
myexit.c
mygetlimits.c
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

What happens to file descriptors/pipes after an *exec*

- A copy of file descriptors and pipes are inherited by the child (as well as the signal state and the scheduling parameters).
- Although the file descriptors are "available" and accessible by the child, their **symbolic names are not!!**
- How do we "pass" descriptors to the program called by *exec*?
  - pass such descriptors as inline parameters
  - use standard file descriptors: 0, 1 and 2 ("as is").

Main program that creates a child and calls *execlp*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

#define READ 0
#define WRITE 1

main(int argc, char *argv[]){
    int fd1[2], fd2[2], filedesc1= -1;
    char myinputparam[20];
    pid_t pid;

    // create a number of file(s)/pipe(s)/etc
    if ( (filedesc1=open("MytestFile", O_WRONLY|O_CREAT, 0666)) == -1){
        perror("file creation"); exit(1);
        }
    if ( pipe(fd1) == -1 ) {
        perror("pipe"); exit(1);
        }
    if ( pipe(fd2)== -1 ) {
                perror("pipe"); exit(1);
                }
```

```
    if ( (pid=fork()) == -1){
        perror("fork"); exit(1);
        }

    if ( pid!=0 ){          // parent process - closes off everything
        close(filedesc1);
        close(fd1[READ]); close(fd1[WRITE]);
        close(fd2[READ]); close(fd2[WRITE]);
        close(0); close(1); close(2);
        if (wait(NULL)!=pid){
            perror("Waiting for child\n"); exit(1);
            }
        }
    else {
        printf("filedesc1=%d\n", filedesc1);
        printf("fd1[READ]=%d, fd1[WRITE]=%d,\n",fd1[READ], fd1[WRITE]);
        printf("fd2[READ]=%d, fd2[WRITE]=%d\n", fd2[READ], fd2[WRITE]);
        dup2(fd2[WRITE], 11);
        execlp(argv[1], argv[1], "11", NULL);
        perror("execlp");
    }
}
```

Program that replaces the image of the child

```
#include <stdio.h> .....
#define READ 0
#define WRITE 1

main(int argc, char *argv[] ) {
    char message[]="Hello there!";
    // although the program is NOT aware of the logical names
    // can access/manipulate the file descriptors!!!
    printf("Operating after the execlp invocation! \n");
    if ( write(3,message, strlen(message)+1)== -1)
        perror("Write to 3-file \n");
    else    printf("Write to file with file descriptor 3 succeeded\n");
    if ( write(5, message, strlen(message)+1) == -1)
        perror("Write to 5-pipe");
    else    printf("Write to pipe with file descriptor 5 succeeded\n");
    if ( write(7, message, strlen(message)+1) == -1)
        perror("Write to 7-pipe");
    else    printf("Write to pipe with file descriptor 7 succeeded\n");
    if ( write(11, message, strlen(message)+1) == -1)
                perror("Write to 11-dup2");
        else    printf("Write to dup2ed file descriptor 11 succeeded\n");
    if ( write(13, message, strlen(message)+1) == -1)
                perror("Write to 13-invalid");
        else    printf("Write to invalid file descriptor 13 not feasible\n");
    return 1;
}
```

Running these (two) programs (with the help of *execlp*)

Execution without parameters:

```
ad@ad-desktop:~/Set005/src$ ./a.out
filedesc1=3
fd1[READ]=4, fd1[WRITE]=5,
fd2[READ]=6, fd2[WRITE]=7
ad@ad-desktop:~/Set005/src$
ad@ad-desktop:~/Set005/src$
```

Execution with a parameter:

```
ad@ad-desktop:~/Set005/src$ ./a.out ./write-portion
filedesc1=3
fd1[READ]=4, fd1[WRITE]=5,
fd2[READ]=6, fd2[WRITE]=7
Operating after the execlp invocation!
Write to file with file descriptor 3 succeeded
Write to pipe with file descriptor 5 succeeded
Write to pipe with file descriptor 7 succeeded
Write to dup2ed file descriptor 11 succeeded
Write to 13-invalid: Bad file descriptor
ad@ad-desktop:~/Set005/src$
```

“Limitations” of Pipes

Classic pipes have at least two drawbacks:

- Processes using pipes must share **common ancenstry**.
- Pipes are **NOT** permanent (persistent).

Named Pipes (FIFOs)

- The **FIFOs** ("**named pipes**") address above defficiencies.
  - FIFOs are permanent on the file system
  - Enable the first-in/first-out one communication channel.
  - *read* and *write* operations function similarly to pipes.
  - A FIFO has an owner and access permissions (as usual).
  - A FIFO can be opened, read, written and finally closed.
  - A FIFO **cannot be** seeked.
  - Blocking and non-blocking version using the <*fcntl.h*>
    $\rightarrow$ why non-blocking FIFOs??

Creation of FIFOs

- System program: */bin/mknod nameofpipe p*
  - *nameofpipe* name of FIFO.
  - Note the *p* parameter above and the *p* in *prw-r–r–* below:

```
ad@ad-desktop:~/Set005/src$ mknod kitsos p
ad@ad-desktop:~/Set005/src$ ls -l kitsos
prw-r--r-- 1 ad ad 0 2010-04-22 16:58 kitsos
ad@ad-desktop:~/Set005/src$ man mkfifo
```

- System call:*int mkfifo(const char *pathname, mode_t mode)*
  - *pathname*: whete the FIFO is created on the filesuystem
  - included files:

    #include <sys/types.h>

    #include <sys/stat.h>
  - *mode* represents the designated access permissions for: owner, group, others.

A simple client-server application with a FIFO

→ "server program:" receivemessages.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define MSGSIZE 65

char *fifo = "myfifo";

main(int argc, char *argv[]){
        int fd, i, nwrite;
        char msgbuf[MSGSIZE+1];

        if (argc>2) {
                printf("Usage: receivemessage & \n");
                exit(1);
                }

        if ( mkfifo(fifo, 0666) == -1 ){
                if ( errno!=EEXIST ) {
                        perror("receiver: mkfifo");
                        exit(6);
                        }
                }
```

receivemessages.c

```
        if ( (fd=open(fifo, O_RDWR)) < 0){
                perror("fifo open problem");
                exit(3);
                }
        for (;;){
                if ( read(fd, msgbuf, MSGSIZE+1) < 0) {
                        perror("problem in reading");
                        exit(5);
                        }
                printf("\nMessage Received: %s\n", msgbuf);
                fflush(stdout);
                }
}
```

“client program:” sendmessages.c

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <error.h>
#define MSGSIZE 65
char *fifo = "myfifo";

main(int argc, char *argv[]){
    int fd, i, nwrite;
    char msgbuf[MSGSIZE+1];

    if (argc<2) { printf("Usage: sendamessage ... \n"); exit(1); }
    if ( (fd=open(fifo, O_WRONLY| O_NONBLOCK)) < 0)
        { perror("fife open error"); exit(1); }

    for (i=1; i<argc; i++){
        if (strlen(argv[i]) > MSGSIZE){
            printf("Message with Prefix %.*s Too long - Ignored\n",10,argv[i]);
            fflush(stdout);
            continue;
            }
        strcpy(msgbuf, argv[i]);
        if ((nwrite=write(fd, msgbuf, MSGSIZE+1)) == -1)
            { perror("Error in Writing"); exit(2); }
        }
    exit(0);
}
```

Running the client-server application

→ Setting up the "server":

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./receivemessages &
[1] 2662
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

→ Running the "client":

```
ad@ad-desktop:~/Set005/src$ ./sendmessages Nikos
ad@ad-desktop:~/Set005/src$ ./sendmessages Nikos "Alexis Delis"
ad@ad-desktop:~/Set005/src$ ./sendmessages "Alex Delis" "Apostolos Despotopoulos
     "
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./sendmessages Thessaloniki+Komotini+
     Xanthi+Kavala kkkkkkkkkkkkkkkkkkkkkkkkk....kkkkkkkkkk
Message with Prefix kkkkkkkkk Too long - Ignored
ad@ad-desktop:~/Set005/src$
```

→ Observing the behavior of the "server":

```
ad@ad-desktop:~/Set005/src$
Message Received: Nikos
Message Received: Nikos
Message Received: Alexis Delis
Message Received: Alex Delis
Message Received: Apostolos Despotopoulos
Message Received: Thessaloniki+Komotini+Xanthi+Kavala
```