Processes
Process Control
Pipes

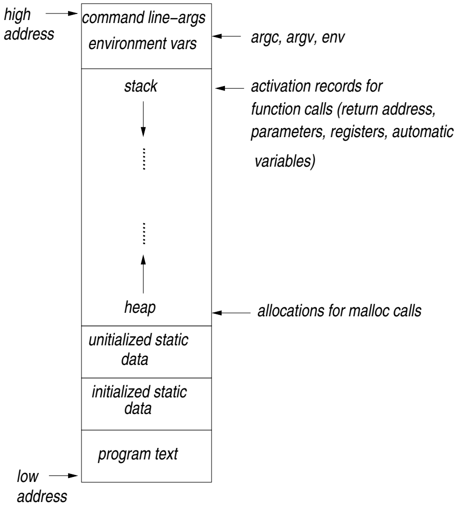
A Unix Process

- Instance of a program in execution.
- OS "loads" the executable in main-memory (core) and starts execution by accessing the first command.
- Each process has a *unique* identifier, its *process-ID*.
- Every process maintains:
  - Program text.
  - Run-time stack(s).
  - Initialized and uninitialized data.
  - Run-time data.

Processes

- Program counter: designates which instruction will be executed next by the CPU.
- Processes communicate among themselves through a number of (IPC) mechanisms including: files, pipes, shared memory, sockets, etc..

Process Instance



Processes...

- 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 to have another process *clone 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 descendants 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 Limits

- Two system calls help get/set limits:

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

- Each resource is associated with two limits:

```
struct rlimit {
     rlim_t rlim_cur;  /* Soft limit -- Current Limit       */
     rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
                       /*  max value for current limit      */
     };
```

- *getrlimit()* returns 0 if all is ok, otherwise a value<>0.
- *setrlimit()* returns 0 if all is ok, otherwise a value<>0.

A program getting the limits

```c
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>

int main(){
struct rlimit myrlimit;
// RLIMIT_AS: maximum size of process's virtual memory in bytes
getrlimit(RLIMIT_AS, &myrlimit);
printf("Maximum address space = %lu and current = %lu\n",
    myrlimit.rlim_max, myrlimit.rlim_cur);

// RLIMIT_CORE: Maximum size of core file
getrlimit(RLIMIT_CORE, &myrlimit);
printf("Maximum core file size = %lu and current = %lu\n",
    myrlimit.rlim_max, myrlimit.rlim_cur);

// RLIMIT_DATA: maximum size of files that the process may create
getrlimit(RLIMIT_DATA, &myrlimit);
printf("Maximum data+heap size = %lu and current = %lu\n",
    myrlimit.rlim_max, myrlimit.rlim_cur);

// RLIMIT_FSIZE: maximum size of files that the process may create
getrlimit(RLIMIT_FSIZE, &myrlimit);
printf("Maximum file size = %lu and current = %lu\n",
    myrlimit.rlim_max, myrlimit.rlim_cur);

// RLIMIT_STACK: maximum size of the process stack, in bytes.
getrlimit(RLIMIT_STACK, &myrlimit);
printf("Maximum stack size = %lu and current = %lu\n",
    myrlimit.rlim_max, myrlimit.rlim_cur);
}
```

Running the Program

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
Maximum address space = 4294967295 and current = 4294967295
Maximum core file size = 4294967295 and current = 0
Maximum data+heap size = 4294967295 and current = 4294967295
Maximum file size = 4294967295 and current = 4294967295
Maximum stack size = 4294967295 and current = 8388608
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

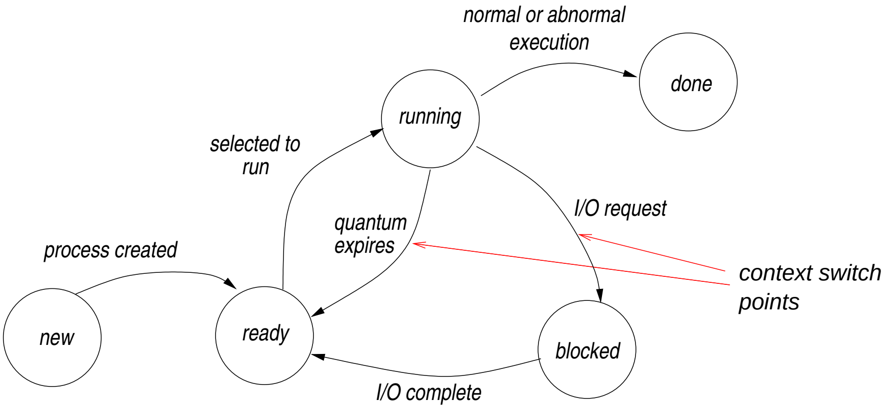
Process IDs

-
  ```
        #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$
```

Process State Diagram



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 (in bash) as 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 $?
157
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

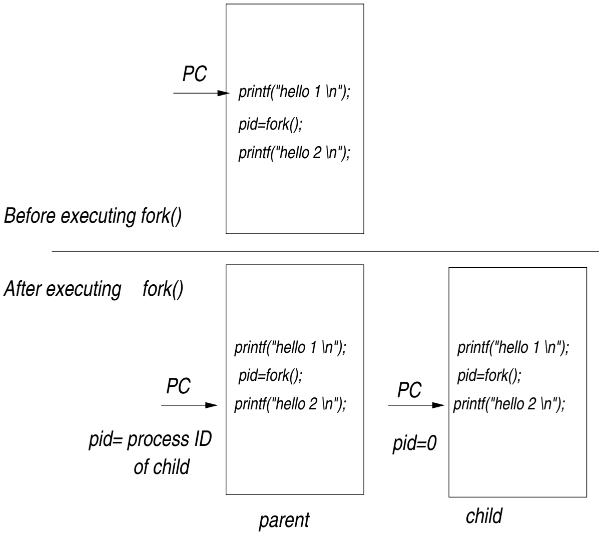
Creating a new process – *fork()*

- 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 it 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
```

Creating a Shallow Tree

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

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

  if (argc!=2){
     printf("Usage: %s number-of-processes \n",argv[0]);
     exit(1);
     }
  n = atoi(argv[1]);
  for (i=1;i<n; i++)
     if ( (childpid = fork()) <= 0 )
        break;

  printf("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 4
i: 1 process ID: 16721 parent ID:16720 child ID:0
i: 2 process ID: 16722 parent ID:16720 child ID:0
i: 4 process ID: 16720 parent ID:3256 child ID:16723
i: 3 process ID: 16723 parent ID:16720 child ID:0
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

Orphan Processes

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

int main(void){
   pid_t pid;

   printf("Original process: PID = %d, PPID = %d\n",getpid(), getppid());
   pid = fork();
   if ( pid == -1 ){
      perror("fork"); exit(1);
      }
   if ( pid != 0 )
      printf("Parent process: PID = %d, PPID = %d, CPID = %d \n",
             getpid(), getppid(), pid);
   else {
      sleep(2);
      printf("Child process: PID = %d, PPID = %d \n",
             getpid(), getppid());
      }
   printf("Process with PID = %d terminates \n",getpid());
}
```

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
Original process: PID = 16816, PPID = 3256
Parent process: PID = 16816, PPID = 3256, CPID = 16817
Process with PID = 16816 terminates
Child process: PID = 16817, PPID = 1
Process with PID = 16817 terminates
ad@ad-desktop:~/SysProMaterial/Set005/src$
```

The *wait()* call

-
  ```c
  #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 spawned.

- 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.

Checking the *status* flag

The int *status* could be checked with the help of the following macros:

- WIFEXITED(status): returns true if the child terminated normally.
- WEXITSTATUS(status): returns the exit status of the child. This consists of the 8 bits of the *status* argument that the child specified in an *exit()* call or as the argument for a return statement in *main()*. This macro should only be used if WIFEXITED returned true.
- WIFSIGNALED(status): returns true if the child process was terminated by a signal.
- WTERMSIG(status): returns the number of the signal that caused the child process to terminate. This macro should only be employed if WIFSIGNALED returned true.
- WCOREDUMP(status): returns true if the child produced a core dump.
- WSTOPSIG(status): returns the number of the signal which caused the child to stop.

Use of *wait*

```
#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

-
  ```c
  #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 $= |\text{pid}|$
  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).

waitpid() example

```
#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);
}
```

Output

```
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$
```

Zombie Processes

- A process that terminates remains in the system until its parent *receives* its exit code.
- All this time, the process is a **zombie**.

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

int main(void){
   pid_t pid;

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

   if ( pid!=0 ){
      while(1){
        sleep(500);
        }
      }
   else {
      exit(37);
      }
}
```

Example with Zombie

```
d@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out &
[1] 17508
ad@ad-desktop:~/SysProMaterial/Set005/src$ ps -a
  PID TTY          TIME CMD
 5822 pts/5    00:00:00 ssh
 7508 pts/3    00:00:00 man
13324 pts/1    00:15:02 soffice.bin
13772 pts/0    00:00:00 ssh
15111 pts/1    00:00:03 gv
17433 pts/1    00:00:00 gs
17508 pts/6    00:00:00 a.out
17509 pts/6    00:00:00 a.out <defunct>
17510 pts/6    00:00:00 ps
ad@ad-desktop:~/SysProMaterial/Set005/src$ kill -9 17508
[1]+  Killed                  ./a.out
ad@ad-desktop:~/SysProMaterial/Set005/src$ ps -a
  PID TTY          TIME CMD
 5822 pts/5    00:00:00 ssh
 7508 pts/3    00:00:00 man
13324 pts/1    00:15:02 soffice.bin
13772 pts/0    00:00:00 ssh
15111 pts/1    00:00:03 gv
17433 pts/1    00:00:00 gs
17512 pts/6    00:00:00 ps
ad@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*

-
  ```c
  #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 invoking 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.
- Note that, by convention, the filename of the executable is passed as an argument, although it is typically identical to the last part of the full path.
- *execle* requires the passing of environment variables in *envp[0], envp[1], envp[2],..,envp[n]* and NULL as delimeter in *envp[n+1]*.

Using *execl()*

```
#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$
```

Problem Statement

Create a complete binary tree of processes. For each process that is not a leaf, print out its ID, the ID of its parent and a logical numeric ID that facilitates a breadth-first walk.

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

int main(int argc, char *argv[]){
   int i, depth, numb, pid1, pid2, status;

   if (argc >1)  depth = atoi(argv[1]);
   else { printf("Usage: %s #-of-Params",argv[0]); exit(0);}

   if (depth>5) {
      printf("Depth should be up to 5\n");
      exit(0);
      }
   numb = 1;
   for(i=0;i<depth;i++){
      printf("I am process no %5d  with PID %5d and PPID %d\n",
              numb, getpid(), getppid());
      switch (pid1=fork()){
      case 0:
         numb=2*numb; break;
      case -1:
         perror("fork"); exit(1);
      default:
         switch (pid2=fork()){
            case 0:
               numb=2*numb+1; break;
            case -1:
               perror("fork"); exit(1);
            default:
               wait(&status); wait(&status);
               exit(0);
            }
     }
   }
}
```

Running the executable

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out 1
I am process no     1  with PID  4147 and PPID 3420
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out 2
I am process no     1  with PID  4150 and PPID 3420
I am process no     2  with PID  4151 and PPID 4150
I am process no     3  with PID  4152 and PPID 4150
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out 3
I am process no     1  with PID  4158 and PPID 3420
I am process no     2  with PID  4159 and PPID 4158
I am process no     3  with PID  4160 and PPID 4158
I am process no     4  with PID  4161 and PPID 4159
I am process no     6  with PID  4162 and PPID 4160
I am process no     5  with PID  4167 and PPID 4159
I am process no     7  with PID  4168 and PPID 4160
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out 4
I am process no     1  with PID  4173 and PPID 3420
I am process no     2  with PID  4174 and PPID 4173
I am process no     3  with PID  4175 and PPID 4173
I am process no     4  with PID  4176 and PPID 4174
I am process no     6  with PID  4177 and PPID 4175
I am process no     8  with PID  4178 and PPID 4176
I am process no    12  with PID  4179 and PPID 4177
I am process no     9  with PID  4184 and PPID 4176
I am process no    13  with PID  4185 and PPID 4177
I am process no     5  with PID  4190 and PPID 4174
I am process no     7  with PID  4191 and PPID 4175
I am process no    10  with PID  4192 and PPID 4190
I am process no    14  with PID  4193 and PPID 4191
I am process no    11  with PID  4198 and PPID 4190
I am process no    15  with PID  4199 and PPID 4191
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 way that Unix addresses one-way communication between two processes (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 a *read* call.

Pipes

-
  ```c
  #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**.

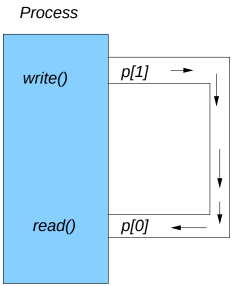
- A pipe's real value appears when it is used in conjunction with a *fork()* and the fact that the file descriptors remain open across a *fork()*.

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$
```

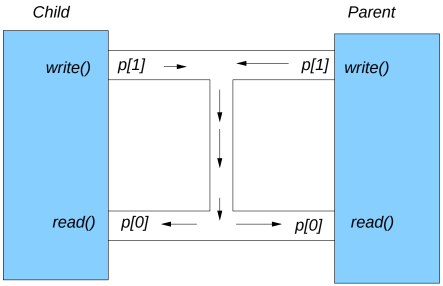


The output and the input are part of the **same** process - not useful!

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

```c
#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:

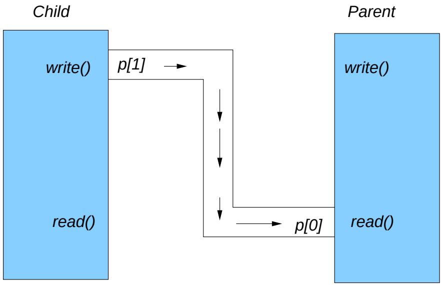


- Either process could write down the file descriptor *p[1]*.
- Either process could read from the file descriptor *p[0]*.
- Problem: pipes are intended to be unidirectional; if both processes start reading and writing indiscriminately, **chaos may ensue**.

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:



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 (pretty cool this time)

```
#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$
```

The size of a pipe

- The size of data that can be written down a pipe is **finite**.
- If there is no more space left in the pipe, an impending write will *block* (until space becomes again available).
- POSIX designates this limit to be 512 Bytes.
- Unix systems often display *much higher* capacity for this buffer area.
- The following is a small program that helps discover "real" upper-bounds for this size in a system.

The *size* program (also cool + note call `fpathconf`)

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

int count=0;
void alrm_action(int);

main(){
  int p[2];
  int pipe_size=0;
  char c='x';
  static struct sigaction act;

  // set up the signal handler
  act.sa_handler=alrm_action;
  sigfillset(&(act.sa_mask));

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

  pipe_size=fpathconf(p[0], _PC_PIPE_BUF);
  printf("Maximum size of (atomic) write to pipe: %d bytes\n", pipe_size);
  printf("The respective POSIX value %d\n",_POSIX_PIPE_BUF);

  sigaction(SIGALRM, &act, NULL);
```

The *size* program

```
  while (1) {
        alarm(20);
        write(p[1], &c, 1);
        alarm(0);
        if (++count % 4096 == 0 )
           printf("%d characters in pipe\n",count);
      }
}

void alrm_action(int signo){
  printf("write blocked after %d characters \n",count);
  exit(0);
}
```

Outcome of execution:

```
ad@ad-desktop:~/SysProMaterial/Set005/src$ ./a.out
Maximum size of (atomic) write to pipe: 4096 bytes
The respective POSIX value 512
4096 characters in pipe
8192 characters in pipe
12288 characters in pipe
16384 characters in pipe
20480 characters in pipe
24576 characters in pipe
28672 characters in pipe
32768 characters in pipe
36864 characters in pipe
40960 characters in pipe
45056 characters in pipe
49152 characters in pipe
53248 characters in pipe
57344 characters in pipe
61440 characters in pipe
65536 characters in pipe
write blocked after 65536 characters
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!!**. We have no variable to refer to the open file descriptor by! Need to know its constant number.
- 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*

```c
#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");
    }
}
```

The other half: 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$
```

redirecting output in an unusual way (hack)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define SENTINEL -1

main(){
    pid_t pid;
    int fd=SENTINEL;

    printf("About to run who into a file (in a strange way!)\n");
    if ( (pid=fork())== SENTINEL){
        perror("fork"); exit(1);
        }

    if ( pid == 0 ){  // child
        close(1);
        fd=creat("userlist", 0644); // should get 1 as fd!!!
        execlp("who","who",NULL);
        perror("execlp");
        exit(1);
        }
    if ( pid != 0){  // parent
        wait(NULL);
        printf("Done running who - results in file \"userlist\"\n");
        }
}
```

Running the program

```
ad@ad-desktop:~/Set005/src$ ./a.out
About to run who into a file (in a strange way!)
Done running who - results in file "userlist"
ad@ad-desktop:~/Set005/src$
ad@ad-desktop:~/Set005/src$
ad@ad-desktop:~/Set005/src$ more userlist
The file descriptor value is 1
ad       tty7         2010-04-21 07:32 (:0)
ad       pts/0        2010-04-21 07:33 (:0.0)
ad       pts/1        2010-04-21 08:01 (:0.0)
ad       pts/2        2010-04-21 11:09 (:0.0)
ad       pts/3        2010-04-21 13:11 (:0.0)
ad       pts/4        2010-04-21 15:19 (:0.0)
ad@ad-desktop:~/SysProMaterial/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)

- **FIFOs** ("**named pipes**") address above deficiencies.
  - FIFOs are permanent on the file system
  - Enable a first-in/first-out 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*>
    → 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 filesystem
  - 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("fifo 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 kkkkkkkkkkkkkkkkkkkkkk....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
```