Processes
Process Control
Pipes

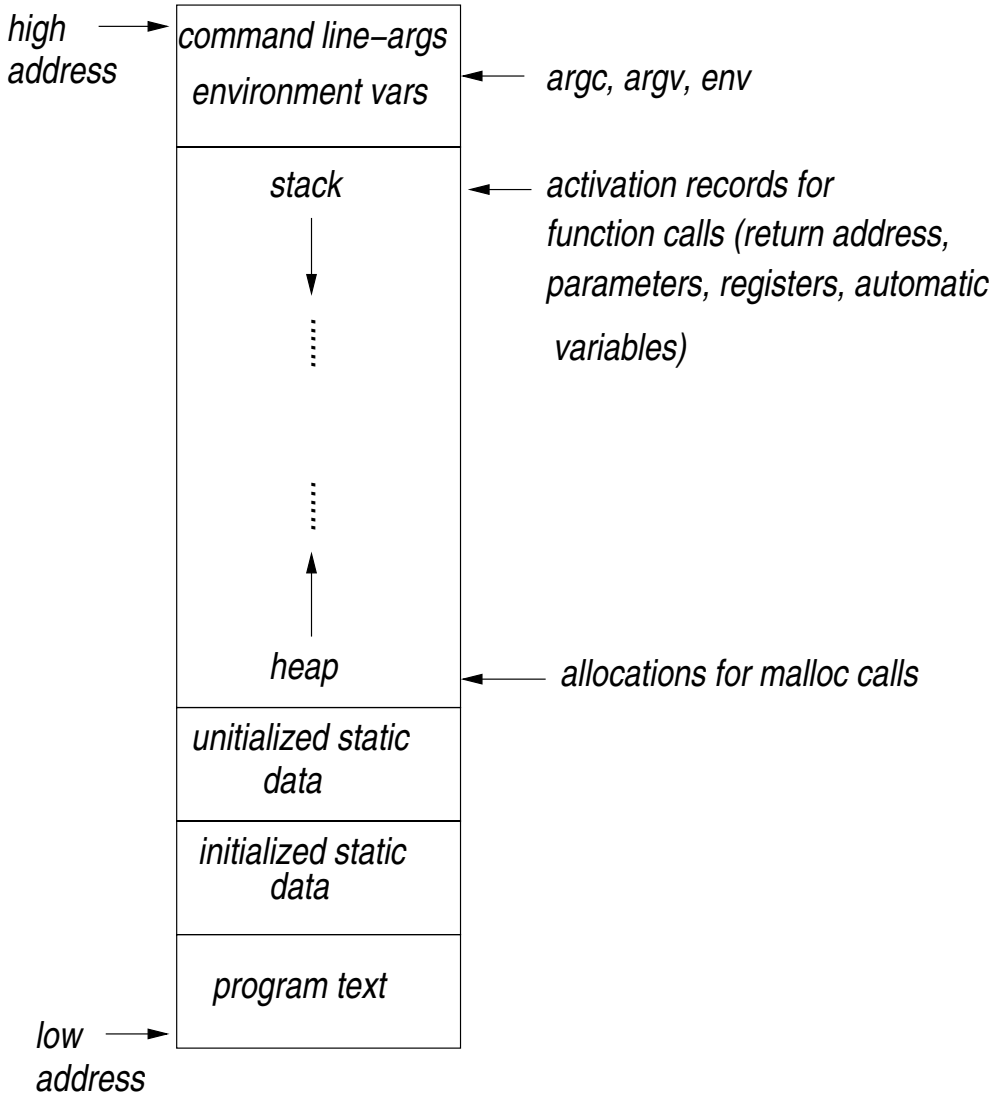
A Unix Process

- 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.
  - Initialized and uninitialized data.
  - Run-time data.

Processes

- Each process commences and goes about its execution by continuously fetching the *next* operation along with its operands (as designated by the assembly language specification).
- *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, fifos, streams, 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` variables, 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 user 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 Limits

- Every process has a set of resource limits that can be queried and/or set.
- 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

```
#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.

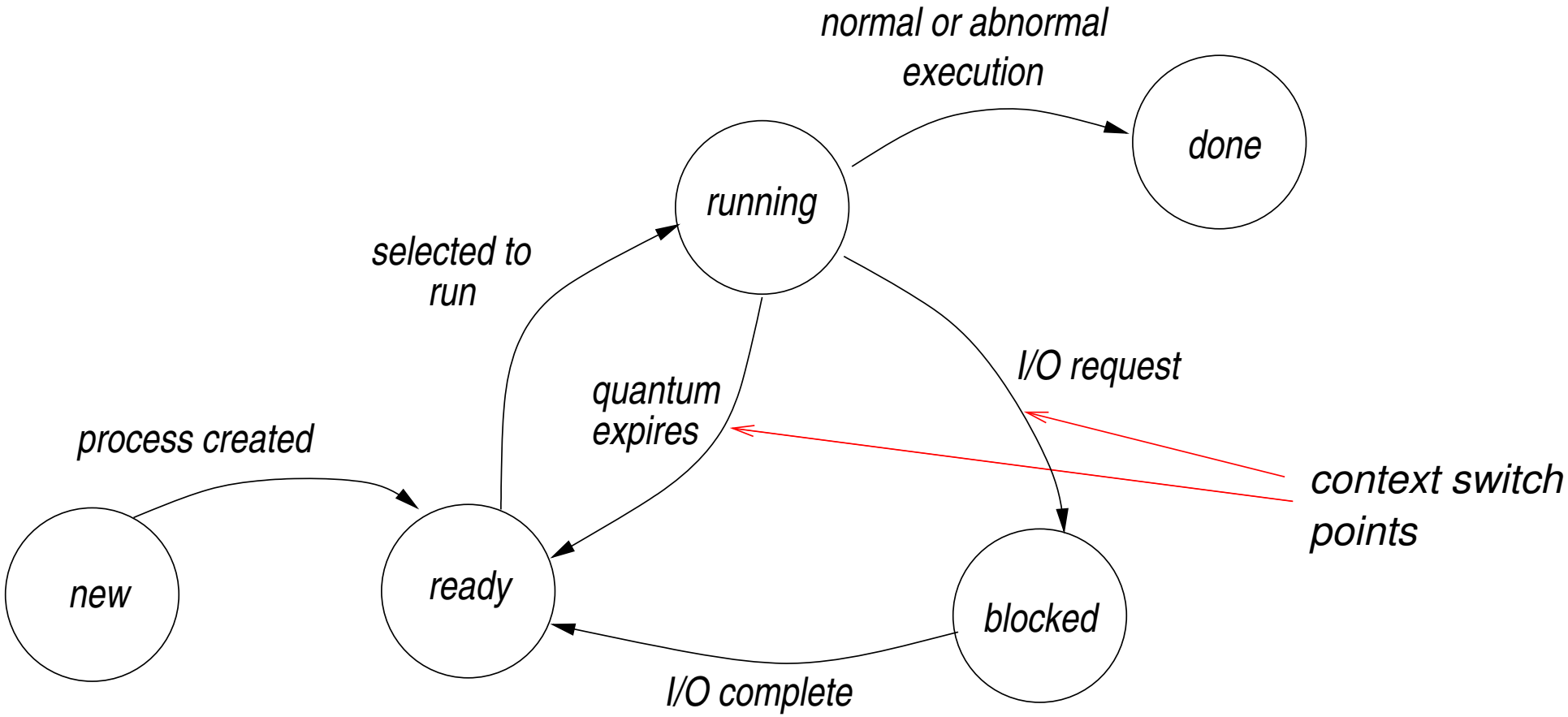
- ```
  #include <stdio.h>
  #include <unistd.h>

  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;
  }
  ```

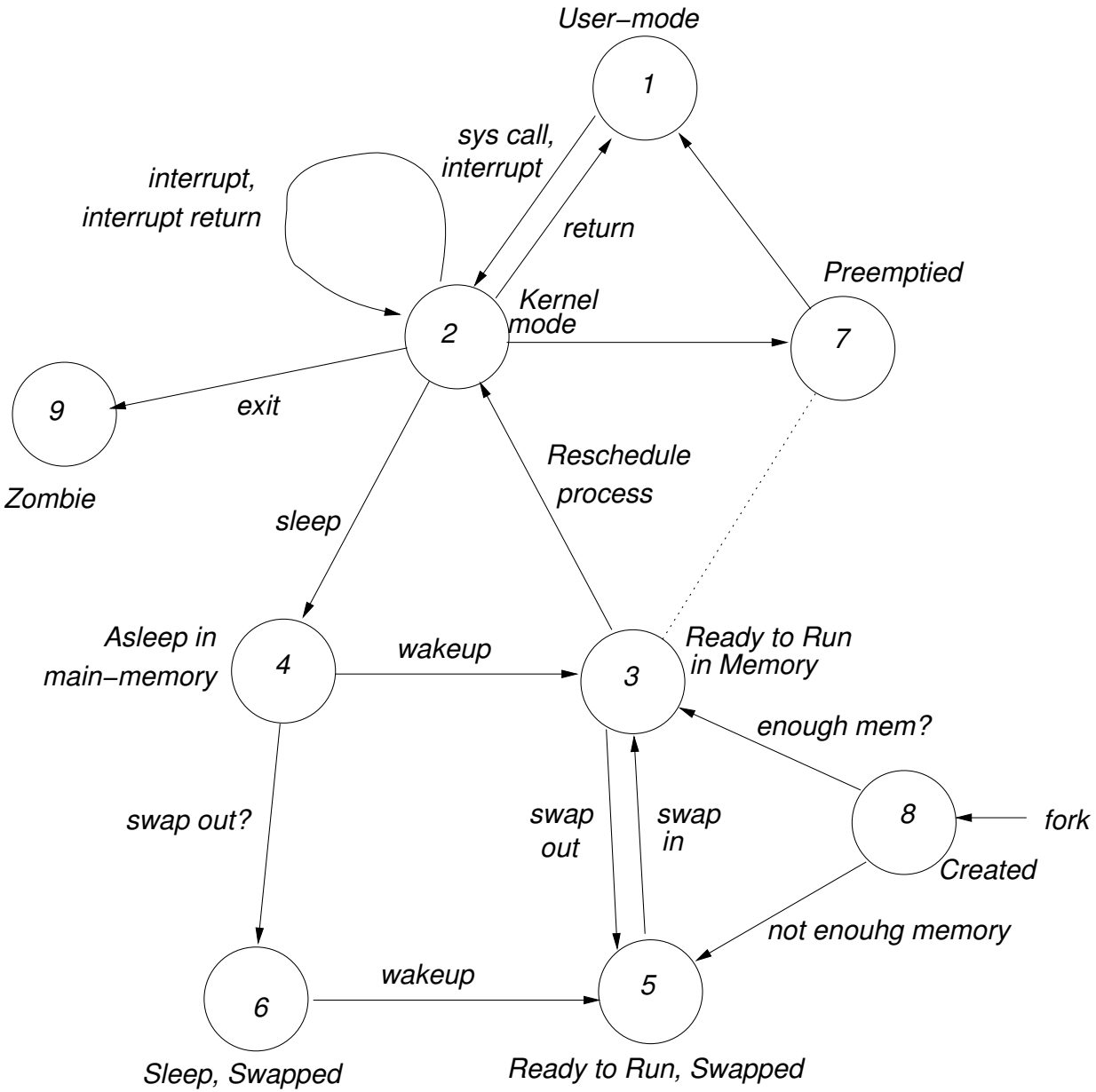
- Running the program...

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

Process State Diagram



Detailed State Diagram in AT&T Unix



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

Creating a new process – `fork()`

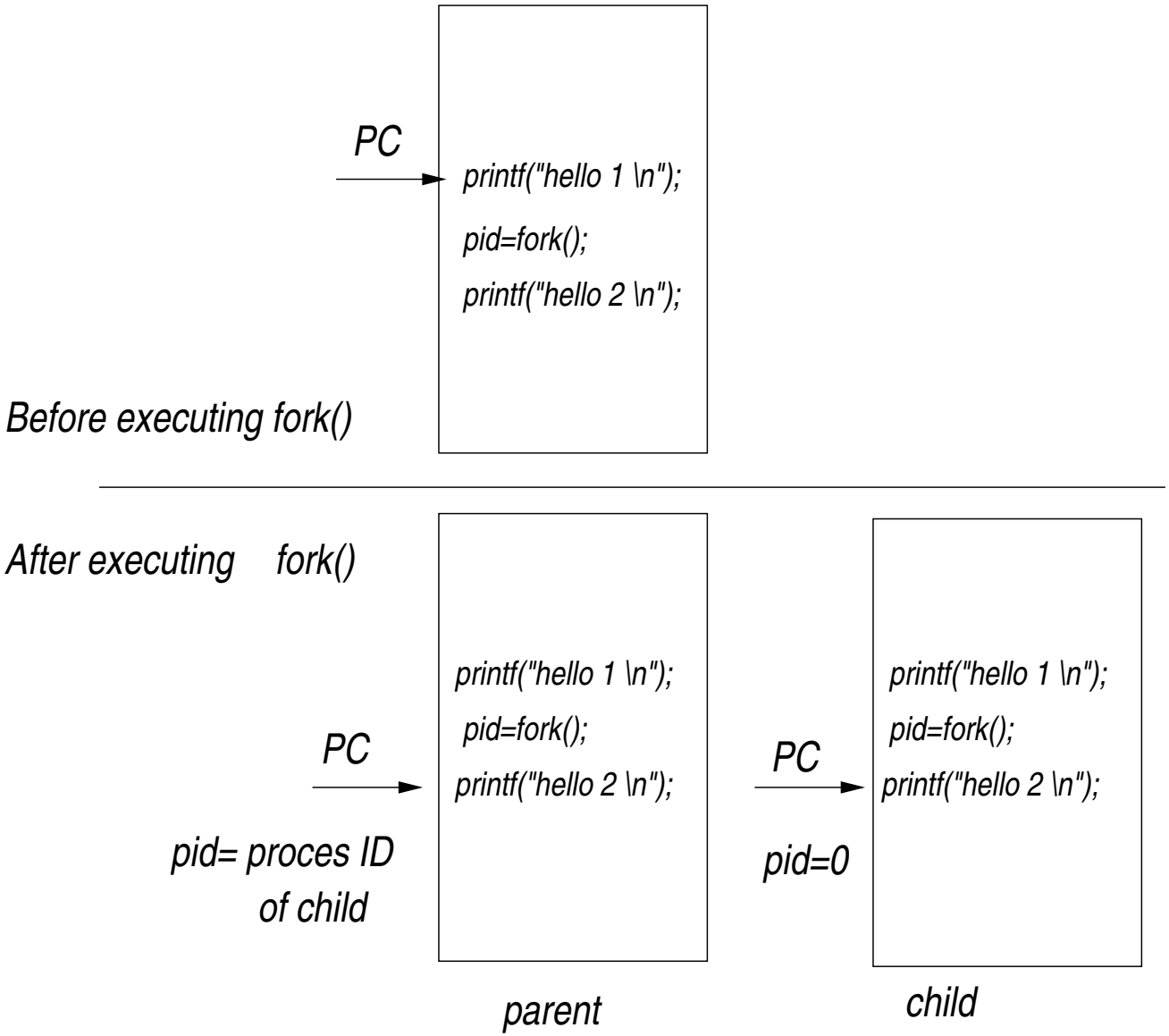
- The system call:

```
#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:~/src$ ./a.out
I am the parent process with ID: 15373
I am the child process with ID: 15374
ad@ad-desktop:~/src$ ./a.out
I am the parent process with ID: 15375
I am the child process with ID: 15376
ad@ad-desktop:~/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:~/src$ ./a.out
I am the child process with ID: 15704 -- 15703
I am the parent process with ID: 15703 -- 15703
ad@ad-desktop:~/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;
}

fprintf(stdout,">>> process ID:%ld parent ID:%ld child ID:%ld\n",
        (long)getpid(),(long)getppid(),(long)childpid );

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

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

Creating a (deep) chain of processes

```
ad@haiku:~/src-set005$ ./mychain-p17 2
>>> process ID:17980 parent ID:5724 child ID:0
i:1 process ID:17980 parent ID:5724 child ID:17981
i:2 process ID:17981 parent ID:17980 child ID:0
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$ ./mychain-p17 4
>>> process ID:17984 parent ID:5724 child ID:0
i:1 process ID:17984 parent ID:5724 child ID:17985
i:2 process ID:17985 parent ID:17984 child ID:17986
i:3 process ID:17986 parent ID:17985 child ID:17987
i:4 process ID:17987 parent ID:17986 child ID:0
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$ echo $$
5724
ad@haiku:~/src-set005$
```

Creating a Shallow Tree

```c
#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);
     }

  printf("The id of the initial process is %d\n",getpid());

  n = atoi(argv[1]);
  for (i=1;i<n; i++)
        if ( (childpid = fork()) == 0 )
            break;

  printf("i: %d process ID: %d parent ID:%d child ID:%d\n",
                 i, getpid(), getppid(), childpid);
  sleep(1);
  return 0;
}
```

Output Shallow Tree

```
ad@haiku:~/src-set005$ ./treeofprocs-p19 2
The id of the initial process is 18601
i: 1 process ID: 18602 parent ID:18601 child ID:0
i: 2 process ID: 18601 parent ID:5724 child ID:18602
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$ ./treeofprocs-p19 3
The id of the initial process is 18607
i: 1 process ID: 18608 parent ID:18607 child ID:0
i: 2 process ID: 18609 parent ID:18607 child ID:0
i: 3 process ID: 18607 parent ID:5724 child ID:18609
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$ ./treeofprocs-p19 4
The id of the initial process is 18612
i: 1 process ID: 18613 parent ID:18612 child ID:0
i: 3 process ID: 18615 parent ID:18612 child ID:0
i: 2 process ID: 18614 parent ID:18612 child ID:0
i: 4 process ID: 18612 parent ID:5724 child ID:18615
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$ echo $$
5724
ad@haiku:~/Dropbox/k24/Transparencies/Set005/src-set005$
```

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@haiku:~/src-set005$ ./myorphan-p21
Original process: PID=19616, PPID=5724
Parent process: PID=19616, PPID=5724, CPID=19617
Process with PID=19616 terminates
ad@haiku:~/src-set005$ Child process: PID=19617, PPID=1983
Process with PID=19617 terminates

ad@haiku:~/src-set005$
```

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 0s.
  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 value of the parameter `status` can 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

```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(10); }
```

```
ad@ad-desktop:~/src$ ./a.out
Hello I am in parent process 17022 with child 17023
exit status from 17023 was 5
ad@ad-desktop:~/src$ echo $?
10
ad@ad-desktop:~/src$
```

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. > 0 : wait for the child whose process ID is equal to the value of `pid`.
  2. 0: wait for any child process whose process `groupID` is equal to that of the calling process.
  3. -1: wait for any child process
  4. $<$ -1: wait for any child in the process group whose process–group ID is `-pid`

- The `options` flag is a disjunction of marcos that indicate the *on-going* status of child(ren) processes.
  ```

The `waitpid` call

- `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` returns:
  1. the process ID of the child whose state just *changed*, if all goes well.
  2. if `WNOHANG` was specified and one or more child(ren) specified by `pid` exist, but *have not yet* changed state, then `0` is returned.
  3. `-1` on error.

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"); exit(1); }

   if ( pid == 0 ){
        printf("Child %d starts sleeping... \n", getpid()); sleep(5);
        printf("Child %d just finished sleeping... \n", getpid());
        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("Father %d process about to exit\n", 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@haiku:~/src-set005$
ad@haiku:~/src-set005$ ./waitpid-p26
Reaching the father 10214 process
PID is 10215
Still waiting for child to return
Child 10215 starts 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
Child 10215 just finished sleeping...
Father 10214 process about to exit
Exit status from 10215 was 57
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$
```

Using `wait` (checking without macros)

```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 `wait()`

```
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$ ./wait_use-p28
Original Process: PID = 10794
Parent process: PID = 10794
Child process: PID = 10795, PPID = 10794
Child terminated: PID = 10795, exit code = 62
Process with PID = 10794 terminates
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$
```

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**.

```
#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){
                printf("Parent %d Process still alive\n",getpid());
                sleep(5);
                }
        }
   else {
        printf("Child process %d terminates!!!",getpid());
        exit(37);
        }
}
```

Example with Zombie

```
ad@haiku:~/src-set005$ ps -a
  PID TTY          TIME CMD
 3473 pts/9    00:00:26 evince
11406 pts/9    00:00:00 vi
11456 pts/0    00:00:00 zombies-p31
11457 pts/0    00:00:00 zombies-p31 <defunct>
11460 pts/0    00:00:00 ps
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$ kill -9 11457
Parent 11456 Process still alive
ad@haiku:~/src-set005$ ps -a
  PID TTY          TIME CMD
 3473 pts/9    00:00:26 evince
11406 pts/9    00:00:00 vi
11456 pts/0    00:00:00 zombies-p31
11457 pts/0    00:00:00 zombies-p31 <defunct>
11467 pts/0    00:00:00 ps
ad@haiku:~/src-set005$ Parent 11456 Process still alive
fg
./zombies-p31
^C
ad@haiku:~/src-set005$ ps -a
  PID TTY          TIME CMD
 3473 pts/9    00:00:26 evince
11406 pts/9    00:00:00 vi
11472 pts/0    00:00:00 ps
ad@haiku:~/src-set005$
```

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 invoking process with.
- `execl`, `execlp` and `execle` require the names of executable and parameters in `arg0, arg1, arg2, .., argn` with NULL following.
- `execv` and `execvp` require the passing of both executable and its arguments in a `struct`: `argv[0], argv[1], argv[2],..,argv[n]` with `NULL` as delimeter in `argv[n+1]`.
- `execle` requires the the passing of environment variables in a `struct`: `envp[0], envp[1], envp[2],..,envp[n]` with `NULL` as delimeter in `envp[n+1]`.

Using execl()

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

main(){
  int retval=0;

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

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

  if (retval==-1)
        perror("execl");
}
```

```
ad@haiku:~/src-set005$ ./exec-demo-p36
I am process 4195536 and I will execute an 'ls -l .;
total 716
-rwxr-xr-x 1 ad ad 7389          10 10:58 binarytree
-rwxr-xr-x 1 ad ad  893          10 10:58 binarytree.c
drwxr-xr-x 3 ad ad 4096          10 10:58 CHRISTOS
-rwxr-xr-x 1 ad ad 7481          10 10:58 demo-file-pipes-exec
-rwxr-xr-x 1 ad ad 1151          10 10:58 demo-file-pipes-exec.c
-rwxr-xr-x 1 ad ad 7350          10 10:58 demo-trick
-rwxr-xr-x 1 ad ad  474          10 10:58 demo-trick.c
....
-rwxrwxr-x 1 ad ad 8814           1 23:28 zombies-p30
-rwxr-xr-x 1 ad ad  403           1 23:28 zombies-p30.c
ad@haiku:~/src-set005$
```

Example with execvp()

```
#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("The sysPro to be executed is %s\n",buff[0]);

      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@haiku:~/src-set005$
ad@haiku:~/src-set005$ ./execvp-date-p37
I am the parent process 3201
The sysPro to be executed is date
I am the child process 3202 and will be replaced with 'date'
Sun Apr  3 11:17:33 EEST 2016
Child terminated with exit code 0
ad@haiku:~/src-set005$
```

Problem Statement

Create a full 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@haiku:~/src-set005$ ./exec-bintree-p39 1
I am process no     1  with PID 13581 and PPID 3729
ad@haiku:~/Dropbox/k24/Transparencies/Set005/src-set005$ ./execvp-p38 2
I am process no     1  with PID 13586 and PPID 3729
I am process no     2  with PID 13587 and PPID 13586
I am process no     3  with PID 13588 and PPID 13586
ad@haiku:~/src-set005$ ./exec-bintree-p39 3
I am process no     1  with PID 13595 and PPID 3729
I am process no     2  with PID 13596 and PPID 13595
I am process no     4  with PID 13598 and PPID 13596
I am process no     3  with PID 13597 and PPID 13595
I am process no     5  with PID 13599 and PPID 13596
I am process no     7  with PID 13603 and PPID 13597
I am process no     6  with PID 13601 and PPID 13597
ad@haiku:~/src-set005$ ./exec-bintree-p39 4
I am process no     1  with PID 13610 and PPID 3729
I am process no     2  with PID 13611 and PPID 13610
I am process no     4  with PID 13613 and PPID 13611
I am process no     3  with PID 13612 and PPID 13610
I am process no     5  with PID 13614 and PPID 13611
I am process no     6  with PID 13616 and PPID 13612
I am process no     8  with PID 13615 and PPID 13613
I am process no     7  with PID 13618 and PPID 13612
I am process no    12  with PID 13621 and PPID 13616
I am process no    14  with PID 13623 and PPID 13618
I am process no    15  with PID 13626 and PPID 13618
I am process no    13  with PID 13624 and PPID 13616
I am process no    10  with PID 13617 and PPID 13614
I am process no     9  with PID 13619 and PPID 13613
I am process no    11  with PID 13620 and PPID 13614
ad@haiku:~/src-set005$
```

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 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;

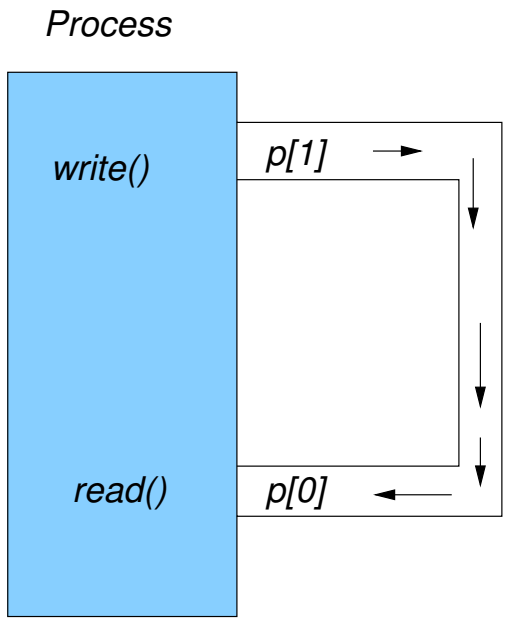
  if (pipe(p)==-1) { perror("pipe call"); exit(1);}

  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

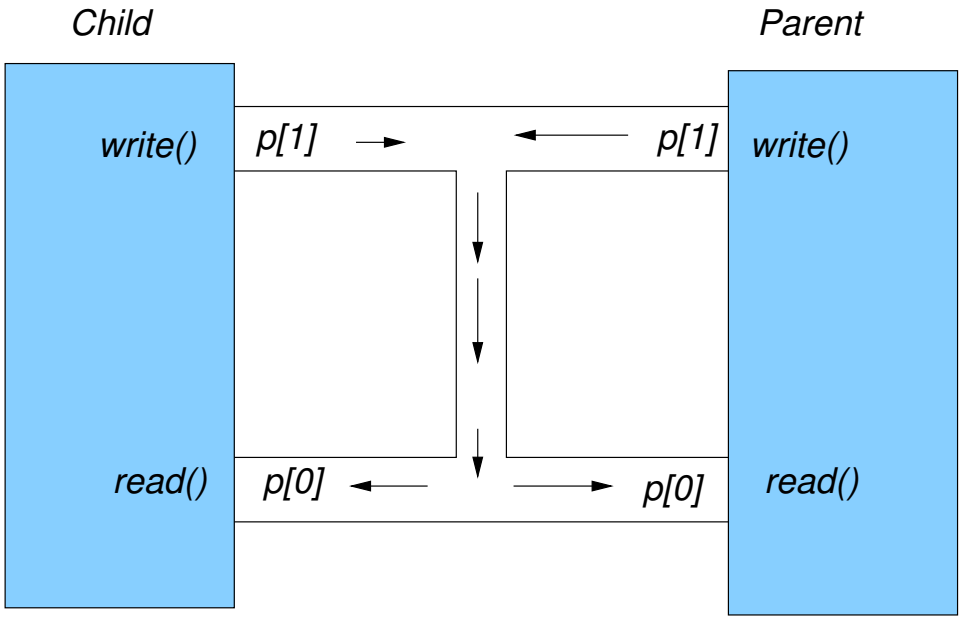
char *msg1="Buenos Dias! #1";
char *msg2="Buenos Dias! #2";
char *msg3="Buenos Dias! #3";

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

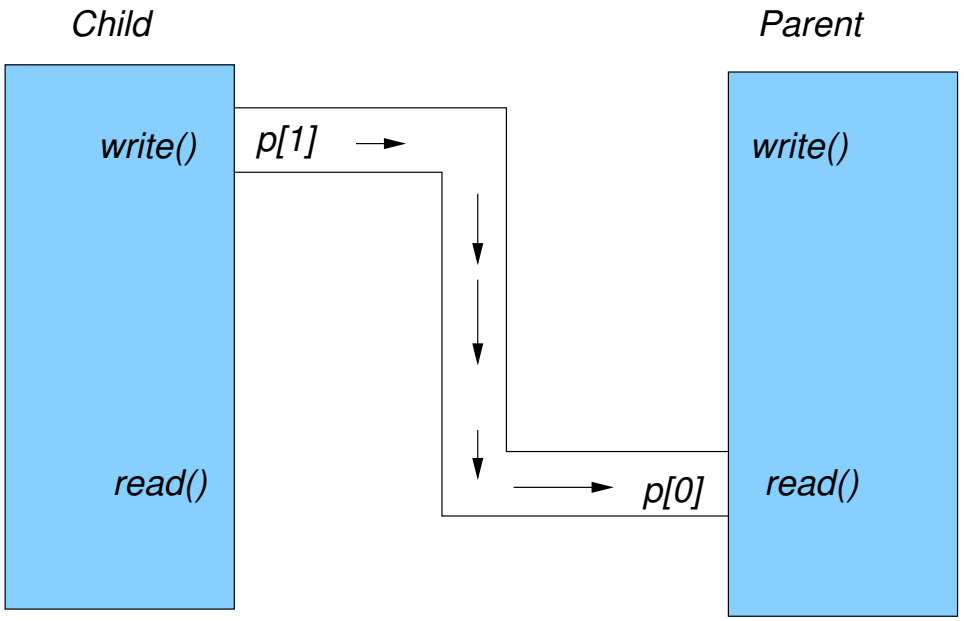
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGSIZE 16

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:



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; this is a test";

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@haiku:~/src-set005$ ./pipes-example-p50
Read 36 bytes: This is a test only; this is a test
ad@haiku:~/src-set005$
```

- Anytime, *read/write-ends* are not 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.

Example with pipe and dup2

```
#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 ( argc!= 3) {printf("./a.out prog1 prg2"); exit(23);}

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);
        perror("execlp");
        }
else    {           // child and reader
        close(fd[WRITE]);
        dup2(fd[READ],0);
        close(fd[READ]);
        execlp(argv[2],argv[2],NULL);
        }
}
```

Some outcomes:

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

The *size* program

```
#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);
```

The *size* program

```
sigaction(SIGALRM, &act, NULL);
while (1) {
       alarm(10);                  /* set alarm                          */
       write(p[1], &c, 1);         /* do the writing of the character */
       alarm(0);                   /* reset alarm                        */
       if (++count % 4096 == 0 )/* report every 4kbytes written out*/
          printf("%d characters in pipe\n",count);
      }
}

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

Outcome of execution:

```
ad@haiku:~/src-set005$ ./pipe-size
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
Alrm-Handler: write blocked after 65536 characters
ad@haiku:~/src-set005$
```

- A `write` on a pipe will execute *atomically* – all data transfered in a single kernel operation.
- Otherwise, writing occurs in stages.
- If multiple processes write to the same pipe, data will inevitably become mingled.

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").

demo-file-pipes-exec creates a child & 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");
    }
}
```

write-portion 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 the last two programs...

Execution with no parameter:

```
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$ ./demo-file-pipes-exec
filedesc1=3
fd1[READ]=4, fd1[WRITE]=5,
fd2[READ]=6, fd2[WRITE]=7
ad@haiku:~/src-set005$
```

Execution with parameter:

```
ad@haiku:~/src-set005$ ./demo-file-pipes-exec ./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@haiku:~/src-set005$
```

redirecting output in an unusual way..

```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);
        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@haiku:~/src-set005$
ad@haiku:~/src-set005$ ./demo-trick
About to run who into a file (in a strange way!)
Done running who - results in file "userlist"
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$
ad@haiku:~/src-set005$ more userlist
ad         :0           Apr  2 15:43 (:0)
ad         pts/8        Apr  3 15:34 (:0.0)
ad         pts/0        Apr  3 01:54 (:0.0)
ad         pts/6        Apr  3 10:08 (:0.0)
ad         pts/9        Apr  3 15:34 (:0.0)
ad         pts/10       Apr  3 15:34 (:0.0)
ad         pts/11       Apr  3 15:34 (:0.0)
ad@haiku:~/src-set005$
```

“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 versions use `<fcntl.h>`
    $\rightarrow$ why non-blocking FIFOs??

Creation of FIFOs (System Program & API)

- 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
```

- `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);
                        }
                }
```

The server program: `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);
                }
}
```

The 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:~/Set005/src$ ./receivemessages &
[1] 2662
ad@ad-desktop:~/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:~/Set005/src$ ./sendmessages Thessaloniki+Komotini+Xanthi+Kavala
    kkkkkkkkkkkkkkkkkkkkkkkkk....kkkkkkkkkk
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
Message with Prefix kkkkkkkkkk Too long - Ignored
```