Topic 4: Processes*

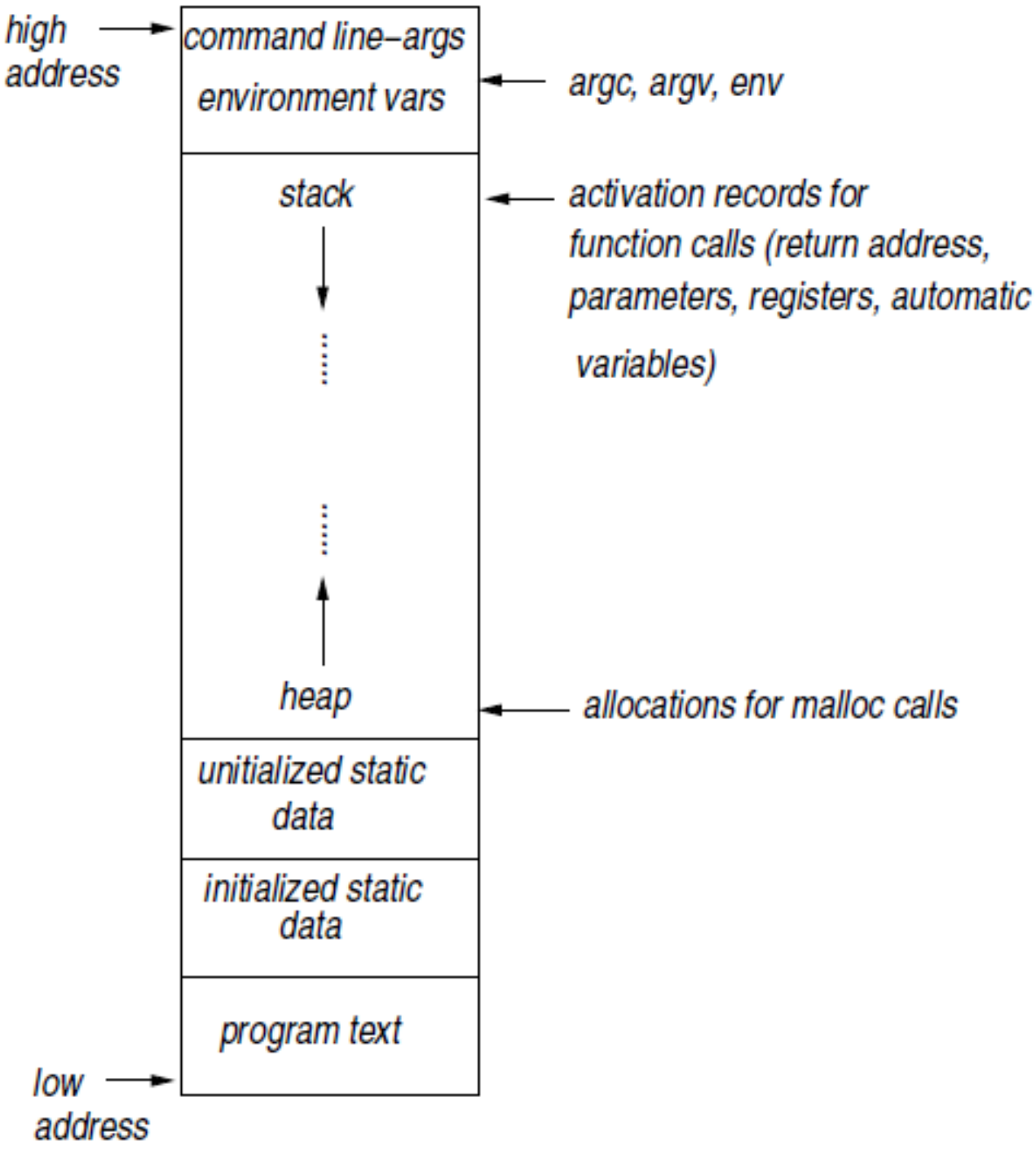
A Unix Process

- 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
  - Initialized and unitialized data
  - Run-time data
  - Run-time stack

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 with other processes* through a number of (IPC) mechanisms including pipes, shared memory, shared memory, sockets, streams, etc.

Process Instance



Processes

- Each Unix process has its own identifier (PID), its code (text), data, stack and a few other features.
- 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.

Όρια διεργασιών

- Συναρτήσεις getrlimit, setrlimit

```
struct rlimit {
    rlimit_t rlim_cur;   //soft limit = current limit
    rlimit_t rlim_max;  //hard limit = max value
                        // for current limit
}

< sys/time.h>
< sys/resource.h>

int getrlimit(int resource, struct rlimit *reslimp);
    Επιστρέφει 0, αν OK, ενώ σε σφάλμα επιστρέφει
    <> 0
int setrlimit(int resource, const struct rlimit
    *reslimp);
    Επιστρέφει 0, αν OK, ενώ σε σφάλμα επιστρέφει
    <> 0
```

Όρια διεργασιών…

- A regular user process can set the current limit with a value up to the max value
- A regular user process can decrease (but not increase) the value of the current limit of a resource
- A superuser process can set both limits (current and max) as long as they are supported by the kernel
- 7 resource limits: RLIMIT_CORE, RLIMIT_CPU, RLIMIT_DATA, RLIMIT_FSIZE, RLIMIT_NOFILE, RLIMIT_STACK, RLIMIT_AS

Πρόγραμμα limits

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

int main(){
struct rlimit myrlimit;

// RLIMIT_AS: maximum size of process's AS 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 data segment 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);
```

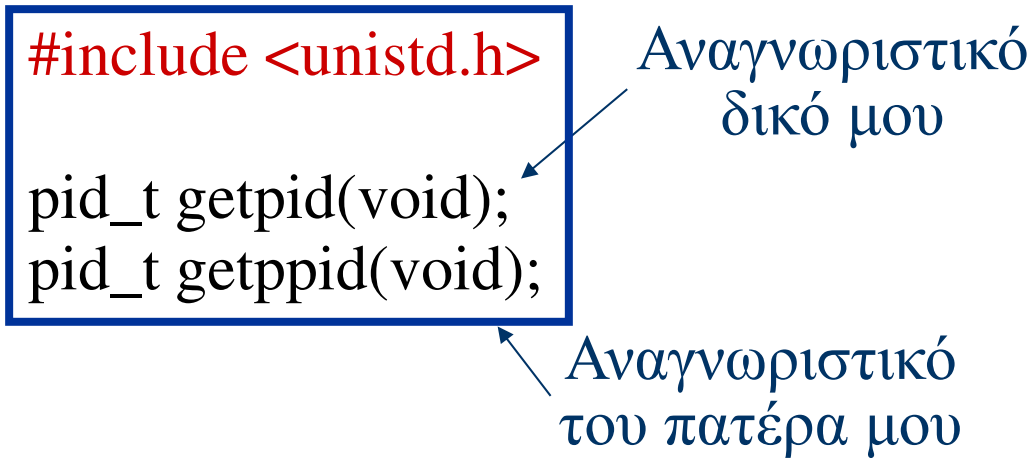
Πρόγραμμα limits

```
// 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);
return 1;
}
```

Εκτέλεση limits

```
mema@bowser> gcc -o limit limit.c
mema@bowser> ./limit
Maximum address space = 4294967295 and current = 2147483647
Maximum core file size = 4294967295 and current = 0
Maximum data+heap size = 536870912 and current = 536870912
Maximum file size = 4294967295 and current = 2147483647
Maximum stack size = 67108864 and current = 8388608
mema@bowser>
```

Process IDs




```
#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;
}
```

Process IDs (output of ./pid)

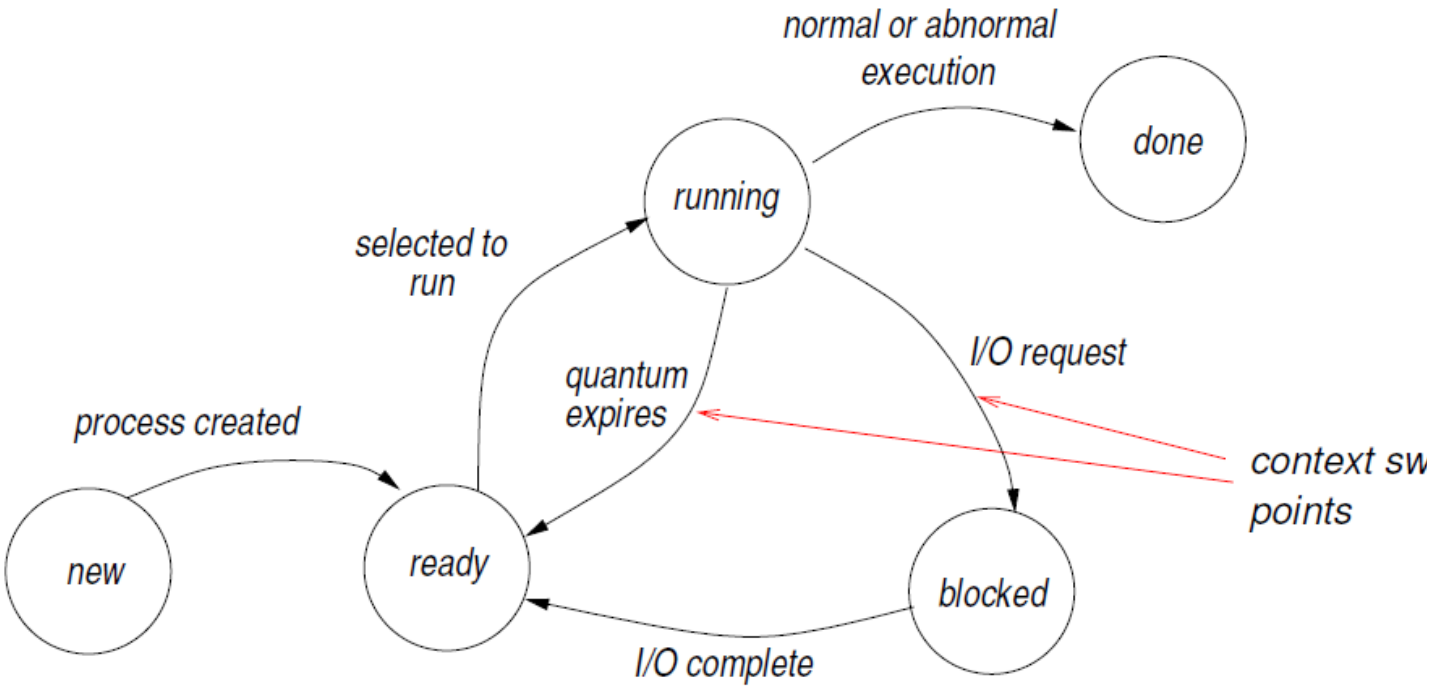
```
mema@bowser> ./pid
Process has as ID the number:
37467
Parent of the Process has as ID:
37407
mema@bowser>
```

Q: Το τρέξαμε αυτό το πρόγραμμα από τη γραμμή εντολής. Ποια είναι η διεργασία-γονέας;

Process IDs (cont'd)

```
mema@bowser> ps auxw |grep mema
273:root    37404  0.0  0.1  6544  2956  ??  Is    2:33AM   0:00.05
sshd: mema [priv] (sshd)
274:mema    37406  0.0  0.1  6544  2972  ??  S     2:33AM   0:00.(
sshd: mema@ttypd (sshd)
437:mema    37407  0.0  0.1  4324  2876  pd  Ss
2:34AM   0:00.05 -tcsh (tcsh)
438:mema    37504  0.0  0.1  1864  1200  pd  R+    2:36AM
0:00.01 ps auxw
439:mema    37505  0.0  0.0   436   280  pd  D+    2:36AM   0:00.(
grep -ni mema
440:mema     9847  0.0  0.2  8200  5088  pf- D    Thu02PM
0:00.44 pine
mema@bowser>
```

Process state diagram



The exit() call

#include <stdlib.h>
void exit(int status)

- Terminates the execution of a process and sets a **status** which is available to the parent process
- ***status*** is the exit status of the child
- When ***status*** is 0, it shows successful exit
- **Status** is also available at the shell variable $?

Κλήση συστήματος exit

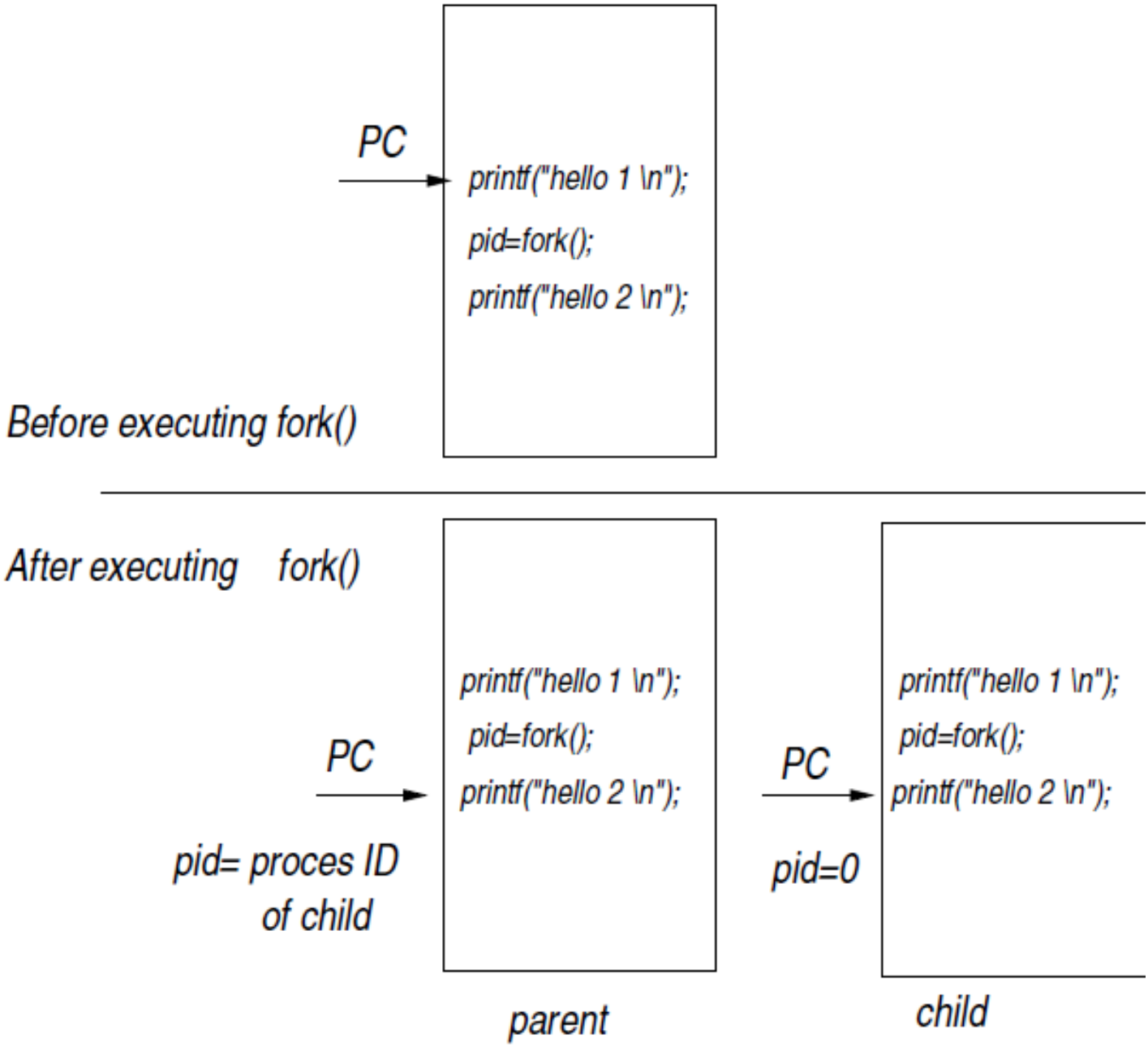
```
#include <stdio.h>
#include <stdlib.h>

#define EXITCODE 157

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

```
mema@bowser> ./exit
Going to terminate with status code 157
mema@bowser> echo $?
157
mema@bowser>
```

Process creation via fork()

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

Where the PCs are after fork()



Παράδειγμα fork

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

int main(){
  pid_t  childpid;

  childpid = fork();
  if (childpid == -1){
     perror("Failed to fork");
     return 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;
}
```

Στο παιδί επιστρέφει 0, στον πατέρα το ID του παιδιού

Παράδειγμα fork

```
mema@bowser> ./fork
I am the child process with ID: 43033
I am the parent process with ID: 43034
mema@bowser>
```

Παράδειγμα fork (2)

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

int main(){
  pid_t  childpid;
  pid_t  mypid;

  mypid = getpid();
  childpid = fork();

  if (childpid == -1){
     perror("Failed to fork");
     return 1;
     }

  if (childpid == 0)
     printf("I am the child process with ID: %lu –
        %lu\n", (long)getpid(), (long)mypid);
  else
     printf("I am the parent process with ID: %lu –
        %lu\n", (long)getpid(), (long)mypid);
  return 0;
}
```

ID του πατέρα. Το mypid αντιγράφεται στο παιδί μετά το fork

getpid() <> mypid

getpid() == mypid

Παράδειγμα fork (2)

```
mema@bowser> ./fork2
I am the child process with ID: 93545 -- 93544
I am the parent process with ID: 93544 -- 93544
mema@bowser>
```

Αλυσίδα διεργασιών


```
#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())
    break;

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

Μόνο το παιδί κάθε fork συνεχίζει το loop

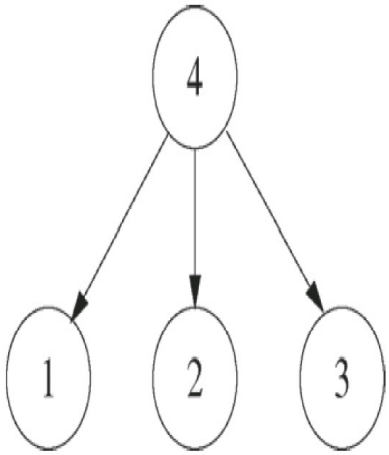
Δε θα τυπωθούν στη σειρά. Εξαρτάται από δρομολογητή διεργασιών

Output Sample(s)

```
mema@bowser> ./processChain 3
i:1 process ID:94267 parent ID:93182 child ID:94268
i:2 process ID:94268 parent ID:94267 child ID:94269
i:3 process ID:94269 parent ID:94268 child ID:0
mema@bowser>
mema@bowser> ./processChain 3
i:1 process ID:95377 parent ID:93182 child ID:95378
i:3 process ID:95379 parent ID:95378 child ID:0
i:2 process ID:95378 parent ID:95377 child ID:95379
mema@bowser>
```

Ρηχό δέντρο διεργασιών

```
#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;
}
```



Μόνο ο πατέρας κάθε fork συνεχίζει το loop

Ρηχό δέντρο διεργασιών

```
mema@bowser> ./processTree  4
i: 2 process ID: 96316 parent ID:96314 child ID:0
i: 1 process ID: 96315 parent ID:96314 child ID:0
i: 3 process ID: 96317 parent ID:96314 child ID:0
i: 4 process ID: 96314 parent ID:93182 child ID:96317
mema@bowser>
```

Orphan Processes

- If a process terminates before its child process, the child process becomes an orphan process
- Orphans become children of init process

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.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);
      }
```

Ορφανές διεργασίες

```
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());
}
```

Output

```
mema@bowser> ./orphanProcess
Original process: PID = 97939, PPID = 93182
Parent process: PID = 97939, PPID = 93182, CPID = 97940
Process with PID = 97939 terminates
mema@bowser> Child process: PID = 97940, PPID = 1
Process with PID = 97940 terminates

mema@bowser>
```

The wait() call

-- int wait(int *status)
-- Προκαλεί την αναμονή μίας διεργασίας μέχρις ότου κάποιο παιδί της τερματίσει ή αλλάξει κατάσταση
-- Αποδέχεται τον κωδικό εξόδου του παιδιού, δηλαδή τον ακέραιο της κλήσης exit, στο αριστερό byte του status, ενώ το δεξιό byte είναι 0
-- Αν το παιδί τερματίσει εξ αιτίας κάποιου σήματος, τότε τα τελευταία 7 bits του status κρατάνε το χαρακτηριστικό αριθμό του σήματος.
--Υπάρχουν μακροεντολές για να βρούμε τιμές σε αυτά τα δυο bytes του status
-- Επιστρέφει την ταυτότητα του παιδιού που τερμάτισε ή λάθος (-1) αν η διεργασία δεν έχει παιδιά

ΠΑΝΤΑ Ο ΠΑΤΕΡΑΣ ΝΑ ΚΑΛΕΙ ΤΗΝ WAIT ΓΙΑ ΚΑΘΕ ΠΑΙΔΙ
Αποδεσμεύονται πόροι διεργασίας

Η init περιοδικά αποδεσμεύει χώρο διεργασιών (zombie) που δεν τις περίμενε ο πατέρας τους

The status flag

- The int 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 returns 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 used if WIFSIGNALED returns true.
- WCOREDUMP(status): returns true if the child produced a core dump.
- WIFSTOPPED(status): returns true if the child process has not terminated, but has stopped and can be restarted
- WSTOPSIG(status): returns the number of the signal which caused the child to stop. This macro should only be used if WIFSTOPPED returns true.

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");
          exit(1);
  }
  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);
     }
```

Use of wait

```
if ((pid= wait(&status)) == -1 ){
   perror("wait failed");
   exit(2);
   }

if ( WIFEXITED(status))  {
   exit_status = WEXITSTATUS(status);
   printf("exit status from %d was %d\n",pid,
            exit_status);
   }
exit(0);
}
```

WIFEXITED(status) returns true if child process terminated normally (by calling exit).
WEXITSTATUS(status) returns input to exit call made by child.

```
mema@bowser> ./wait
Hello I am in parent process 44746 with child 44747
exit status from 44747 was 5
mema@bowser>
```

The waitpid() call

#include <sys/wait.h>

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

- pid may take various values:
  1. < -1: wait for any child whose process groupID = |pid|
  2. -1: wait for any child (εδώ, ισοδύναμη με wait())
  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.
- status is same as in wait()
- options is a disjunction of macros that indicate the *ongoing* 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 signal)
- waitpid() returns
  - The process ID of the child whose state just *changed*, if all goes well
  - If WNOHANG was specified and one or more child(ren) specified by pid exist, but *have not yet* changed state, then 0 is returned
  - -1 on error

Use of waitpid()

```
#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 is sleeping... \n",getpid());
        sleep(5);
        exit(57);
        }

   printf("reaching the father %lu process \n",
            getpid());
   printf("My child's PID is %lu \n", pid);
```

Use of waitpid()

```
while( (waitpid(pid, &status, WNOHANG)) == 0 ){
        printf("Still waiting for child to return\n");
        sleep(1);
        }

   printf("reaching the father %lu process \n",
            getpid());

   if (WIFEXITED(status)){
       exit_status = WEXITSTATUS(status);
       printf("Exit status from %lu was %d\n",
                  pid, exit_status);
       }
   exit(0);
}
```

Output

```
mema@bowser> ./waitpid
reaching the father 50485 process
My child's PID is 50486
Still waiting for child to return
Child 50486 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 50485 process
Exit status from 50486 was 57
mema@bowser>
```

Use of wait() and >>

```
#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;

  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);
      }
```

Use of wait() and >>

```
  else {
      printf("Child process: PID = %d,
                PPID = %d \n", getpid(), getppid());
      exit(62);
   }
   printf("Process with PID = %d terminates",
            getpid());
   sleep(1);
}
```

Output

```
mema@bowser> ./wait2
Original Process: PID = 51182
Parent process: PID = 51182
Child process: PID = 51183, PPID = 51182
Child terminated: PID = 51183, exit code = 62
Process with PID = 51182 terminates.
mema@bowser>
```

Zombie Processes

A process that terminates remains in the system until its parent *receives* its exit code (via a wait()) call. Until then, 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){      /* Never terminates */
            sleep(500);
       }
   }
   else {
      exit(37);
   }
}
```

```
mema@bowser>  ./zombie &
[1] 17508
mema@bowser> ps -a
PID    TTY       TIME        CMD
5822   pts/5     00:00:00    ssh
7508   pts/3     00:15:02    man
13772  pts/0     00:00:00    ssh
17508  pts/6     00:20:17    zombie
17509  pts/6     00:20:17    zombie <defunct >
17510  pts/6     00:20:30    ps
mema@bowser> kill -9 17508
[1]+ Killed      ./zombie
mema@bowser> ps –a
PID    TTY      TIME         CMD
5822   pts/5     00:00:00    ssh
7508   pts/3     00:15:02    man
13772  pts/0     00:20:30    ssh
17512  pts/6     00:00:00    ps
mema@bowser>
```

Παράδειγμα

Δημιουργήστε ένα πλήρες δυαδικό δένδρο από διεργασίες με βάθος Ν. Για κάθε διεργασία που δεν είναι φύλλο στο δένδρο εκτυπώστε την ταυτότητα της, την ταυτότητα του πατέρα της και έναν αύξοντα αριθμό σύμφωνα με μία κατά πλάτος διάσχιση του δένδρου.

```
#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;          /* Holds the number of each
                          process */
   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:                              /* Left child code */
        numb=2*numb; break;
```

```
case -1:                    /* Error creating left child *
        perror("fork"); exit(1);
default:                    /* Parent code */
        switch (pid2=fork()){
          case 0:           /* Right child code */
            numb=2*numb+1; break;
          case -1:          /* Error creating right child
            perror("fork"); exit(1);
          default:          /* Parent code */
            wait(&status); wait(&status);
            exit(0);
          }
    }
  }
}
```

Μια διεργασία σε βάθος k θα δημιουργήσει παιδία αν k<depth και θα εμφανίσει το pid του και το ppid του γονέα της.

Αποτέλεσμα

```
mema@bowser> ./binaryTree 1
I am process no 1  with PID  8970 and PPID 7968
mema@bowser> ./binaryTree 2
I am process no 1  with PID  8981 and PPID 7968
I am process no 2  with PID  8982 and PPID 8981
I am process no 3  with PID  8984 and PPID 8981
mema@bowser> ./binaryTree 3
I am process no 1  with PID  8988 and PPID 7968
I am process no 2  with PID  8989 and PPID 8988
I am process no 4  with PID  8990 and PPID 8989
I am process no 5  with PID  8992 and PPID 8989
I am process no 3  with PID  8995 and PPID 8988
I am process no 6  with PID  8996 and PPID 8995
I am process no 7  with PID  9000 and PPID 8995
mema@bowser>
```

Αποτέλεσμα

```
mema@bowser> ./binaryTree 4
I am process no 1  with PID  9003 and PPID 7968
I am process no 2  with PID  9004 and PPID 9003
I am process no 4  with PID  9005 and PPID 9004
I am process no 8  with PID  9006 and PPID 9005
I am process no 3  with PID  9008 and PPID 9003
I am process no 6  with PID  9009 and PPID 9008
I am process no 12  with PID  9010 and PPID 900
I am process no 9  with PID  9013 and PPID 9005
I am process no 5  with PID  9015 and PPID 9004
I am process no 13  with PID  9016 and PPID 900
I am process no 10  with PID  9017 and PPID 901
I am process no 7  with PID  9019 and PPID 9008
I am process no 14  with PID  9021 and PPID 901
I am process no 11  with PID  9026 and PPID 901
I am process no 15  with PID  9030 and PPID 901
mema@bowser>
```

The exec 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[]) ;

- They all replace the calling process (including text, data, bss, stack) with the executable designated by either the path or file.
- On success, they do not return, on error -1 is returned, and errno is set appropriately.
- These calls, collectively known as the exec calls, are a front-end to execve.

```
int execl(char *path, char *arg0, char *arg1,
            …, char *argn, NULL);
int execv(char *path, char *argv[]);
int execlp (char *file, char *arg0, char *arg1,
            …, char *argn, NULL);
int execvp(char *file, char *argv[]);
int execle(char *path, char *arg0, char *arg1,
              …, char *argn, NULL,  char *envp[]) ;
```

- execl, execv, execle 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, 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 an array: 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 an array: envp[0], envp[1], envp[2],..,envp[n] with NULL as delimeter in envp[n+1].

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 with the designated “environment” variables to be 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.

```
#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 ");
}
```

```
mema@bowser>   ./exec
I am process 134513516 and I will execute an 'ls -l .;
total 64
-rwxr -xr-x 1 mema mema 8413 2010-04-19 23:56 a.out
-rw-r--r-- 1 mema mema 233 2010-04-19 23:56 exec-demo .c
-rwx ------ 1 mema mema 402 2010-04-19 00:42 fork1 .c
-rwx ------ 1 mema mema 529 2010-04-19 00:59 fork2 .c
mema@bowser>
```

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);
      }
```

Παράδειγμα execvp

```
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);
        }
}
```

Output

```
mema@bowser> gcc -o execvp execvp.c
mema@bowser> ./execvp
I am the parent process 35796
date
I am the child process 35797 and will be replaced
with 'date'
Sun Jan 17 01:21:55 EST 2010
Child terminated with exit code 0
mema@bowser>
```

Pipes

- Sharing files is a way for various processes to communicate among themselves (but this entails a number of problems)
- Pipes are one Unix mechanisms for IPC that provides one-way communication between two processes (often parent and child)
- 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 the help of a `read` call.

Pipes

- #include<unistd.h>

  int pipe(int pipefd[2]);

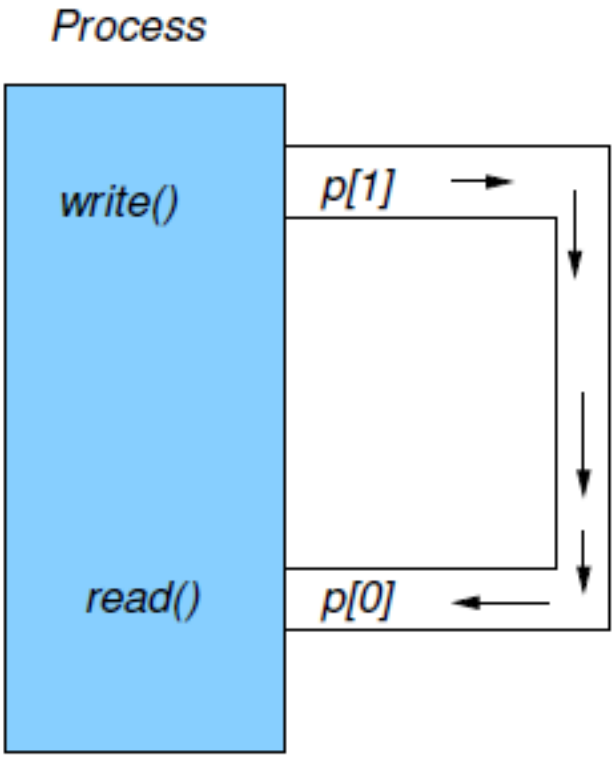
- Creates a unidirectional data channel that can be used for interprocess communication in which pipefd[0] refers to the read end of the pipe and pipefd[1] to the write end
- Returns 0 on success, -1 on erros (errno is set appropriately)
- A pipe's real value appears when it is used in conjunction with 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

```
mema@bowser> ./pipe-example0
Buenos Dias!  #1
Buenos Dias!  #2
Buenos Dias!  #3
mema@bowser>
```

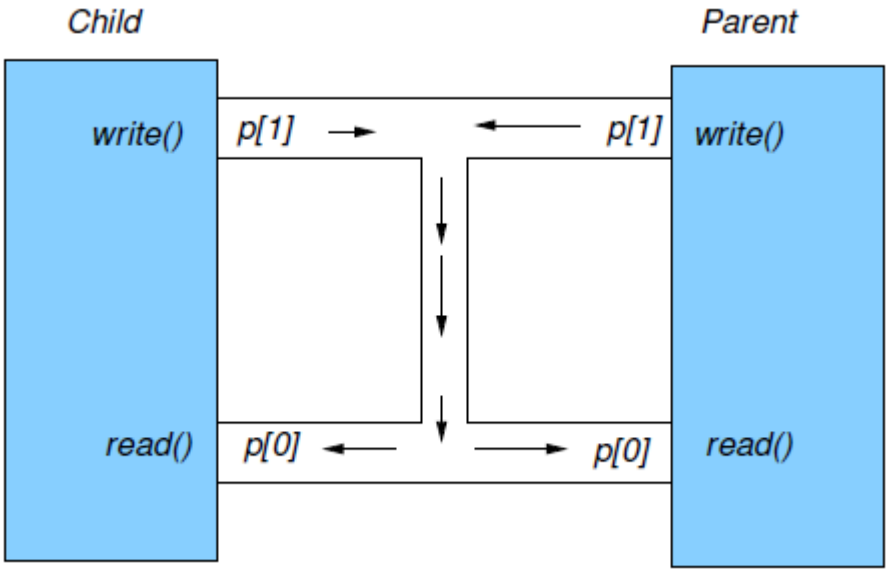


A *somewhat* more useful example

```
#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

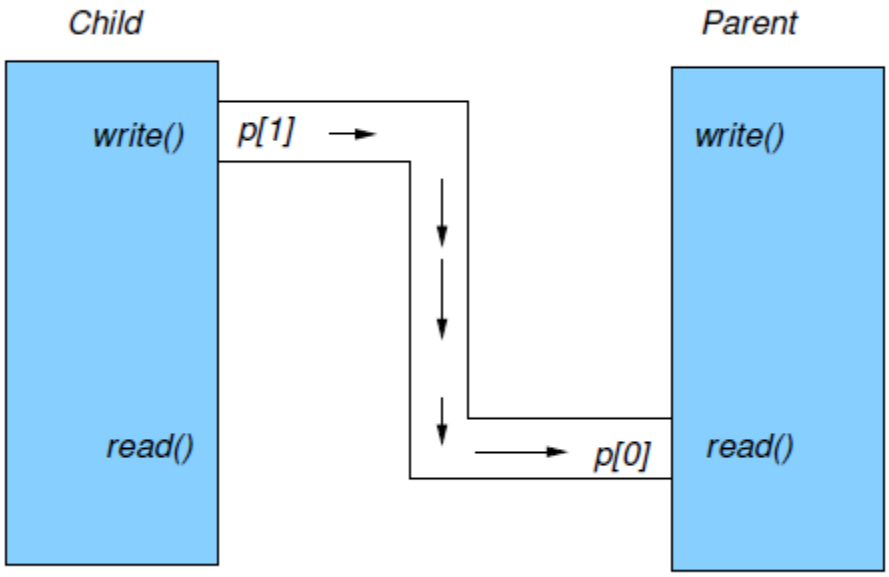
A much cleaner version

```
#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;
```

A much cleaner version

```
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

```
#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]);
     }
```

Να κλείνετε τα άκρα που δεν χρησιμοποιείτε

Να κλείνετε τα άκρα που δεν χρησιμοποιείτε

```
else{                   // parent
    close(fd[WRITE]);
    bytes=read(fd[READ], message,
                    sizeof(message));
    printf("Read %d bytes: %s \n",bytes, message)
    close(fd[READ]);
  }
}
```

```
mema@bowser> ./pipeReadWrite
Read 20 bytes: This is a test only
mema@bowser>
```

Another example

```
#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";
   pid_t  childpid;
   int fd[2];

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

   childpid = fork();

   if (childpid == -1) { perror("failed to fork");
                         exit (23);}

   if (childpid)  // parent code
       close(fd[0]);
       write(fd[1],bufout,strlen(bufout)+1);
```

Γράψιμο

Διάβασμα

```
else            // child code
        close(fd[1]);
        read(fd[0],bufin,strlen(bufin)+1);

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

Output

```
mema@bowser> ./pipe
[9221]: my bufin is hello, my bufout is hello
(parent process 9220)
[9220]: my bufin is empty, my bufout is hello
(parent process 7968)
mema@bowser>
```

Read() call and pipes

- If pipe has data, returns immediately
  - Data are read FIFO
  - Data can be read only ONCE
- If pipe is empty, and another process has the pipe open for writing, then read() blocks.
- If pipe is empty and no process has the pipe open for writing, then read() returns 0.

Example: implement the functionality of "|" (command-line Unix pipe)

```
#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);
     perror("execlp");
     }
```

Οτι γράφει το argv[1] γράφεται στο σωλήνα

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

Ότι διαβάζεται από το argv[2] διαβάζεται από το σωλήνα

Τα προγράμματα διαβάζουν/γράφουν από/στη προκαθορισμένη είσοδο/έξοδο χωρίς να ξέρουν ότι έχει γίνει ανακατεύθυνση μέσω σωλήνα.

Execution output

```
mema@bowser> ./pipeline ls wc
      32      32     278
mema@bowser> ./pipeline ls head
binaryTree
binaryTree.c
execvp
execvp.c
exit
exit.c
exit2
exit2.c
fork
fork.c
mema@bowser> ./pipeline ps sort
  PID  TT  STAT      TIME COMMAND
 9847  pf- D      0:01.43 pine
73155  p5  Ss     0:00.11 -tcsh (tcsh)
73323  p5  R+     0:00.00 ps
73324  p5  S+     0:00.00 sort
mema@bowser> ./pipeline who wc
12      72     634
```

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 available again.

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 the actual upper-bounds for this size in a system.

```
#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);
```

```
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);
}
```

```
mema@bowser> ./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
write blocked after 65536 characters
mema@bowser>
```

What happens to file descriptors after an exec?

- For each of the calls execl, execv, execlp, execvp:
- All file descriptors remain open after an exec
  - 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")
  - See example: demo-file-pipes-exec και write-portion.c

Main program that creates a child and calls execlp
(called demo-file-pipes-exec)

```
#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 *(called write-portion)*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.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, it 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;
}
```

```
mema@bowser> ./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
mema@bowser>
```

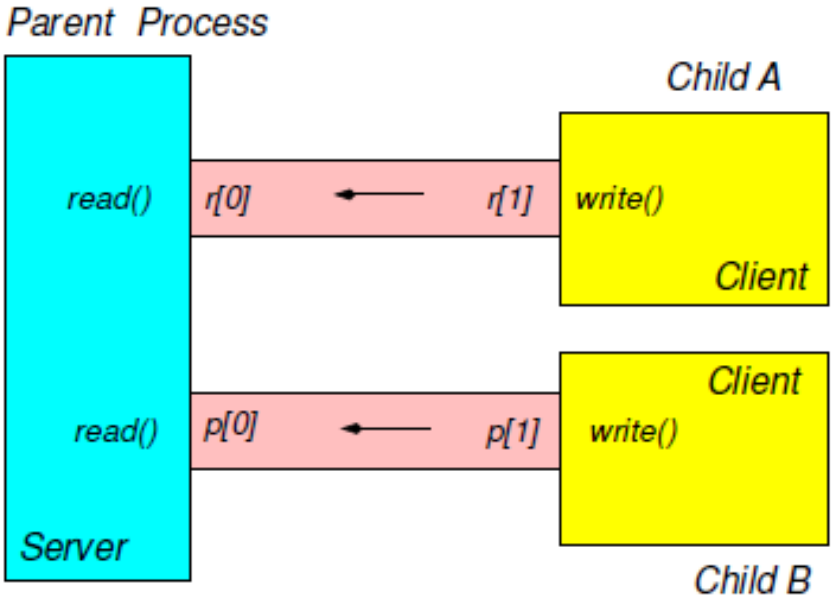
Example: redirecting stdout

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

Μικρότερος διαθέσιμος περιγραφέας είναι το 1!!!

```
mema@bowser> ./demo-trick
About to run who into a file (in a strange way!)
Done running who - results in file "userlist"
mema@bowser> more userlist
mema       pts/2        Mar 27 16:07   (linux03.di.uoa.gr)
```

Reading from *Multiple Pipes*



- Server should be able to deal with the situation where there is data present on more than one pipes.
- If nothing is pending, the server should block.
- If data arrives in one of more pipe, server has to become aware of where the information is to receive it.

```
# include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

“Limitations” of Pipes

- Classic pipes have at least two drawbacks:
  - Can only be used between processes that have an ancestor in common
  - Classic pipes are NOT permanent (persistent)

FIFO (Named Pipes)

- Named pipes (or FIFOs)
  - Can be used by two totally unrelated processes to communicate
  - Works like a pipe (one-way data flow)
  - Are permanent in the file system
  - Looks like a file, has a name; has an *owner* and *access* permissions
  - Any process with appropriate rights can open for reading or writing

    (using *open, read, write, close* syscalls)
  - **Cannot be** seeked
  - Default: on open for reading and no writer, blocks for an open for writing from another process (and vice versa)
  - Blocking and non-blocking versions use <fcntl.h>
    - Why non-blocking FIFOs?

Creating a FIFO (at command line)

- System program: mknod nameofpipe p
  - nameofpipe is the name you give to the FIFO
  - Note the p parameter above and the p in the permissions listing below:

```
mema@bowser> mknod myfifo p
mema@bowser>   ls -l  myfifo
prw-r--r-- 1 mema mema 0 2018-04-15 15:37 myfifo
mema@bowser>
```

  - The command “mkfifo” has same effect, type “man mkfifo” for details

Creating a FIFO (programmatically)

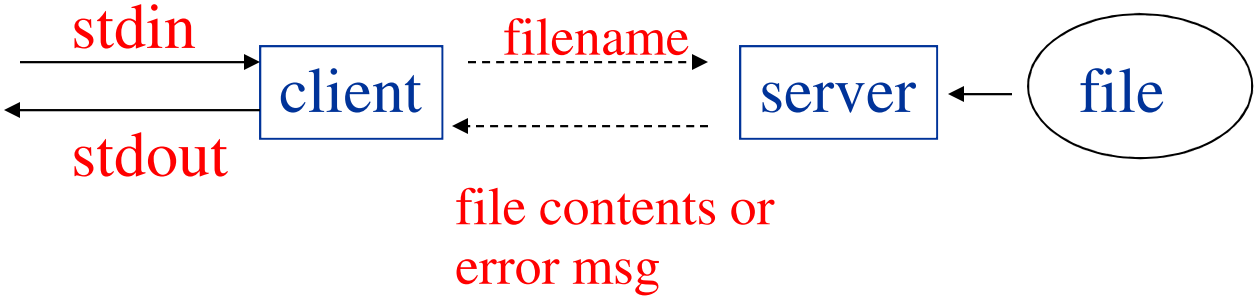
- int mkfifo(const char *path, int perms);
  - Creates a named pipe: *path* is where the FIFO is created on the filesystem
  - *perms* designates the access permissions for the owner, group, others
  - Returns 0 on success, -1 on failure (sets errno accordingly)
  - Included files:

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

Client-server example

- Client reads filename from standard input and writes it to IPC channel
- Server reads filename from IPC channel, tries to open file for reading and on success, reads and writes file to IPC channel;

  on failure, sends error msg to IPC
- Client reads from IPC channel and writes bytes to standard output



namedPipes-client.c

```
#include  <sys/types.h>
#include  <sys/stat.h>
#include <sys/errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern int errno;

#define MAXBUFF 1024
#define FIFO1   "/tmp/fifo.1"
#define FIFO2   "/tmp/fifo.2"
#define PERMS   0666
main(){
   int readfd, writefd;

   /* Open the FIFOs.  We assume server has
already created them.  */

   if ( (writefd = open(FIFO1, O_WRONLY))  < 0)
{
      perror("client: can't open write fifo \n");
   }
   if ( (readfd = open(FIFO2, O_RDONLY))  < 0)
{
      perror("client: can't open read fifo \n");
   }

   client(readfd, writefd);
```

```
    close(readfd);
    close(writefd);

    /* Delete the FIFOs, now that we're done.  */

    if ( unlink(FIFO1) < 0) {
     perror("client: can't unlink \n");
   }
    if ( unlink(FIFO2) < 0) {
     perror("client: can't unlink \n");
    }

   exit(0);
}

client(int readfd, int writefd) {

    char buff[MAXBUFF];
    int n;

    /* Read the filename from standard input,
     *  write it to the IPC descriptor.
     */

    if (fgets(buff, MAXBUFF, stdin) == NULL)
     perror("client: filename read error \n");
```

```
    n = strlen(buff);
    if (buff[n-1] == '\n')
        n--;     /* ignore newline from fgets() */

    if (write(writefd, buff, n) != n)
        perror("client: filename write error");

    /* Read data from the IPC descriptor and write to
     * standard output.
     */

  while ( (n = read(readfd, buff, MAXBUFF)) > 0)
        if (write(1, buff, n) != n)   /* fd 1 = stdout */ {
            perror("client: data write error \n");
        }
    if (n <0) {
        perror("client: data read error \n");
    }

}
```

namedPipes-server.c

```
#include  <sys/types.h>
#include  <sys/stat.h>
#include <sys/errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern int errno;

#define MAXBUFF 1024
#define FIFO1   "/tmp/fifo.1"
#define FIFO2   "/tmp/fifo.2"
#define PERMS   0666
main(){
   int readfd, writefd;

   /* Create the FIFOs, then open them -- one for
    * reading and one for writing.
    */

   if ( (mkfifo(FIFO1, PERMS) < 0) &&
         (errno !=   EEXIST) ) {
              perror("can't create fifo");
   }
   if ((mkfifo(FIFO2, PERMS) < 0) &&
        (errno != EEXIST)) {
            unlink(FIFO1);
            perror("can't create fifo");
   }
```

```
    if ( (readfd = open(FIFO1, O_RDONLY))  < 0)  {
        perror("server: can't open read fifo");
    }

if ( (writefd = open(FIFO2, O_WRONLY))  < 0)  {
        perror("server: can't open write fifo");
    }

    server(readfd, writefd);
    close(readfd);
    close(writefd);
    exit(0);
}

server(int readfd, int writefd) {

    char buff[MAXBUFF];
    char errmesg[256];
    int n, fd;

    /* Read the filename from the IPC descriptor. */
    if ((n= read(readfd, buff, MAXBUFF)) <= 0) {
        perror("server: filename read error ");
    }

    buff[n] = '\0';  /* null terminate filename */
```

```
  if ( (fd = open(buff, 0)) <0) {
    /* Error. Format an error message and send it
     *  back to the client
     */

    sprintf(errmesg, ":can't open, %s\n", buff);
    strcat(buff, errmesg);
    n = strlen(buff);
    if (write(writefd, buff, n) != n)  {
       perror("server: errmesg write error");
    }
  } else {
   /*
    * Read the data from the file and write to
    * the IPC descriptor.
    */

    while ( (n = read(fd, buff, MAXBUFF)) > 0)
         if (write(writefd, buff, n) != n) {
             perror("server: data write error");
         }
    if (n < 0) {
      perror("server: read error");
    }
  }
}
```

Execution output

```
linux01:/home/users/mema>./namedPipes-server &
[1] 4095
linux01:/home/users/mema>cat testFile.txt
See the file name/contents go
from client over named pipes
to server and back!!!  Neat!!

linux01:/home/users/mema>./namedPipes-client
testFile.txt
See the file name/contents go
from client over named pipes
to server and back!!!  Neat!!

[1]  + Done                    ./namedPipes-server
```


User types file name here