OPERATING SYSTEMS: PROCESSES

Signals, exceptions and pipes

Content

- **Signals.**
- Timers.
- Exceptions.
- Redirection and Pipes
- Environment of a Process.

Signals. Concept.

- Mechanisms to allow notifying a process of an event occurrence.
- When a process receives a signal
  - It is processed inmediately.
- Possible actions:
  - Ignore the signal, to be immune to it
  - Call the default signal processing routine
  - Call the signal processing routine defined by the process

Signals

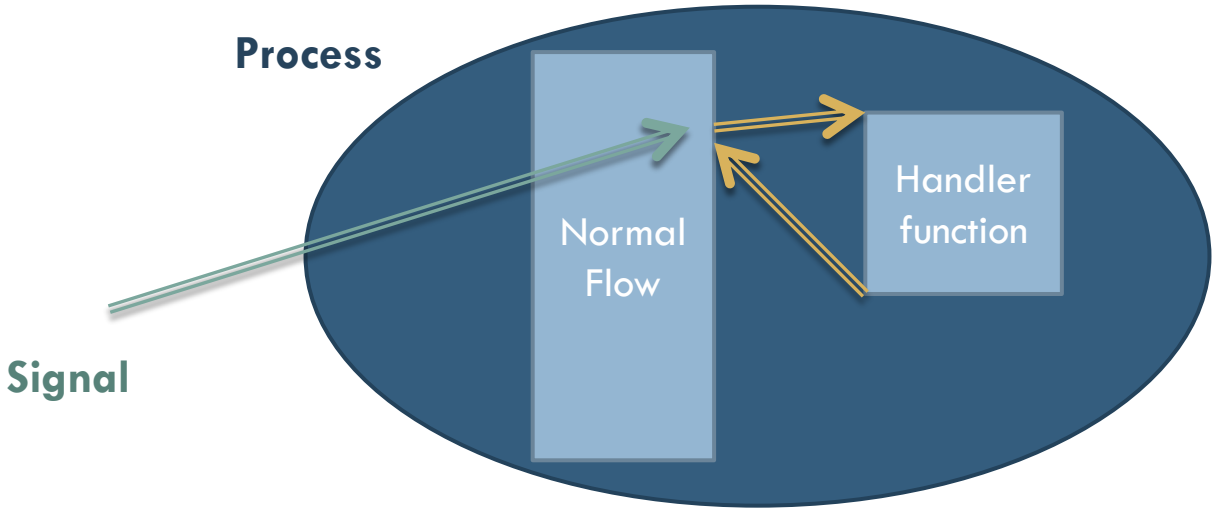
- Examples:
  - A parent process receives signal SIGCHLD when a child process finishes.
  - A process receives a signal SIGILL when it tries to execute an illegal machine instruction.

Mechanism of UNIX-like OS

Signals

- Signals interrupt to the process asynchronously.

- Send or generation
  - Process-Process (within the group) with kill.
  - OS-Process.



Signals

- OS transmits the signal to process:
  - Process must be ready to receive it.
    - Specifying a signal procedure with **sigaction**.
    - Masking the signal with **sigprogmask**
  - If it is not ready, it performs the default action:
    - Generally process dies.
    - Some signals are ignored or have another effect.
- When a process receives a signal:
  - If it is running: Stops running the current machine instruction.
  - If there is a handler routine for the signal: Branch to run the handler.
  - If the handler does not finish the process: Return to the point where the signal was received.

Predefined signals


```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1
36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4  39) SIGRTMIN+5
40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8  43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1
64) SIGRTMAX
```

POSIX services for handling signals

- `int kill(pid_t pid, int sig)`
  - Send signal sig to process **pid**.
  - Special cases:
    - **pid==0**
      - Signal to process with **gid** equal to process **gid**.
    - **pid==-1**
      - Signal to all processes (except system processes)
    - **pid <-1**
      - Signal to all processes with **gid** equal to absolute value of **pid**

- `int sigaction(int sig,`
  `struct sigaction *act,`
  `struct sigaction *oact)`
  - Permits to specify actions to the signal **sig**.
  - Old action can be stored in **oact**.

Sigaction struct definition


```
struct sigaction {
  void (*sa_handler)(); /* handlers*/
  sigset_t sa_mask; /* blocked signals */
  int sa_flags;     /* options */
};
```

- Handler:
  - **SIG_DFL**: Default action (usually terminates process).
  - **SIG_IGN**: Ignores signal.
  - Address of handler function.
- Mask of signals to block during handler.
- Options usually to zero.

Signals sets

- `int sigemptyset(sigset_t * set);`
  - Creates an empty signal set.
- `int sigfillset(sigset_t * set);`
  - Creates a full set of all possible signals.
- `int sigaddset(sigset_t * set, int signo);`
  - Adds a signal to a set of signals.
- `int sigdelset(sigset_t * set, int signo);`
  - Removes a signal from a signal set.
- `int sigismember(sigset_t * set, int signo);`
  - Checks whether a signal belongs to a signal set.
  - Checks if one signal is part of a signal set.

Example

- Ignore signal **SIGINT**
  - Produced when **Ctrl+C** keys are pressed.

```
struct sigaction act;
act.sa_handler = SIG_IGN;
act.flags = 0;
sigemptyset(&act.sa_mask);
Sigaction(SIGINT, &act, NULL);
```

POSIX services for signals

- `int pause(void)`
  - Blocks a process until signal reception.
  - Does not specify a timeout.
  - Does not allow to select type of signal awaited.
  - Does not unblock process upon ignored signals.

- `int sleep(unsigned int sec)`
  - Suspends a process until a timeout elapses or a signal is received.

Ejemplo: capture SIGSEV


```
/*Program to raise SIGSEGV signal writing in 0 memory
position. */

#include …
#include <signal.h>

void capturar_senyal(int senyal){
  printf("Error: illegal memory usage\n");
  signal(SIGSEGV,SIG_DFL);}

main(void){
  int *p;
  signal(SIGSEGV,capturar_senyal);
  printf ("Handeler set up\n");
  p=0;
  printf ("I put 5 in variable\n");
  *p=5; }
```

Content

- Signals.
- **Timers.**
- Exceptions.
- Redirection and Pipes
- Environment of a Process.

Timers

- Operating system keeps a timer per process (UNIX).
  - Kept in process **PCB** a counter with remaining time for the timer to elapse.
  - Operating system routine updates all timers.
  - If a timer reaches zero, it runs the handler function.

- In UNIX, the operating system sends a signal **SIGALRM** to process when its timer elapses.

POSIX services for timers

- `int alarm(unsigned int sec)`
  - Sets a timer.
  - If argument is zero, deactivates timer.

Example: Print message every 10 seconds


```
#include <signal.h>
#include <stdio.h>
void handle_alarm(void) {
  printf("Activated \n");
}

int main()  {
  struct sigaction act;

  /* Setis handler for SIGALRM */
  act.sa_handler = handle_alarm;
  act.sa_flags = 0;
  sigaction(SIGALRM, &act, NULL);

  act.sa_handler = SIG_IGN;             /* ignore SIGINT */
  sigaction(SIGINT, &act, NULL);
  for(;;){      /* SIGALRM every 10 secons */
    alarm(10);
    pause();
  }
}
```

Timed termination


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

pid_t pid;
void handle_alarm(void) {
  kill(pid, SIGKILL);
}

main(int argc, char **argv) {
  int status;
  char **args;
  struct sigaction act;
  args = &argv[1];
  pid = fork();
  switch(pid) {
    case -1:   /* error in fork() */
      perror ("fork");
    exit(-1);
    case 0:    /* child */
      execvp(args[0], args);
      perror("exec");
      exit(-1);
    default:   /* parent */
      /* set handler*/
      act.sa_handler = handle_alarm;
      act.sa_flags = 0;
      sigaction(SIGALRM, &act, NULL);
      alarm(5);
      wait(&status);
  }
  exit(0);
}
```

Content

- Signals.
- Timers.
- **Exceptions.**
- Redirection and Pipes
- Environment of a Process.

Excepciones

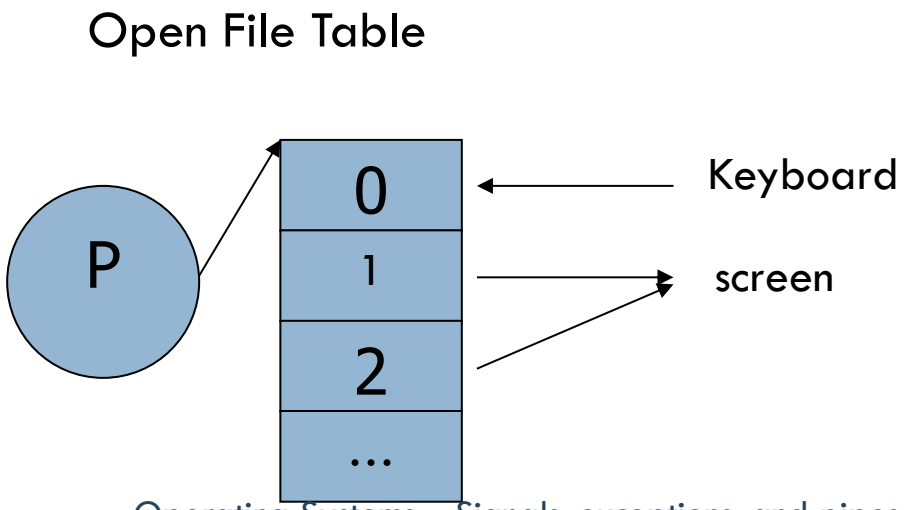
- The hardware detects special conditions:
  - Page fault, write to read-only page, stack overflows, segment violation, syscall, ..
- It transfers control to the SO for its treatment, which:
  - Save process context
  - Run routine if necessary
  - Send a signal to the process indicating the exception

Content

- Signals.
- Timers.
- Exceptions.
- **Redirection and Pipes**
- Environment of a Process.

Default File Descriptors

- Process created with three files: stdin, stdout, and stderr (file descriptors 0, 1, 2).
  - read(0, buf, 4) ; // read from stdin (initially Keyboard)
  - write(1, "Message", 10) ; // goes to stdout (initially terminal)
  - write(2, "Error", 6) ; // goes to stderr (initially terminal)

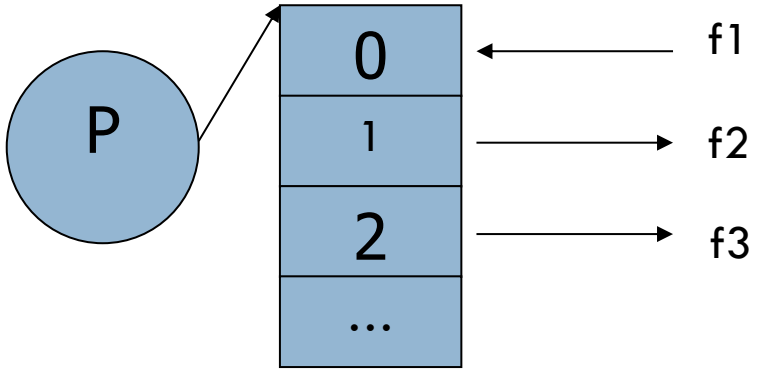


Redirecting stdin and stdout

- How to do it on command line:
  - ./ls > f2 // ls write output to file f2 rather than terminal.
  - ./wc < f1 // wc read from f1 not terminal.
- How to redirect stdind/stdout/stderr from/to files?
- Key: When Unix allocates file descriptor, it always chooses the lowest available.
  - Example: If close stdin and open new file,
    - e.g., close(0), fd = open("f1", ….)
  - then fd = 0 will be allocated and all subsequent reads from stdin will read instead from "f1".

Redirection example


```
main()
{
int fd ;
// Redirecting STDIN
close(0);      /* close stdin */
fd = open("f1",O_CREAT | O_RDONLY);  // fd=0
// Redirecting STDOUT
close(1);      /* close stdout */
fd = open("f2",O_CREAT | O_WRONLY); // fd = 1
// Redirecting STDERR
close(2);      /* close stderr */
fd = open("f3",O_CREAT | O_WRONLY); // fd = 3
}
```

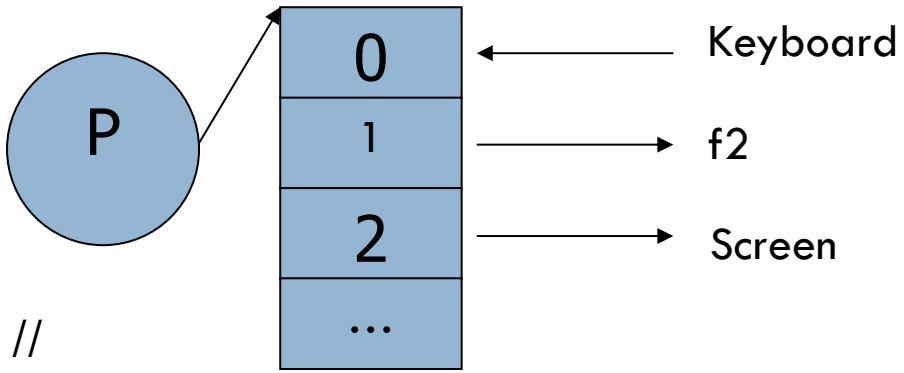


Dup System Call

- int **dupfd = dup(fd);**
  - Duplicates a file descriptor -- “aliased”
    - Both point to same file, that being the argument to dup.
    - Reads/writes from fd and dupfd going to the same file.
    - Dupfd is again the first empty free in the open file table.
  - Useful for situation such as:
    - Writing some output at the same time to standard output and a file -- all using **printf**.
    - Redirecting I/O.

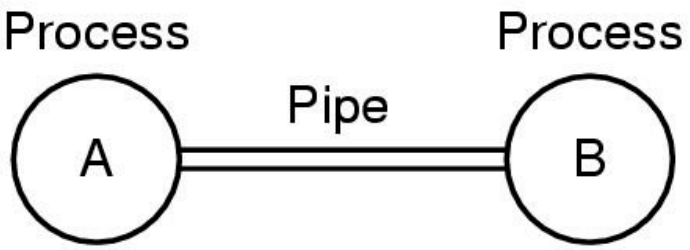
Redirection example using dup


```
main()
{
int fd, dupfd ;
// Redirecting STDOUT
fd = open("f2",O_CREAT | O_WRONLY); //
close(1);      /* close stdout */
dupfd = dup(fd);  // STDOUT = fd
close(fd);      /* close file */
printf ("prueba \n");
}
```



Unix Pipes

- Pipe sets up communication channel between two (related) processes.



Two processes connected by a pipe

- One process writes to the pipe, the other reads from the pipe.

Pipes

- A simple, **unnamed** pipe provides a one-way flow of data.
  - Can be thought as a special file that can store a limited amount of data in a first-in-first-out manner, exactly akin to a queue.
  - Parent/child processes communicating via unnamed pipe.

- Other variations:
  - Stream pipes
  - FIFOs

pipe System Call (unnamed)

- An unnamed pipe is created by calling ***pipe()***, which returns an array of 2 file descriptors (int).
  - The file descriptors are for reading and writing, respectively
- System call:

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

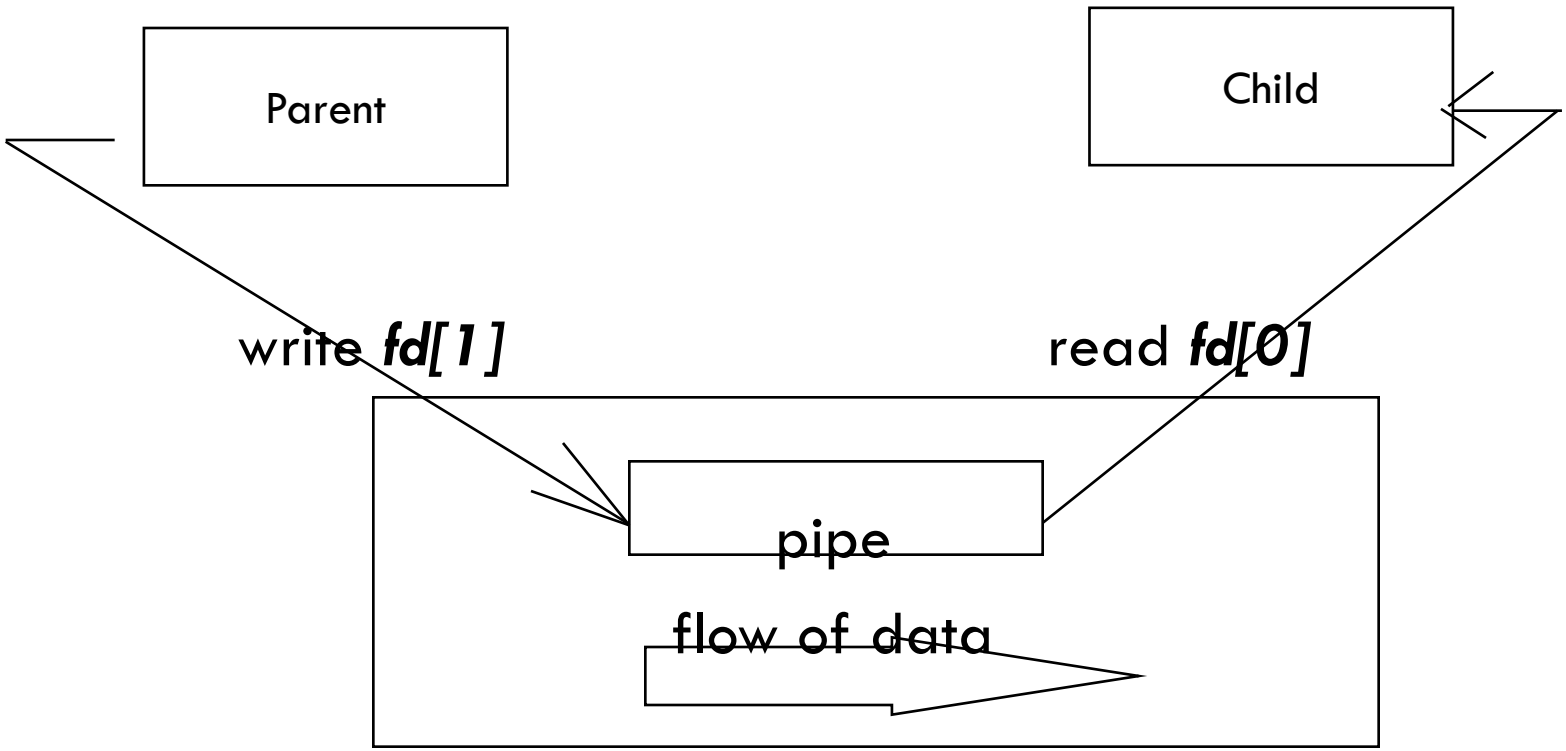
Return: Success: 0; Failure: -1;

- Looks exactly the same as reading from/to a file.

  fd[0] now holds descriptor to read from pipe

  fd[1] now holds descriptor to write into pipe

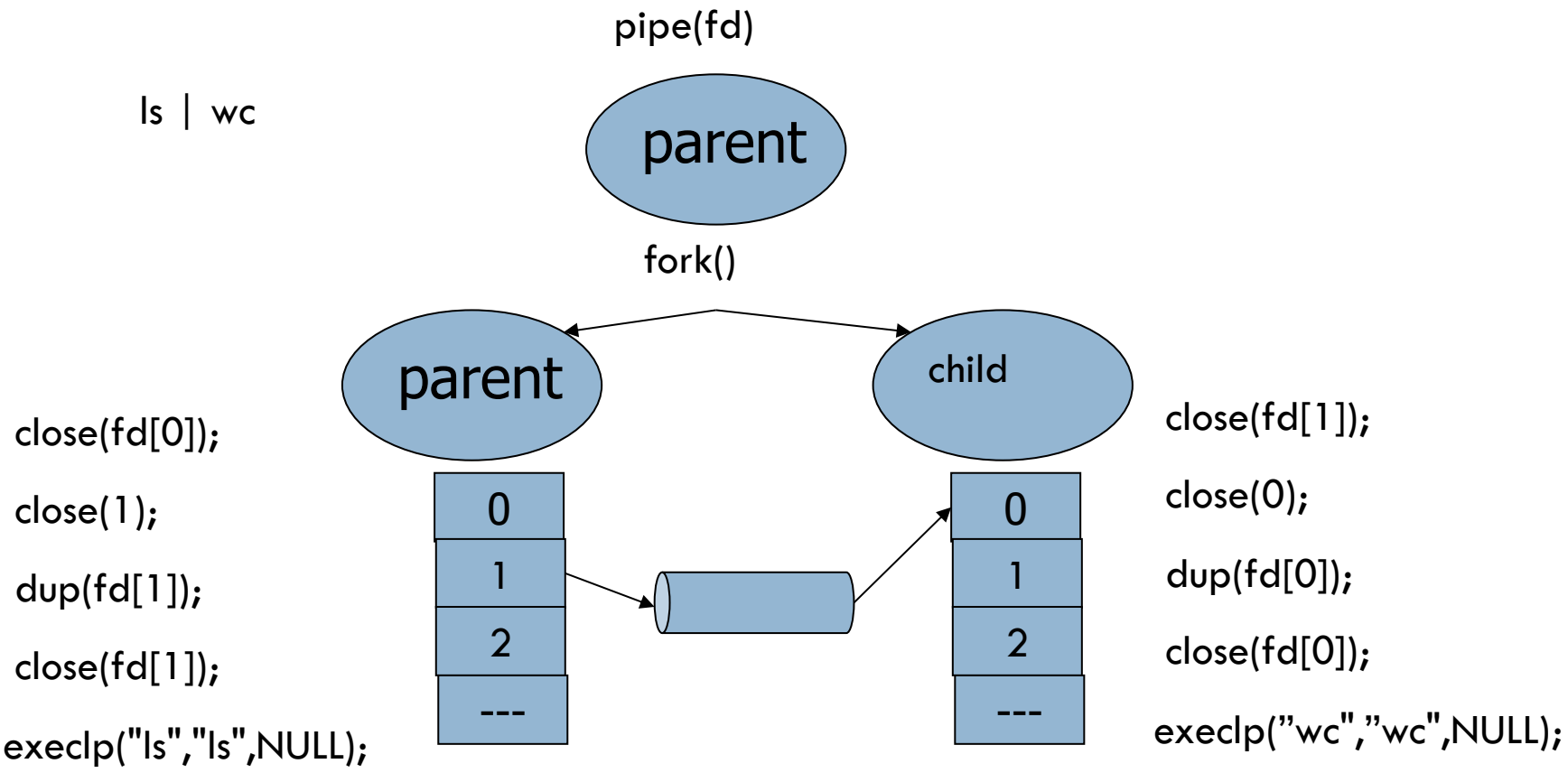
Piping Between Two Processes



Simple Example: Parent/child processes communicating via unnamed pipe.


```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
char *message = ”Message from parent!!" ;
main()
{  char buf[1024] ;
    int fd[2];
    pipe(fd);    /*create pipe*/
    if (fork() != 0) { /* I am the parent */
        write(fd[1], message, strlen (message) + 1) ;
        }
   else { /*Child code */
        read(fd[0], buf, 1024) ;
        printf(”Child received message: %s\n", buf) ;
        }
}
```

Redirection of standard I/O through pipes



Example: "ls | grep a"


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

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

  pipe(fd);
  if (fork()!=0) { /* código del parent */
     close(STDIN_FILENO);
     dup(fd[STDIN_FILENO]);
     close(fd[STDIN_FILENO]);
     close(fd[STDOUT_FILENO]);
     execlp("grep", "grep", "a", NULL);
  } else { /* código del child */
     close(STDOUT_FILENO);
     dup(fd[STDOUT_FILENO]);
     close(fd[STDOUT_FILENO]);
     close(fd[STDIN_FILENO]);
     execlp("ls", "ls", NULL);
  }
  return 0;
}
```

Content

- Signals.
- Timers.
- Exceptions.
- Redirection and Pipes
- **Environment of a Process.**

Concept

- The environment of a process is inherited from the parent. Data:
  - Vector of arguments used to execute the program
  - Environment vector, list of variables <name, value> sent from parent to child
- Passing env variables from parent to child:
  - Flexible form to communicate both processes and determinate child execution aspects in user mode.
- This mechanism allows allows to set up parameters to the level of each individual process:
  - Instead of having the same configuration for all processes

Environment of a process

- Mechanism to pass information to a process.
- Set of **<name,value>** pairs.

- Example

```
PATH=/usr/bin:/home/joe/bin
TERM=vt100
HOME=/home/joe
PWD=/home/joe/books/second
TIMEZONE=MET
```

Environment of a process

- Environmet of a process placed in process stack at initiation.

- Access:
  - Operating system places some default values (e.g.: PATH).
  - Access through commands (set, export).
  - Access through OS API (putenv, getenv).

Environment of a process

- A process gets third argument to main:
  - Address of a table with environment variables.

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

int main(int argc, char** argv, char** envp) {
  for (int i=0;envp[i]!=NULL;i++) {
    printf("%s\n",envp[i]);
  }
  return 0;
}
```

Environment of a process

- `char * getenv(const char * var);`
  - Get the value of an environment variable.
- `int setenv(const char * var, const char * val, int overwrite);`
  - Modifies or adds an environment variable
- `int putenv(const char * par);`
  - Modifies or adds a pair **var=value**.

Summary

- Environment variables allow to pass information to processes
- POSIX signals can be ignored or handled.
- Timers have different resolution in POSIX and Win32.
- Structured exception handling allows to handle anomalous situations through an extension to C language.

OPERATING SYSTEMS: PROCESSES

Signals, exceptions and pipes