İŞLETİM SİSTEMLERİ UYGULAMA

Outline

- Process management -> Section-1
  - creation, differentiation, termination
- Inter-process communcation (IPC) -> Section-1
  - Unix-based
  - Posix
- Threads -> Section-2
- Thread synchronization -> Section-2

Section Outline

- Process definition
- Process handling in Unix
- Process creation
- Process differentiation
- Process termination
- Process synchronization

What is a Process

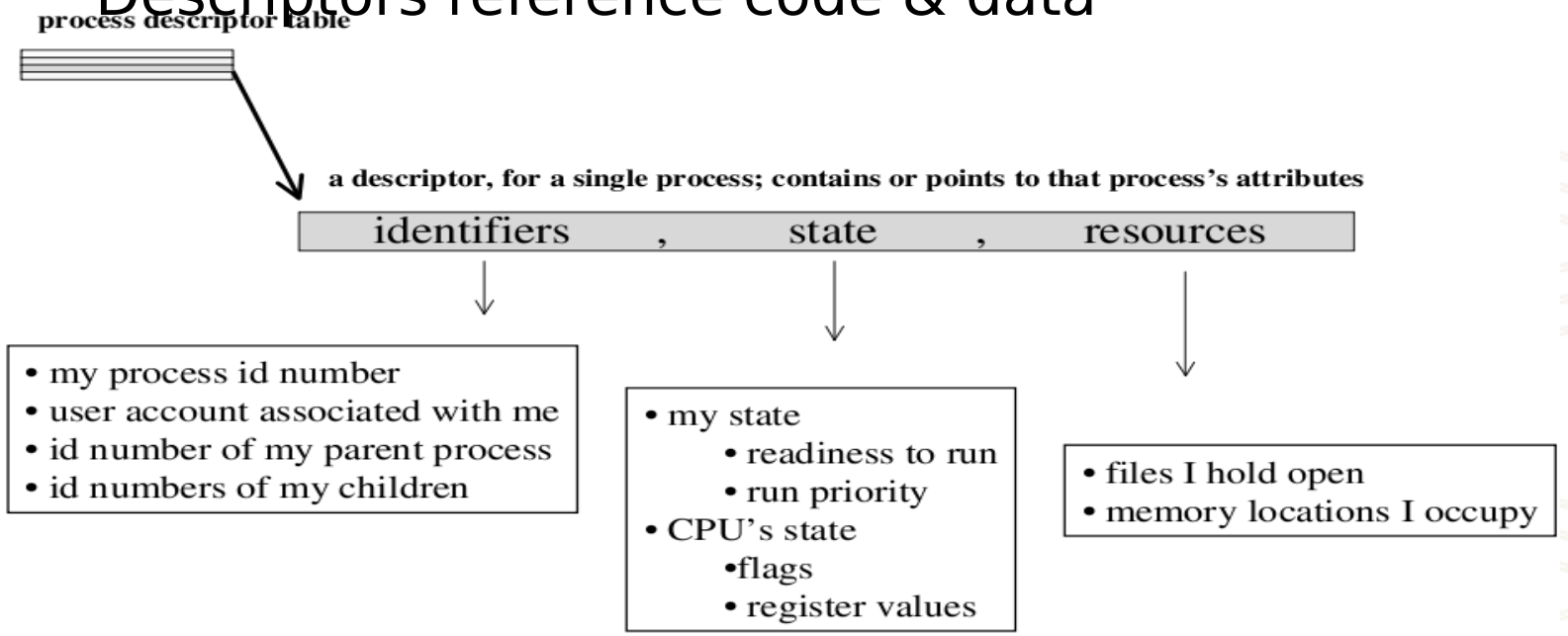
- **Definition ;**
  - A process is an instance of a running program.
- Not the same as “**program**” or “**processor**”
  - A program is a set of instructions and initialized data in a file, usually found on a disk.
- A **process** is an instance of that **program** while it is running, along with the state of all the CPU registers and the values of data in memory.

Process Handling

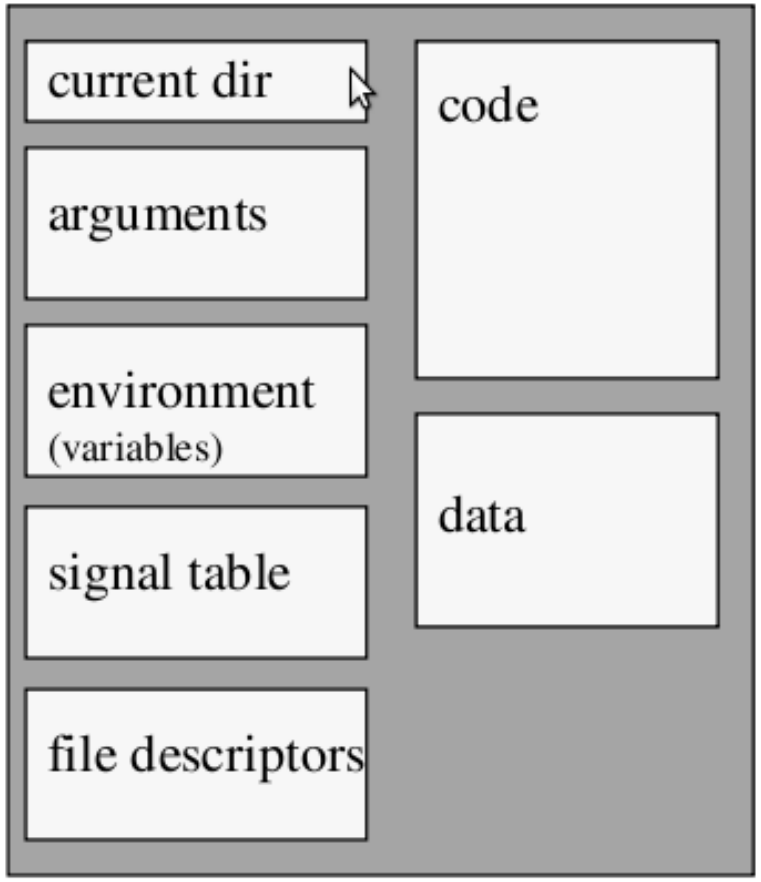
- Constituents of a process
  - Its code
  - Data
    - its own
    - OS’s data used by/for process
  - Various attributes OS needs to manage it

Process Handling

- OS keeps track of all processes
  - Process table/array/list
  - Elements are process descriptors (aka control blocks)
  - Descriptors reference code & data

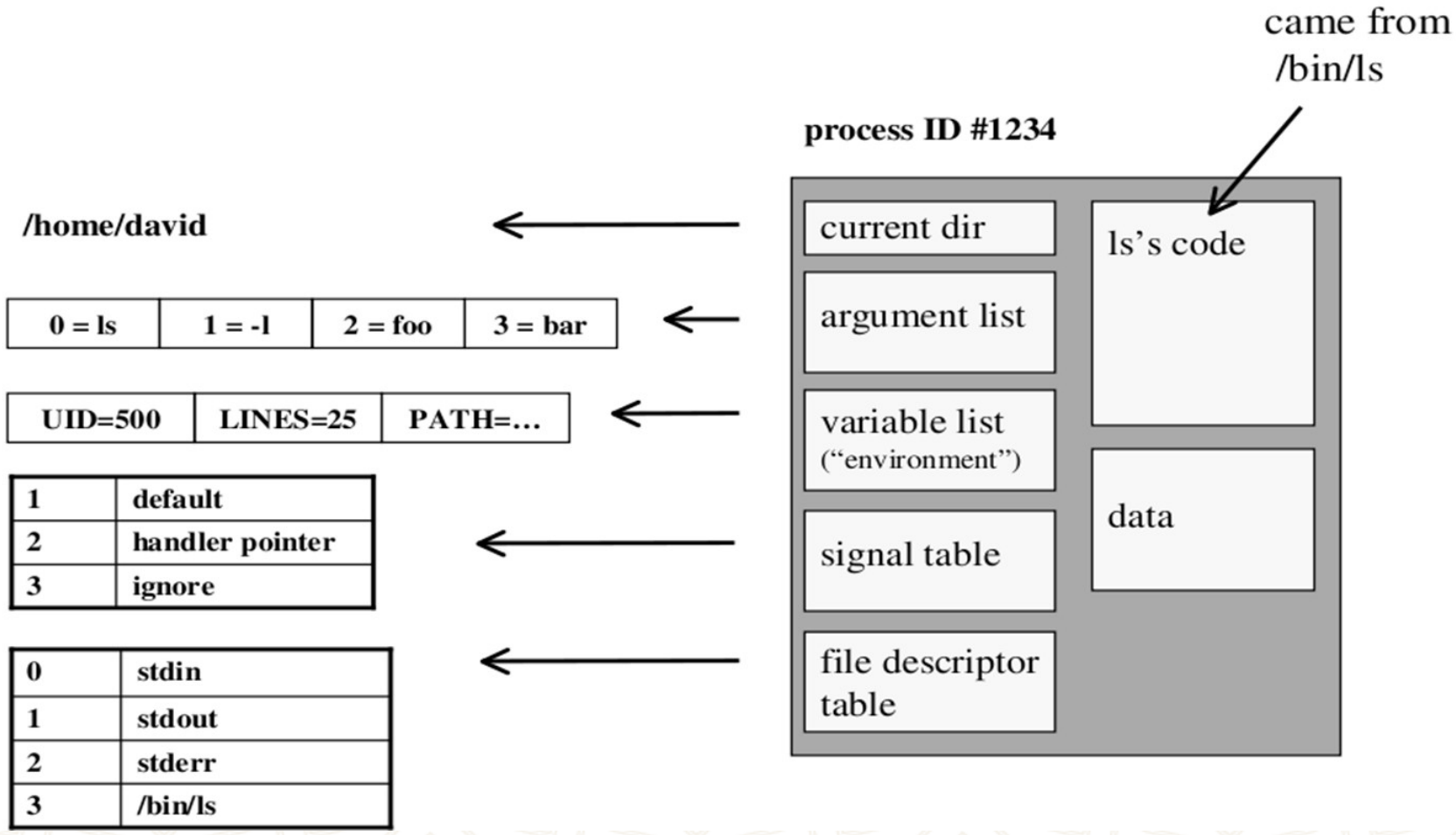


Single process in Unix (consolidated view)



- Some important properties
  - code
  - data
  - current directory
  - argument list
  - environment list
  - responses to signals
  - list of open files

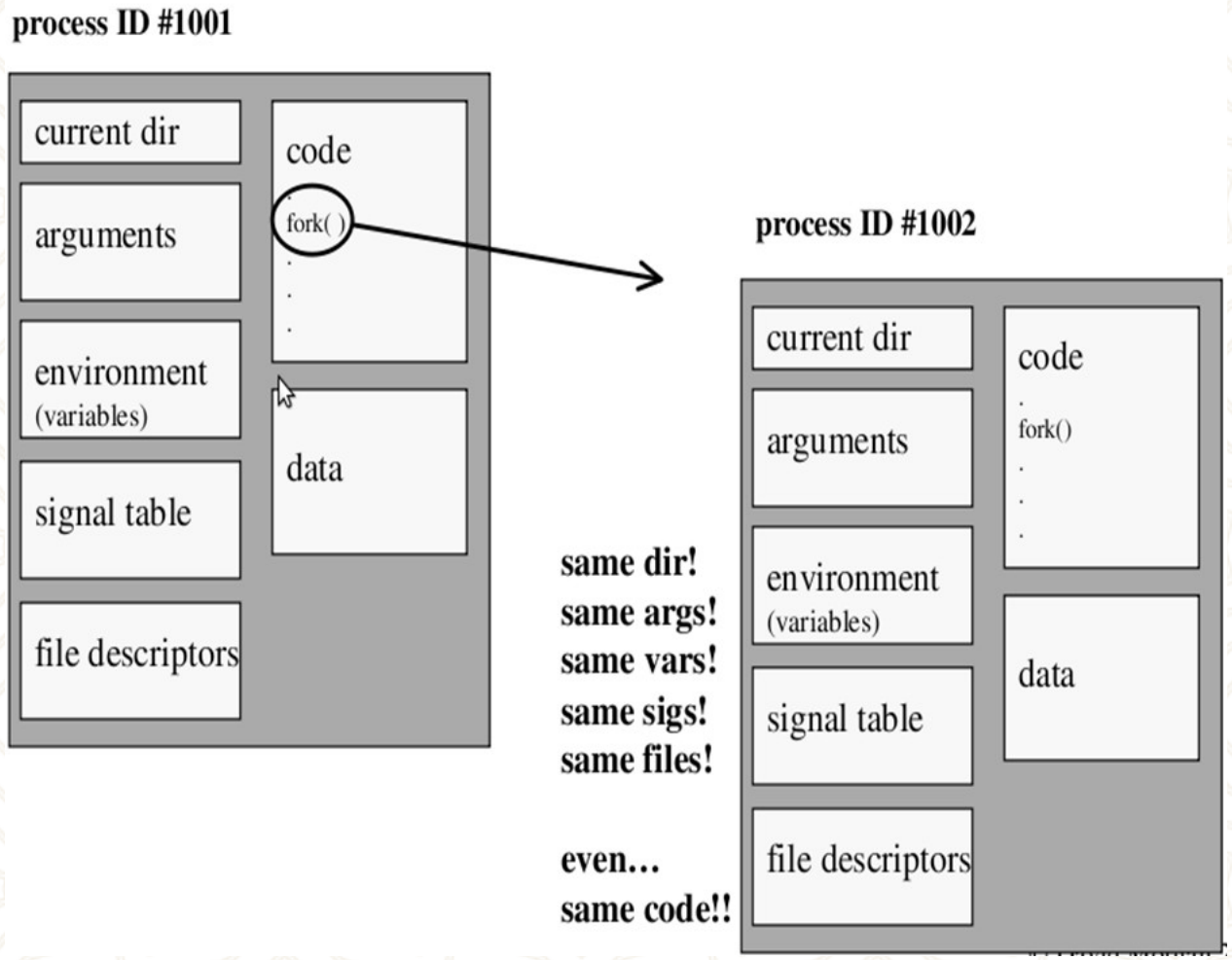
ls -l foo bar


came from
/bin/ls
process ID #1234
/home/david
0 = ls
1 = -l
2 = foo
3 = bar
UID=500
LINES=25
PATH=...
1
default
2
handler pointer
3
ignore
0
stdin
1
stdout
2
stderr
3
/bin/ls
current dir
argument list
variable list
("environment")
signal table
file descriptor
table
ls's code
data

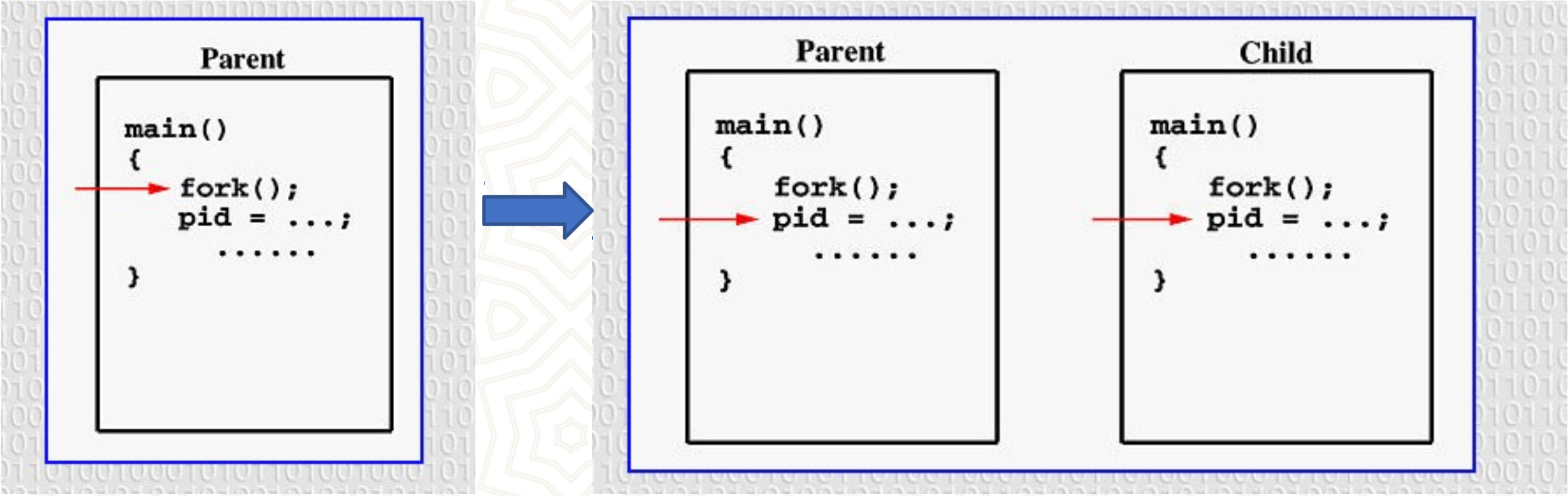
Process Creation

- OS perspective
  - find empty slot in process table
  - write a process descriptor and
  - put it there
  - read in program code from disk
- User perspective
  - System calls
    - **fork(), exec()**

fork() system call



fork() system call



A Simple fork() Example

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

int main ( void )  {

        printf("Message before fork\n");

        fork();

        printf("Message after fork\n");

        return 0;
}
```

- a simple fork example

- Message after fork is printed **twice** !!

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Fork1
Message before fork
Message after fork
lucid@ubuntu:~/Downloads$ Message after fork

lucid@ubuntu:~/Downloads$
```

- **Ex_1_fork1.c**

Self Identification

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

int main ( void )  {

        int forkResult;

        printf("process id : %i\n",getpid());
        forkResult = fork();
        printf("process id : %i - result : %d\n",
                getpid(), forkResult);

        return 0;
}
```

- **Ex_2_fork2.c**

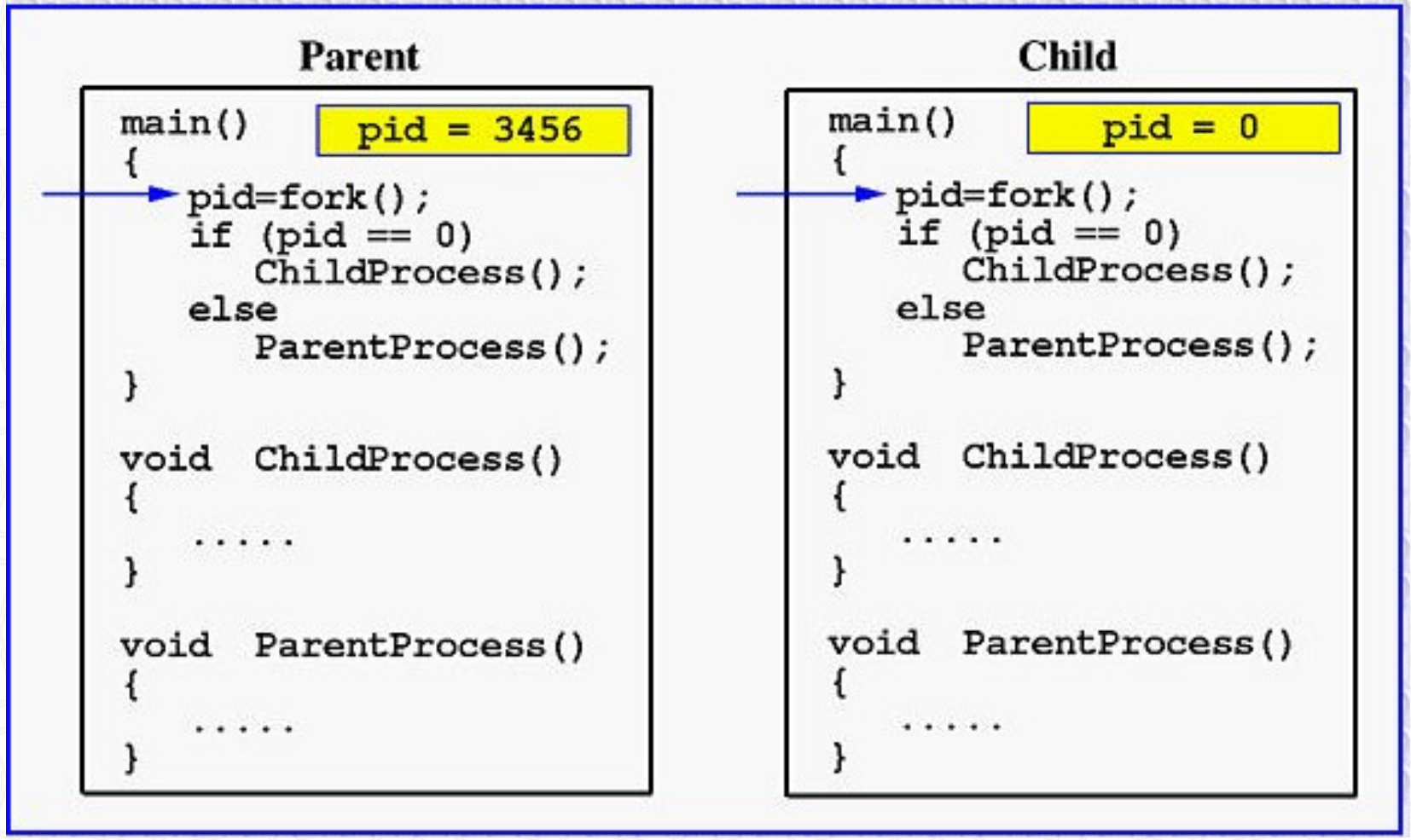
- for the parent process fork returns **child's pid**
- for the child process fork returns **0**

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Fork2
process id : 2682
process id : 2682 - result : 2683
lucid@ubuntu:~/Downloads$ process id : 2683 - result : 0
```

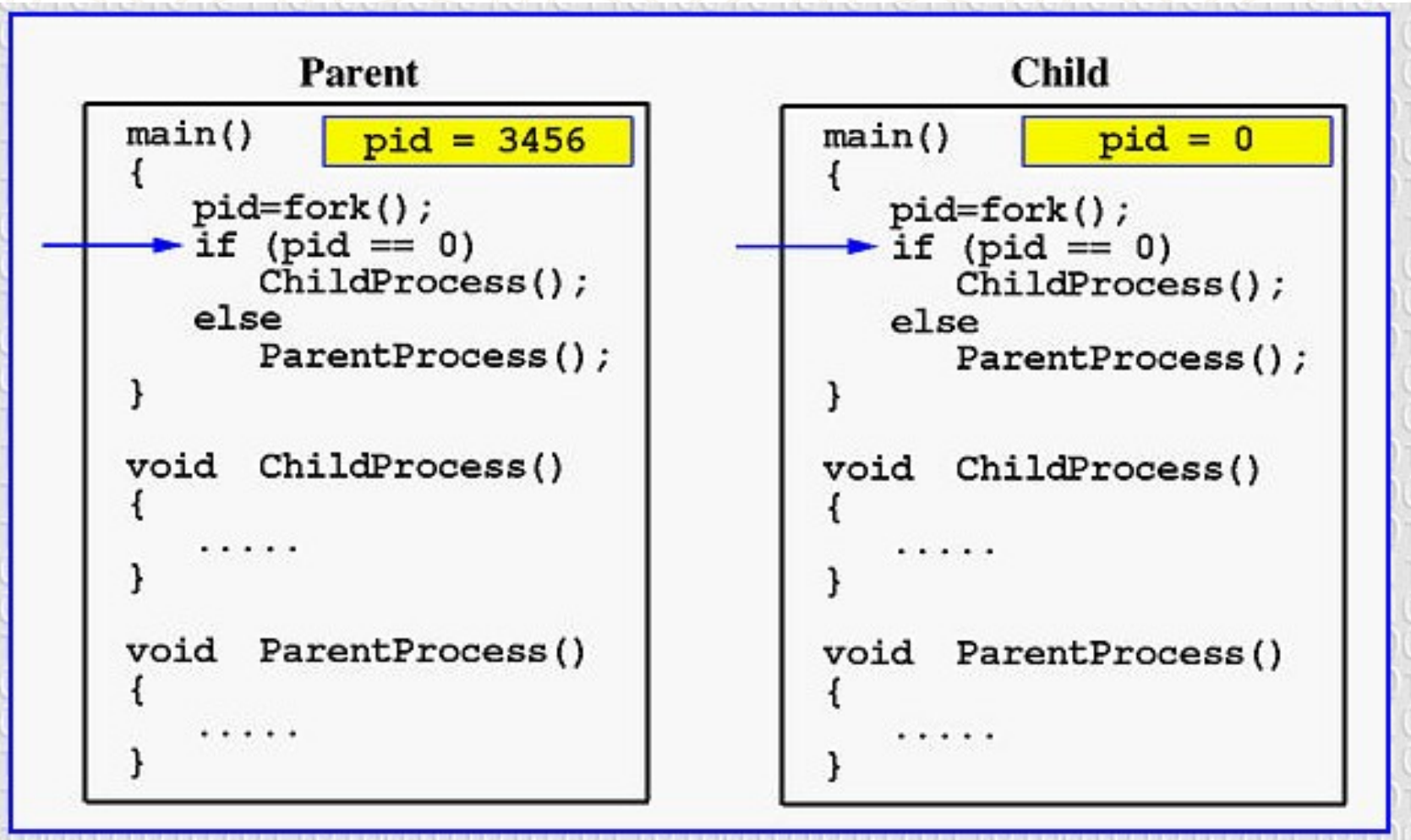
Process Differentiation

- **identical?** not what we had in mind!
- more useful if child does different stuff
- can we give it **different** behaviour?
  - in the form of source code
  - in the form of an existing binary executable
    - exec() family of functions

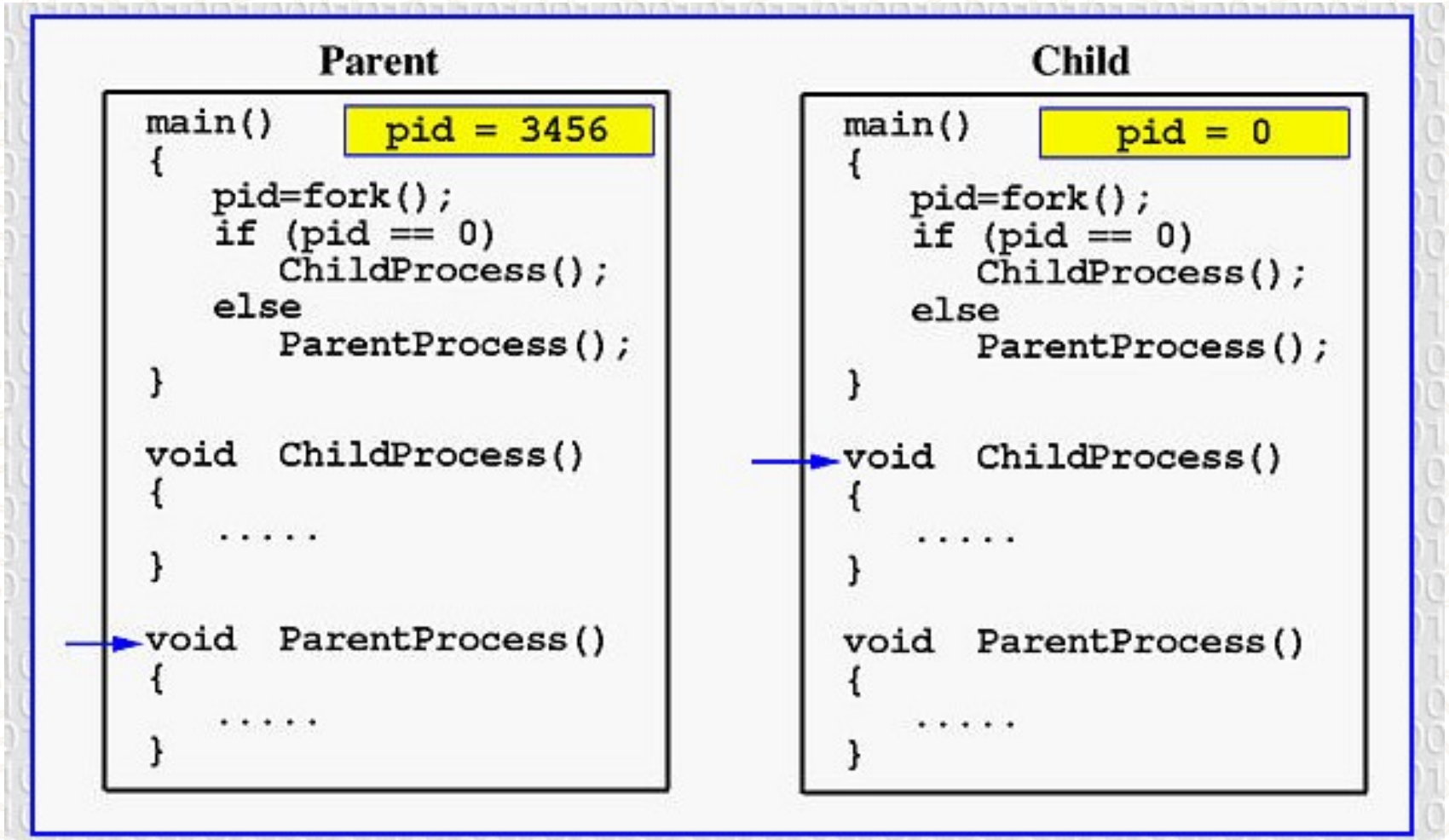
Process Differentiation



Process Differentiation



Process Differentiation



Process Differentiation
by source code

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

int main ( void )  {

        printf("(%i) Parent does something...\n", getpid());

        if(fork())  {  // Parent
                printf("(%i) Parent do completely different stuff\n",getpid());
        } else {       // Child
                printf("(%i) Child can do some stuff\n",getpid());
        }


        exit(0);
}
```

- **Ex_3_fork3.c**

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Fork3
(2767) Parent does something...
(2767) Parent do completely different stuff
lucid@ubuntu:~/Downloads$ (2768) Child can do some stuff
```

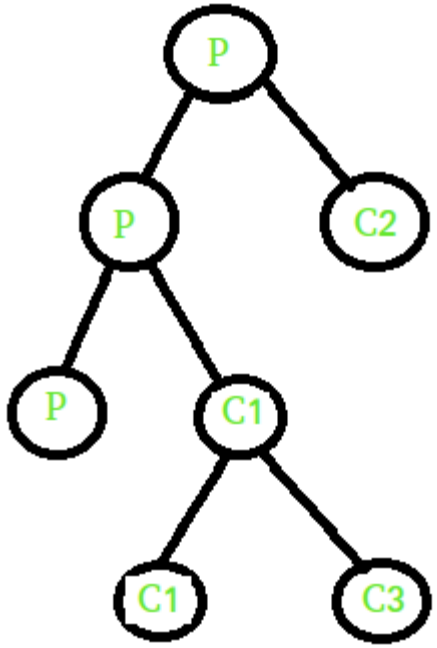
Fork Practice Question

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

int main()
{
    if (fork()) {
        if (!fork()) {
            fork();
            printf("1 ");
        }
        else {
            printf("2 ");
        }
    }
    else {
        printf("3 ");
    }
    printf("4 ");
    return 0;
}
```

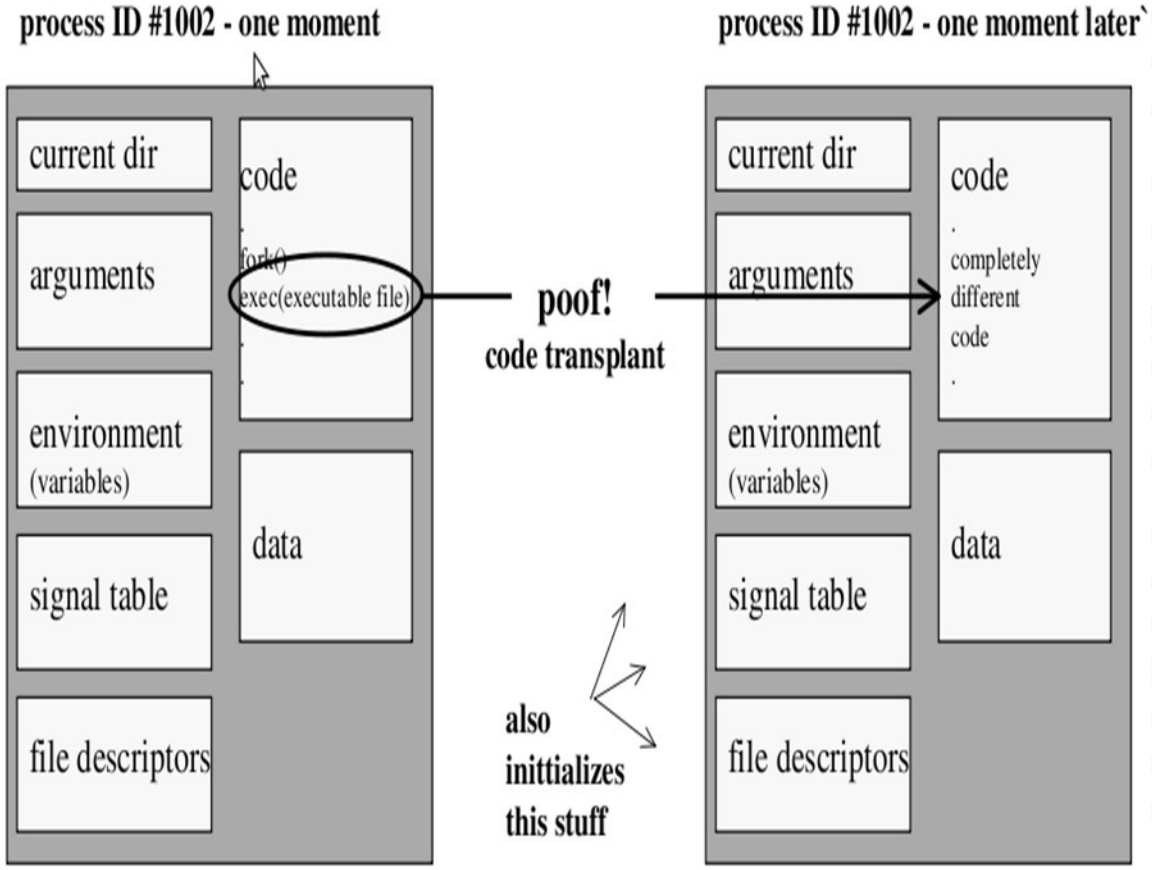
Output:

```
2 4 1 4 1 4 3 4
```



- **Ex_4_fork4.c**

Process Differentiation
by exec() function



Process Differentiation
by exec() function

- **exec() family of functions**
  - int **execl** (const char *pathname, const char *arg0, ...);
  - int **execv** (const char *pathname, char *const argv[]);
  - int **execle** (const char *pathname, const char *arg0, ..., 0, char *const envp[]);
  - int **execlp** (const char *filename, const char *arg0, ...);
  - int **execvp** (const char *filename, char *const argv[]);
  - int **execve** (const char *pathname, char *const argv[], char *const envp[]);

A Simple exec() Example

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

int main ( void )  {

        printf("Parent does stuff and then calls fork...\n");

        if(fork())  { // Parent
                printf("... parent do something completely different\n");
        } else  {     // Child
                printf("Child runs an executable...\n");
                execl("/bin/ls","/bin/ls","-l","/etc/apache2/conf.d/",NULL);
        }

        exit(0);
}
```

```
lucid@ubuntu:~/Downloads$ ./Exec
Parent does stuff and then calls fork...
... parent do something completely different
lucid@ubuntu:~/Downloads$ Child runs an executable...
/bin/ls: cannot access /etc/apache2/conf.d/: No such file or directory
```

- **Ex_5_exec.c**

Process Termination

- void **exit** (int status);
  - exits a process
  - normally return with status 0
- int **atexit** (void (*function)(void));
  - registers function to be executed on exit
- int **wait** (int *child_status)
  - suspends current process until one of its children terminates

exit() vs return

- **return**
  - is an instruction of the language that returns from a function call.
- **exit**
  - is a system call (not a language statement) that terminates the current process.

atexit() example

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

void parentCleaner ( void );

int main ( void )  {

        if(fork())  { // parent process
                atexit(parentCleaner);
                printf("this is parent %i\n",getpid());
        } else  {     // child process
                printf("this is child %i\n",getpid());
        }
        exit(0);
}

void parentCleaner ( void )  {

        printf("cleaning up parent...\n");

}
```

```
File  Edit  View  Terminal  Help
lucid@ubuntu:~/Downloads$ ./Exit1
this is parent 3262
cleaning up parent...
lucid@ubuntu:~/Downloads$ this is child 3263
```

- registers a function to clean up resource at process termination

- **Ex_7_atexit.c**

Zombie Process

- When process terminates, still consumes system resources
  - Various tables maintained by OS
  - Called a zombie; living corpse, half alive, half dead
- **Reaping**
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process
- What if parent does not reap ?
  - if any parent terminates without reaping a child, then child will be reaped by **“init”** process
  - so, only need explicit reaping in long-running processes

Zombie example
non-terminating parent

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

int main ( void ) {

        if(fork())  {  // Parent
                printf("Running parent, pid : %i\n",getpid());
                while(1);
        } else  {    // Child
                printf("Terminating child, pid : %i\n", getpid());
                exit(0);
        }

        exit (0);
}
```

```
lucid@ubuntu:~/Downloads$ ps -ef | grep Zombie
lucid     3380  2182 71 03:42 pts/0    00:00:21 ./Zombie1
lucid     3381  3380  0 03:42 pts/0    00:00:00 [Zombie1] <defunct>
lucid     3402  3382  0 03:43 pts/1    00:00:00 grep --color=auto Zombie
lucid@ubuntu:~/Downloads$
```

- **Ex_8_zombie1.c**

Zombie example
non-terminating child

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

int main ( void ) {

        if(fork())  {  // Parent
                printf("Running parent, pid : %i\n",getpid());
                exit(0);
        } else  {    // Child
                printf("Terminating child, pid : %i\n", getpid());
                while(1);
        }

        exit (0);
}
```

```
lucid@ubuntu:~/Downloads$ ps -ef | grep Zombie
lucid     3467     1 77 03:45 pts/0    00:00:29 ./Zombie2
lucid     3473  3382  0 03:46 pts/1    00:00:00 grep --color=auto Zombie
lucid@ubuntu:~/Downloads$
```

- **Ex_9_zombie2.c**

Synchronizing with child

- int wait(int *child_status)
  - suspends current process until one of its children terminates
  - return value is the pid of the child process that terminated
  - If the child has already terminated, then wait returns its pid immediately
  - If child_status != NULL, then the object it points to will be set to a status indicating why the child process terminated

wait() Example

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

#define numOfChilds 5
int main ( void )  {

        int i;
        int child_status;
        pid_t pid[numOfChilds];
        pid_t wpid;

        for (i = 0; i < numOfChilds; i++)  {
                if ((pid[i] = fork()) == 0)  {
                        exit(100+i);            // create & exit child
                }
        }

        for (i = 0; i < numOfChilds; i++) {
                wpid = wait(&child_status);     // wait for child
                if (WIFEXITED(child_status))  { // check exit status
                        printf("Child %d terminated with exit status %d\n",
                                wpid, WEXITSTATUS(child_status));
                } else {
                        printf("Child %d terminate abnormally\n", wpid);
                }
        }
        exit(0);
}
```

```
lucid@ubuntu:~/Downloads$ ./Wait1
Child 3630 terminated with exit status 100
Child 3631 terminated with exit status 101
Child 3633 terminated with exit status 103
Child 3634 terminated with exit status 104
Child 3632 terminated with exit status 102
lucid@ubuntu:~/Downloads$
```

- **Ex_10_wait1.c**

References

- **man pages**
- http://www.cs.princeton.edu/courses/archive/fall01/cs217/slides/process.pdf
- http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15213-f08/www/lectures/lecture-11.pdf
- http://csapp.cs.cmu.edu
- http://homepage.smc.edu/morgan_david/linux/a12-processes.pdf
- https://www.geeksforgeeks.org/fork-system-call/amp/

Section Outline

- What is IPC
- IPC standards
- Posix IPC Methods
  - Pipes
  - Fifos
  - Signals
  - Semaphores
  - Message queues
  - Shared memory

What is IPC

- **Inter-process communication (IPC)** is a set of methods for the exchange of data among multiple threads in one or more processes.
  - Processes may be running on one or more computers connected by a network.
  - IPC methods are divided into methods for **message passing**, **synchronization**, **shared memory**, and **remote procedure calls** (RPC).

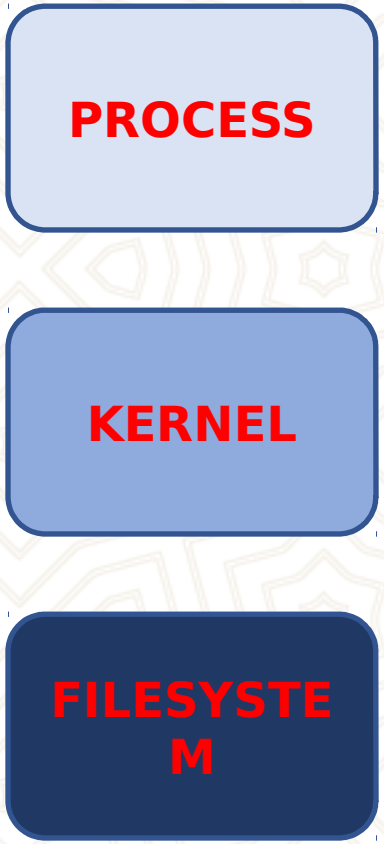
IPC Methods

- Unix
- System V
- POSIX
- Others
  - Sockets
  - Dbus
  - So on...

POSIX IPC

- Pipe
- FIFO
- **Signals**
- **Semaphores**
- Message Queues
- Shared Memory

Persistence of IPC Objects

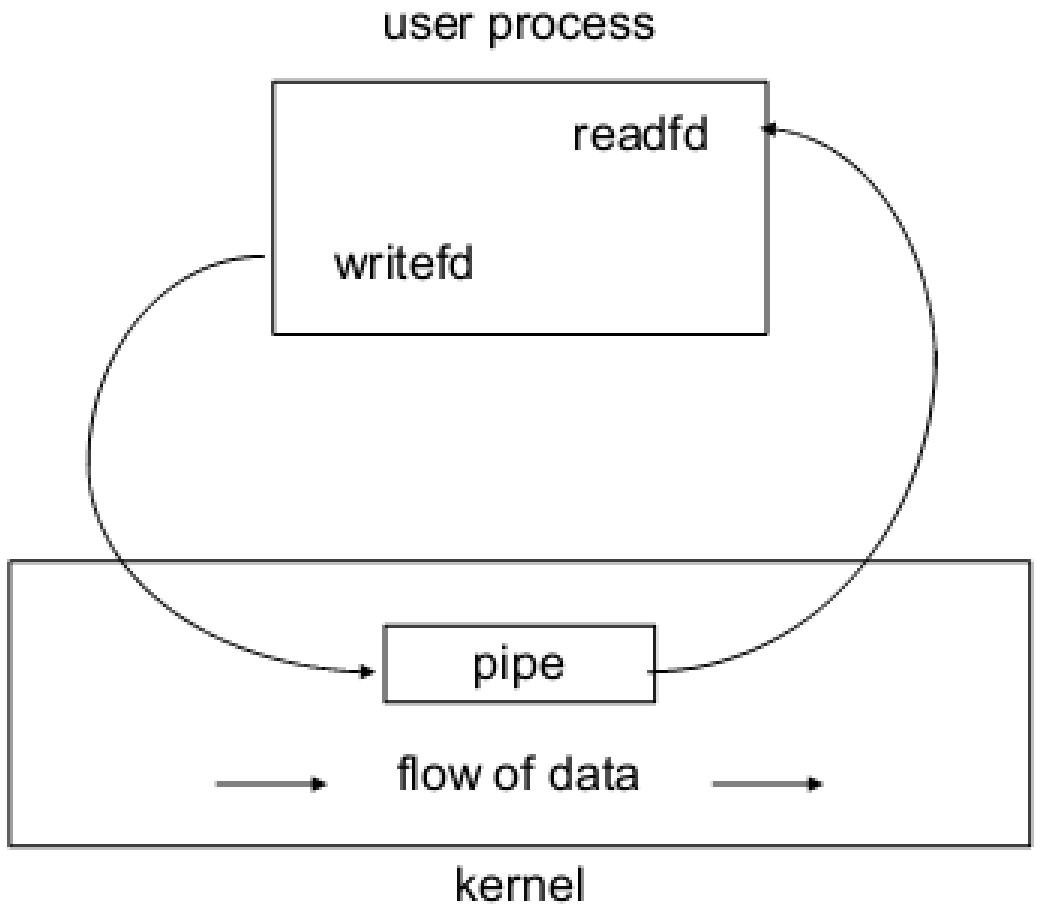


- process-persistent IPC:
  - exists until last process with
  - IPC object closes the object
- kernel-persistent IPC
  - exists until kernel reboots or
  - IPC object is explicitly deleted
- file-system-persistent IPC
  - exists until IPC object is
  - explicitly deleted

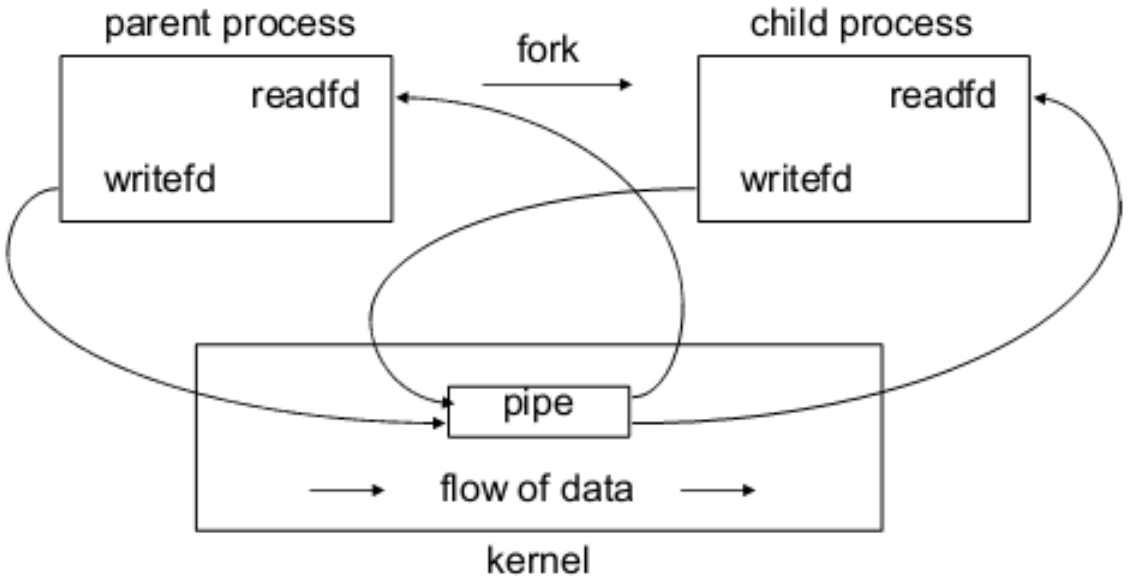
Pipes

- A pipe provides a one-way flow of data
  - example: who | sort| lpr
- The difference between a file and a pipe:
  - pipe is a data structure in the kernel.
- A pipe is created by using the pipe system call
  - int **pipe** (int* filedes);
  - Two file descriptors are returned
- **filedes[0]** is open for reading
- **filedes[1]** is open for writing
- Typical size is **512 bytes** (Minimum limit defined by POSIX)

Pipe
(Single Process)

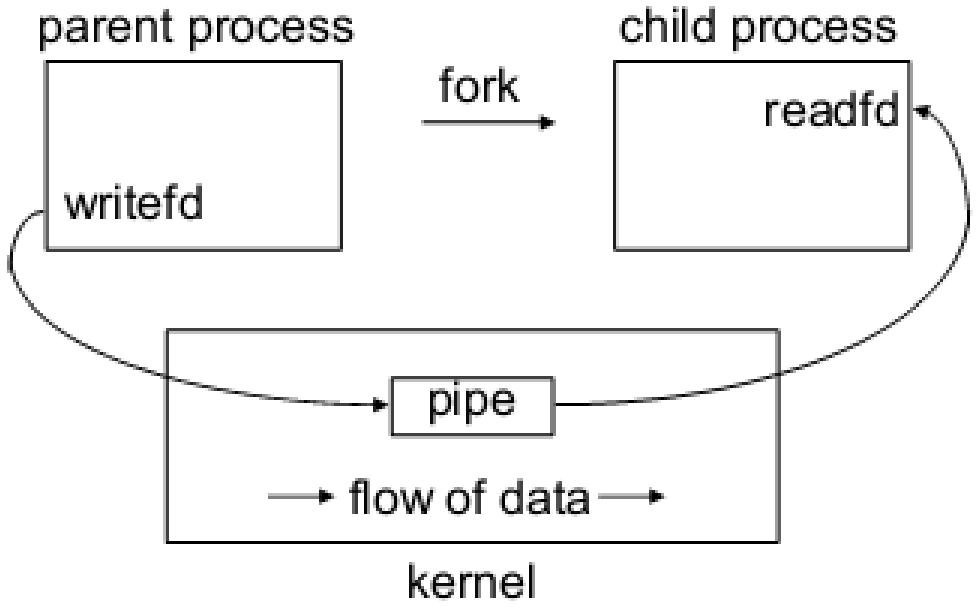


Pipe
(Two Process)



Just after fork

Pipe
(Two Process)



- Parent opens file, child reads file
  - parent closes **read** end of pipe
  - child closes **write** end of pipe

A simple pipe example

```c
int main (int argc, char *argv[])  {

        int pipe1[2];
        pid_t childpid;

        pipe(pipe1);

        if((childpid=fork())==0)  {// Child
                close(pipe1[1]);
                server(pipe1[0]);
                exit(0);
        }

        close(pipe1[0]);
        client(pipe1[1]);

        waitpid(childpid, NULL,0); // wait for child to terminate
        exit(0);
}
```

```c
void client (int writefd ) {

        size_t len;
        char buff[MAX_LINE];

        fgets(buff, MAX_LINE, stdin);
        len = strlen(buff);
        if(buff[len-1]=='\n')
                len--;
        write(writefd, buff, len);
}
```

```
lucid@ubuntu:~/Downloads$ ./Pipe
hello.txt
This
is
the
content
of
Hello.txt
file
lucid@ubuntu:~/Downloads$
```

- **Ex_1_pipe.c**

FIFOs

- Pipes have no names, they can only be used between processes that have a parent process in common.
- FIFO stands for first-in, first-out
- Similar to a pipe, it is a one-way (half duplex) flow of data
- A FIFO has a pathname associated with it, allowing unrelated processes to access a single FIFO
- FIFOs are also called ***named pipes***

FIFOs

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

int **mkfifo** (const char *pathname, mode_t mode)

returns 0 if OK, -1 on error

FIFO example

```c
int main (int argc, char *argv[])  {

        int readfd, writefd;

        if((mkfifo(FIFO1, FIFO_MODE )<0)&&(errno != EEXIST))  {
                printf("can not open %s\n",FIFO1);
                exit(-1);
        }

        if((mkfifo(FIFO2, FIFO_MODE )<0)&&(errno!= EEXIST))  {
                printf("can not open %s\n",FIFO2);
                exit(-1);
        }

        readfd  = open(FIFO1, O_RDONLY);
        writefd = open(FIFO2, O_WRONLY);

        server(readfd, writefd);

        exit(0);
}
```

```
lucid@ubuntu:~/Downloads$ ./Client
enter a file name
hello.txt

sending file name to server

This
is
the
content
of
Hello.txt
file
lucid@ubuntu:~/Downloads$
```

```
lucid@ubuntu:~/Downloads$ ./Server
received file name (hello.txt)
sending contents of the file back to client...
lucid@ubuntu:~/Downloads$
```

- **Ex_2_client.c**
- **Ex_2_server.c**

Example: https://www.geeksforgeeks.org/named-pipe-fifo-example-c-program/

Signals

- Definition
- Signal Types
- Generating Signals
- Responding to a Signal
- POSIX Signal Functions
- Signals & System Calls

Definition

- A signal is an **asynchronous** event which is delivered to a process
- Asynchronous means that the event can occur at any time
  - may be unrelated to the execution of the process
  - e.g. user types ctrl-C, or the modem hangs

Common use of Signals

- Ignore a Signal
- Clean up and Terminate
- Dynamic Reconfiguration
- Report Status
- Turn Debugging on/off
- Restore Previous Handler
- Signals & System Calls

Generating a Signal

- Use the Unix command
  - $> kill -KILL 4481
  - Sends a SIGKILL signal to pid 4481
- ps -l
  - To make sure process died

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

int **kill** (pid_t pid, int sig);

Sends a signal to a process or a group of processes

Return 0 if ok, -1 on error

PID Options

- If pid is positive, then signal sig is sent to the process with the ID specified by pid.
- If pid equals 0, then sig is sent to every process in the process group of the calling process.
- If pid equals -1, then sig is sent to every process for which the calling process has permission to send signals, except for process 1 (init), but see below.
- If pid is less than -1, then sig is sent to every process in the process group whose ID is -pid.
- If sig is 0, then no signal is sent, but error checking is still performed; this can be used to check for the existence of a process ID or process group ID

Responding to a Signal

- A process can;
  - Ignore/discard the signal (not possible for SIGKILL & SIGSTOP)
  - Execute a signal handler function, and then possibly resume execution or terminate
  - Carry out default action for that signal
- The choice is called the process' **signal disposition**

POSIX Signal System

- The POSIX signal system, uses signal sets, to deal with pending signals that might otherwise be missed while a signal is being processed
- The signal set stores collection of signal types
- Sets are used by signal functions to define which signal types are to be processed
- POSIX contains several functions for creating, changing and examining signal sets

POSIX Functions

```
#include <signal.h>
int sigemptyset ( sigset_t *set );
int sigfillset ( sigset_t *set );
int sigismember ( const sigset_t *set,int signo );
int sigaddset ( sigset_t *set, int signo );
int sigdelset ( sigset_t *set, int signo );
int sigprocmask ( int how, const sigset_t *set,
sigset_t *oldset);
```

sigprocmask()

- A process uses a signal set to create a mask which defines the signals it is blocking from delivery
  - Good for critical sections where you want to block certain signals.
- How meanings
  - SIG_BLOCK set signals are added to mask
  - SIG_UNBLOCK set signals are removed from mask
  - SIG_SETMASK set becomes new mask

A Critical Code Region

```
sigset_t newmask, oldmask;
sigemptyset( &newmask );
sigaddset( &newmask, SIGINT );
/* block SIGINT; save old mask */
sigprocmask( SIG_BLOCK, &newmask, &oldmask );
/* critical region of code */
/* reset mask which unblocks SIGINT */
sigprocmask( SIG_SETMASK, &oldmask, NULL );
```

sigaction()

- Supercedes (more powerful than) signal()
  - can be used to code a non-resetting signal()

```
#include <signal.h>
    int sigaction (int signo,
    const struct sigaction *act,
    struct sigaction *oldact )
```

sigaction Structure

```
struct sigaction   {
   void (*sa_handler)( int );  //action to be taken or  SIG_IGN, SIG_DFL
   sigset_t sa_mask;              //additional signal to be blocked
   int sa_flags;                   // modifies action of the signal
   void (*sa_sigaction)( int, siginfo_t *, void * );
}
```

sa_flag

SIG_DFL reset handler to default upon return

SA_SIGINFO denotes **extra information** is passed to handler (.i.e. specifies the use of the “second” handler in the structure.

sigaction() Behavior

- A signo signal causes the sa_handlersignal handler to be called.
- While sa_handler executes, the signals in sa_mask are blocked. Any more signo signals are also blocked.
- sa_handler remains installed until it is changed by another sigaction()call. **No reset problem**

A Simple Example !!!

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

void ouch( int );

int main (void)  {

        struct sigaction act;

        act.sa_handler = ouch;

        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;

        sigaction(SIGINT, &act, 0);

        while(1)  {
                printf("Hello world\n");
                sleep(1);
        }
        exit (0);
}

void ouch( int sigNo ) {

        printf("received SIGINT...\n");
}
```

```
lucid@ubuntu:~/Downloads$ ./Signal
Hello world
Hello world
Hello world
Hello world
Hello world
^Creceived SIGINT...
Hello world
Hello world
Hello world
^Creceived SIGINT...
Hello world
Hello world
Hello world
^Z
[1]+  Stopped                 ./Signal
lucid@ubuntu:~/Downloads$
```

- **Ex_3_signal1.c**

Ignoring Signals

- Other than SIGKILL and SIGSTOP, signals can be ignored.
- Instead of in the previous program:

act.sa_handler = catchint /* or whatever */

We use :

act.sa_handler = SIG_IGN;

The ^C key will be ignored

Restoring Previous Action

- The third parameter to sigaction, oact, can be used:

```
/* save old action */
sigaction( SIGTERM, NULL, &oact);
/* set new action */
act.sa_handler = SIG_IGN;
sigaction( SIGTERM, &act, NULL );
/* restore old action */
sigaction( SIGTERM, &oact, NULL );
```

Changing and Reverting to the default handler

```c
/*you_shot_me.c*/
void handler_3(int signum){
  printf("Don't you dare shoot me one more time!\n");

  //Revert to default handler, will exit on next SIGINT
  signal(SIGINT, SIG_DFL);
}

void handler_2(int signum){
  printf("Hey, you shot me again!\n");

  //switch handler to handler_3
  signal(SIGINT, handler_3);
}

void handler_1(int signum){
  printf("You shot me!\n");

  //switch handler to handler_2
  signal(SIGINT, handler_2);
}


int main(){

  //Handle SIGINT with handler_1
  signal(SIGINT, handler_1);

  //loop forever!
  while(1);

}
```

```
#> ./you_shout_me
^CYou shot me!
^CHey, you shot me again!
^CDon't you dare shoot me one more time!
^C
```

https://www.usna.edu/Users/cs/aviv/classes/ic221/s17/units/06/unit.html