



# Linux System Programming

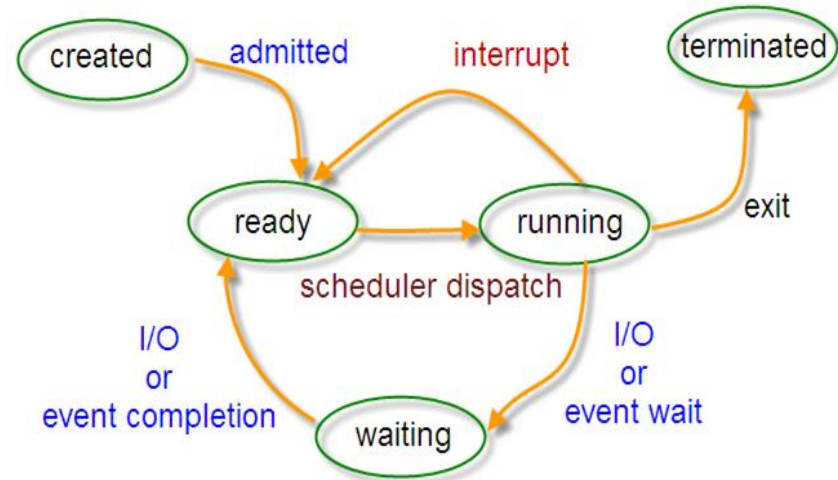
## Part 4 - Processes

IBA Bulgaria  
2018

# Processes in Linux

- **Processes** are object code in execution: active, alive, running programs.
- Processes consist of **data, resources, state**, and a **virtualized computer**.
- Each process is represented by a **unique identifier**, the process ID (**pid**).
- The process that the kernel "runs" when there are no other runnable processes is the **idle process** (pid=0).
- The process that spawns a new process is known as the **parent**; the new process is known as the **child**.
- Each process is owned by a **user** and a **group**.

## The process state:





# Managing the processes

- **ps** - report a snapshot of the current processes.
- **top** - display Linux processes.
- **kill** - send a signal to a process.
- **pgrep**, **pkill** - look up or signal processes based on name and other attributes.
- **killall** - kill processes by name.
- **Ctrl-C** - kill the foreground process.
- **Ctrl-Z** - suspend the foreground process.
- **jobs** - display status of jobs.
- **fg** - returns the suspended process to the foreground.



# Process programming in C

- **getpid()** - get the pid of the calling process.
- **getppid()** - get the pid of the parent process.
- **execl()**, **execlp()**, **execle()**, **execv()**, **execvp()**, **execvpe()** - execute a file.
- **fork()** - create a child process.
- **exit()** - cause normal process termination.
- **wait()**, **waitpid()**, **waitid()** - wait for process to change state.

# Execute a command

```
#include <unistd.h>  
  
int execl (const char *path, const char *arg, ...);  
int execlp (const char *file, const char *arg, ...);  
int execl_e (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[]);  
int execve (const char *filename, char *const argv[], char *const envp[]);
```

The mnemonics are simple:

- **l** and **v** delineate whether the arguments are provided via a list or an array (vector).
- **p** denotes that the user's full path is searched for the given file. Commands using the p variants can specify just a filename, so long as it is located in the user's path.
- **e** notes that a new environment is also supplied for the new process.

# Show own source file

Initialize the variables

Execute **'nano showsrc.c'**

If execution failed, write a message and exit

Print unreachable message for demo purposes

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

int main(int argc, char * argv[])
{
    int ret;
    ret = execl ("/bin/nano", "nano", "showsrc.c", NULL);
    /* Or we could use this:
    const char *args[] = { "nano", "showsrc.c", NULL };
    ret = execvp ("nano", args); */
    if (ret == -1){
        printf("Failed to run nano!\n");
        return EXIT_FAILURE;
    }

    printf("I shouldn't be here!\n");
    return EXIT_SUCCESS;
}
```

## Running a child process

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
pid_t getpid (void);
pid_t getppid (void);
```

Parent process  
pid = 15

```
pid = fork ( );
if (pid > 0){
    printPIDs("PARENT");
    wait(&child_status);
}
else if (!pid){
    printPIDs("CHILD");
    exit(0);
}
else if (pid == -1) {
    printf("ERROR!");
    return EXIT_FAILURE;
}
```

Child process  
pid = 16

```
pid = fork ( );
if (pid > 0){
    printPIDs("PARENT");
    wait(&child_status);
}
else if (!pid){
    printPIDs("CHILD");
    exit(0);
}
else if (pid == -1) {
    printf("ERROR!");
    return EXIT_FAILURE;
}
```

- A successful call to **fork( )** creates a new process, identical in almost all aspects to the invoking process.
- The parent process gets the **pid** of the child process and the child process gets **0**.
- On error, a child process is not created, **fork( )** returns -1, and **errno** is set to:
  - **EAGAIN** - The kernel failed to allocate certain resources, such as a new pid.
  - **ENOMEM** - Insufficient kernel memory was available to complete the request.

# Write sentences into file

Initialize the functions and variables

Fork the process and store the **pid**(s)

If **pid** is positive (parent):

Print parent **pid** and **ppid**

If **pid** is zero (child):

Print child **pid** and **ppid**

If **pid** is -1 print error and exit

```
void printPIDs(char* process_name)
...
int main(int argc, char* argv[])
{
    pid_t pid;

    pid = fork ( );
    if (pid > 0) printPIDs("PARENT");
    else if (!pid){
        printPIDs("CHILD");
        exit(0);
    }
    else if (pid == -1) {
        printf("ERROR while forking!");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```



# Waiting and Terminating child processes

```
#include <stdlib.h>

void exit (int status);
```

- A call to **exit()** performs some basic shutdown steps, and then instructs the kernel to terminate the process.
- The **status** parameter is used to denote the process' exit status.
- When a process terminates, the kernel sends the signal **SIGCHLD** to the parent.


Parent process  
pid = 15

```
pid = fork ( );
if (pid > 0){
    printPIDs("PARENT");
    wait(&child_status);
}
else if (!pid){
    printPIDs("CHILD");
    exit(0);
}
else if (pid == -1) {
    printf("ERROR!");
    return EXIT_FAILURE;
}
```

Child process  
pid = 16

```
pid = fork ( );
if (pid > 0){
    printPIDs("PARENT");
    wait(&child_status);
}
else if (!pid){
    printPIDs("CHILD");
    exit(0);
}
else if (pid == -1) {
    printf("ERROR!");
    return EXIT_FAILURE;
}
```

# Waiting and Terminating child processes



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

pid_t wait (int *status);
int WIFEXITED (status);
int WIFSIGNALED (status);
int WIFSTOPPED (status);
int WIFCONTINUED (status);
int WEXITSTATUS (status);
int WTERMSIG (status);
int WSTOPSIG (status);
int WCOREDUMP (status);
```

- A call to **wait()** returns the *pid* of a terminated child, or **-1** on error.
- If no child has terminated, the call blocks until a child terminates.
- **WIFEXITED** returns true if the process terminated normally.
- In terminated normally, **WEXITSTATUS** provides the lower order eight bits that were passed to `_exit`.
- **WIFSIGNALED** returns true if a signal caused the process' termination.
- In case of signal termination, **WTERMSIG** returns the number of that signal.
- In case of signal termination, **WCOREDUMP** returns true if the process dumped core in response to receipt of the signal.
- **WIFSTOPPED** and **WIFCONTINUED** return true if the process was stopped or continued, respectively.
- If **WIFSTOPPED** is true, **WSTOPSIG** provides the number of the signal that stopped the process.

# Execute and print status

Verify we have enough arguments

Fork the process and store the **pid(s)**

The parent waits for the child and exits on error

On success, prints the status of the termination

The child executes the command in the arguments

Exit with success

```
if (argc < 2) {  
    printf("Usage: %s command, [arg1 [arg2]...]\n", argv[0]);  
    return EXIT_FAILURE;  
}
```

```
if (pid == 0) {  
    execvp(argv[1], &argv[1]);  
    perror("execvp");  
    return EXIT_FAILURE; // Never get there normally  
}  
else {  
    if (wait(&status) == -1) {  
        perror("wait");  
        return EXIT_FAILURE;  
    }  
    if (WIFEXITED(status))  
        printf("Child terminated normally with exit code %i\n",  
              WEXITSTATUS(status));  
    if (WIFSIGNALED(status))  
        printf("Child was terminated by a signal #%i\n",  
              WTERMSIG(status));  
    if (WCOREDUMP(status))  
        printf("Child dumped core\n");  
    if (WIFSTOPPED(status))  
        printf("Child was stopped by a signal #%i\n", WSTOPSIG  
              (status));  
}
```

# Zombies and simple signals handling

- When a child dies before its parent, the kernel puts the child into a special process state - the process is then called a **zombie**.
- A process in this state waits for its parent to inquire about its status and only after this the child process cease to exist even as a zombie.
- If the parent never inquires about a child's status then the zombie becomes a **ghost** - very bad practice.
- If the parent process terminates before its children, then they are **reparented** to the **init** process.
- The init process, in turn, periodically waits on all of its children, ensuring that none remain zombies for too long.

```
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

```
#include <signal.h>
```

```
int sigaction(int signum, const struct sigaction *act,  
              struct sigaction *oldact);
```

```
struct sigaction {  
    void (*sa_handler)(int);  
    void (*sa_sigaction)(int, siginfo_t *, void *);  
    sigset_t sa_mask;  
    int sa_flags;  
    void (*sa_restorer)(void);  
};
```

# Play with zombies

Define the termination signal handler

Initialize variables to use the signal handler

Set action for SIGCHLD, exit on error

Do 10 times:

Fork the process and store the **pid(s)**

The parent prints the pid of  
the new child

The child prints a message  
and exits

Enter endless loop

```
static void sigchld_hdl (int sig)
{
    while (waitpid(-1, NULL, WNOHANG) > 0) {}
}
...
act.sa_handler = sigchld_hdl;
if (sigaction(SIGCHLD, &act, 0)) {
    perror ("sigaction");
    return 1;
}
for (i = 0; i < 10; i++) {
    int pid = fork();
    if (pid == 0) {
        printf("I will leave no zombie\n");
        exit(0);
    } else {
        printf("Created a process with the PID %i\n", pid);
    }
}
while (1) {
    sleep(1);
}
```

# Exercise



## Program *TempFileGenerator*:

Write a program ('**tempgen.c**'), which takes 1 argument: **files\_count**. The program should generate **files\_count** files called '**temp<nr>.tmp**' (where **nr** is between **1** and **files\_count**). Each file is created by a separate process and contains **1000** times the **pid** of the corresponding process as text (pid=1001 should take 4 bytes per ID).

The parent process should notify after each file is created.

Modify the program or write a new one, in which the processes write only in one file, where the **pid** sequences do not overlap (we have 1000 times **pid\_file\_1**, then 1000 times **pid\_file\_2**... then 1000 times **pid\_files\_cound**).