Operating Systems 2021/2022

TP Class 02 – Processes

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Some slides based on previous versions from Bruno Cabral, Paulo Marques and Luis Silva.

operating system
noun
the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

Abbreviation: OS

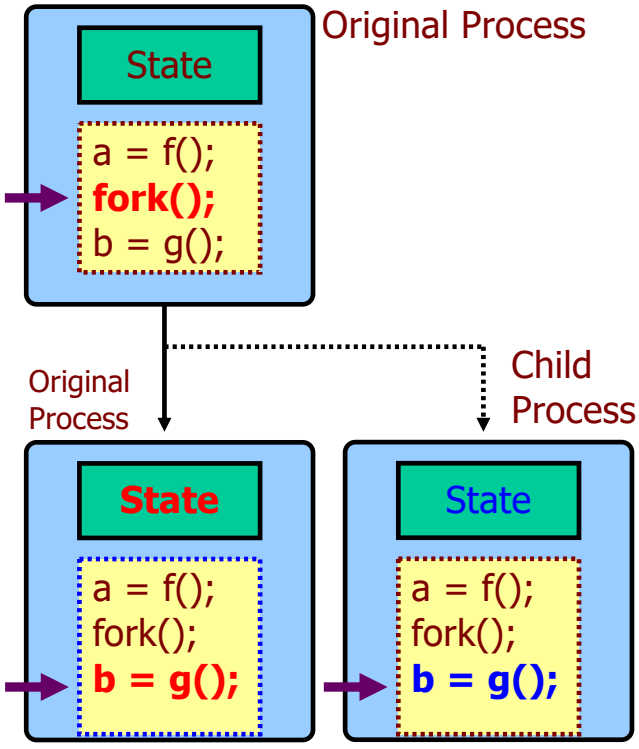
Source: Dictionary.com

1 2 9 0 Departamento de Engenharia Informática FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA

Updated on
20 February 2022

Process Model

- Process creation in Unix is based on spawning child processes which inherit all the characteristics of their fathers
  - Variables, program counter, open files, etc.
  - Spawning a process is done using the fork() system call
- After forking, each process will be executing having different variables and different state.
  - The Program Counter will be pointing to the next instruction
  - Changing a variable in the child program doesn't affect its father (and vice-versa)



Process Management

- Each process has an unique identifier (**PID**). Each process has a father, which is also identified (PPID).
- pid_t getpid(void);
  - Returns the PID of the current process.
- pid_t getppid(void);
  - Returns the PID of the parent process.
- pid_t fork(void);
  - Creates a new process which inherits all its father's state. It returns **0 to the child process** and **the child's PID to the original process**.
- pid_t wait(int* status);
  - Waits until a child process exits. The status of the child is set in status. (status is the return value of the process). Returns -1 in case of error.
- pid_t waitpid(pid_t who, int* status, int options);
  - Same as wait() but allows to wait for a particular child. By using WNOHANG in options, allows for checking if a child has already exited without blocking. 0 in who means "wait for any child". Returns -1 in case of error.

Using processes to do different things

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

int main()
{
  pid_t id;

  id = fork();
  if (id == 0)
    {
      printf("[%d] I'm the son!\n", getpid());
      printf("[%d] My parent is: %d\n", getpid(), getppid());
    }
  else
    {
      printf("[%d] I'm the father!\n", getpid());
      wait(NULL);
    }
  return 0;
}
```

The key idea is to create asymmetry between processes

demo1.c

And the result is...

```
pmarques@null:~/IPC$ gcc -Wall hello_world.c -o hello_world
pmarques@null:~/IPC$ ./hello_world
[2190] I'm the father!
[2191] I'm the son!
[2191] My parent is: 2190
pmarques@null:~/IPC$ _
```

But why did we do wait(NULL)?

```
...
printf("[%d] I'm the father!\n", getpid());
wait(NULL);
...
```

wait(NULL) ensures that the father process waits for any of its children, discarding child exit status.

Process Termination in UNIX

- A process is only truly eliminated by the operating system when its father calls wait()/waitpid() on it.
  - This allows the parent check things like the exit code of its son's
- **Zombie Process**: One that has died and its parent has not acknowledged its death (by calling wait())
  - Be careful with this if your are designing servers. They are eating up resources!!
- **Orphan Process**: One whose original parent has died. In that case, its parent becomes *init* (process 1). (not always true; in some recent versions of operating systems the parent becomes the nearest sub-reaper process - a sub-reaper fulfills the role of *init*(1) for its descendant processes.)

Let's generate some Zombies

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

void worker() {
  printf("[%d] Hi, I'm a worker process! Going to die...\n",
         getpid());
}

int main()
{
  for (int i=0; i<10; i++) {
    if (fork() == 0) {
      worker();
      exit(0);
    }
  }
  printf("[%d] Big father is sleeping!\n", getpid());
  sleep(10);

  return 0;
}
```

demo2.c

Let's see adoption

```c
#include (...)

void worker()
{
  sleep(10);
  printf("[%d] Let's see who my dady is: %d\n", getpid(),
         getppid());
}

int main()
{
  for (int i=0; i<10; i++)
  {
     if (fork() == 0)
     {
          worker();
          exit(0);
     }
  }

  printf("[%d] Big dady is going away!\n", getpid());
  return 0;
}
```

demo3.c

How to structure code

```
(...)

if ((id = fork()) == 0)
{
  // Huge amount of code
  // ...
}
else
{
  // Huge amount of code
  // ...
}

(...)
```

Don't do it!

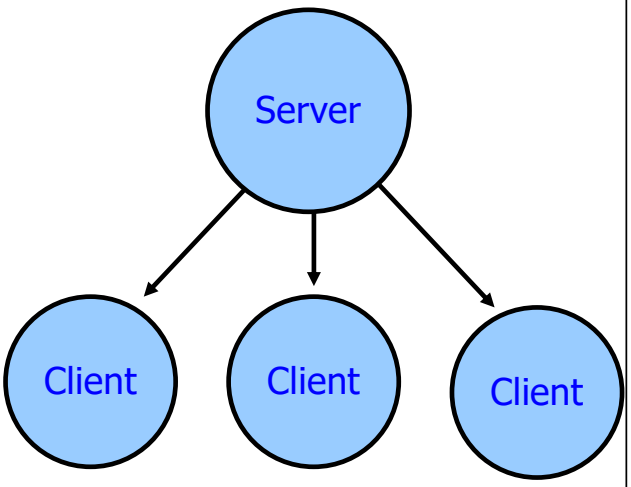
Fairly common...

How to structure code

```
void client(int id)
{
  // Client code
  // ...
}

if ((id = fork()) == 0)
{
  client(id);
  exit(0);
}
else if (id == -1)
{
  error();
}

// Original process code
// ...
```



Note: You still have to consider how to take care of zombies

How a process becomes another executable

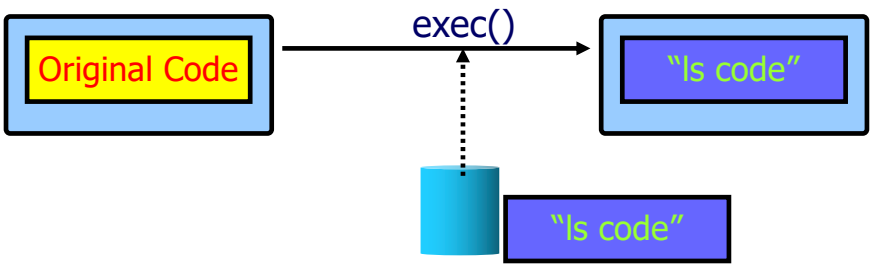
- Somehow the OS must be able to execute code starting from an executable file
  - e.g. how does the shell (bash) becomes 'ls'?
- int execl(const char *path, const char *arg, ...);
- int execlp(const char *file, const char *arg, ...);
- int execle(const char *path, const char *arg, ..., char *const envp[]);
- int execv(const char *path, char *const argv[]);
- int execvp(const char *file, char *const argv[]);
- "exec family" of functions
  - Allow to substitute the current process executable image by another one. The substitution is complete!
  - The functions that have a 'p' make use of the environment PATH; The functions that have a 'v' make use of a pointer to an array on parameters; The functions that have an 'l' have the parameters passed separated by commas
  - Make sure that the first parameter is the name of the program!

Example

- Simple program that lists the files in the current directory

```
pmarques@null:~/IPC$ g++ -Wall list_files.c -o list_files
pmarques@null:~/IPC$ ./list_files
.       adopt.c         list_files    posix_version    zombie.c
..      hello_world     list_files.c  posix_version.c
adopt   hello_world.c   posix         zombie
pmarques@null:~/IPC$ _
```

- Note: A successful exec() never returns
  - The code, the stack, the heap, it's all replaced by the new executable



The corresponding code...

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

int main()
{
  if (execlp("ls", "ls", "-a", NULL) == -1)
    perror("Error executing ls: ");
  else
    printf("This cannot happen!\n");

  return 0;
}
```

demo4.c

Using an array can be more flexible...

```c
char* ls_param[] = { "ls", "-a", NULL };

if (execvp(ls_param[0], ls_param) == -1)
  perror("Error executing ls: ");
```


Code to wait for the end of an exec call...

```c
makeargv(buf, DELIM, &args);
pid_t child = fork();
if (child == 0)
{
    execvp(args[0], args);
    perror("Erro ao executar o comando");
    exit(-1);
}
waitpid(child, NULL, 0);
```

Information about processes in Linux

- Command `ps` (process status)
  - reports a snapshot of the current processes
  - To see all processes in the OS in full-format listing

    `ps -ef`
  - To select all processes from user "user1"

    `ps -ef | grep user1`
  - Some process state codes shown in `ps`
    - Seen when `ps` is used with a state output modifier
      - E.g.: select all processes and show specific columns
        - `ps -eo uid,pid,tty,time,cmd,stat`
    - R running
    - T stopped by a job control signal or because it is being traced
    - Z defunct ("zombie")
    - + the process is in the foreground process group.

Class demos included

- Demo01 – create process

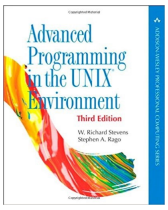
  `demo01.c`
- Demo02 – zombies

  `demo02.c`
- Demo02 – adoption

  `demo03.c`
- Demo04 – exec

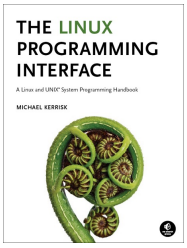
  `demo04.c`

References



- **Advanced Programming in the UNIX Environment**
  3rd Edition (2013)
  W. Richard Stevens, Stephen A. Rago
  Addison-Wesley
  - Chapter 8: Process Control

Where to learn more?



- The Linux Programming Interface
  2010
  Michael Kerrisk
  No Starch Press
  - Chapters 24 to 27

INTRODUCTION TO ASSIGNMENT 03 – "PROCESSES"

Thank you! Questions?

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.
—Rudyard Kipling