Sistemas de Operação /
Fundamentos de Sistemas Operativos
Processes in Unix/Linux

Artur Pereira `<artur@ua.pt>`

DETI / Universidade de Aveiro

Outline

Process
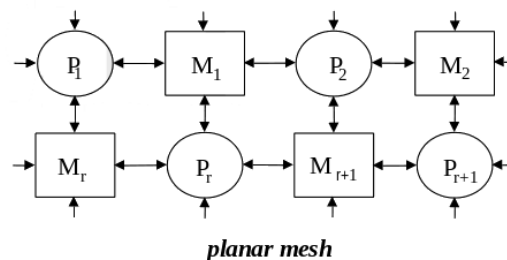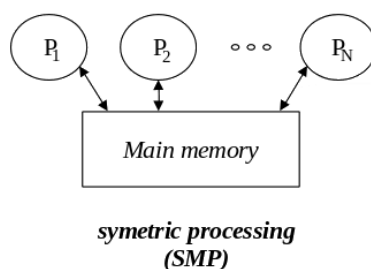Program vs. process

- Program – set of instructions describing how a task is performed by a computer
  - In order for the task to be actually performed, the corresponding program has to be executed
- Process – an entity that represents a computer program being executed
  - it represents an activity of some kind
  - it is characterized by:
    - addressing space – code and data (actual values of the diferent variables) of the associated program
    - input and output data (data that are being transfered from input devices and to output devices)
    - process specific variables (PID, PPID, ...)
    - actual values of the processor internal registers
    - state of execution
- Different processes can be running the same program

- In general, there are more processes than processors – *multiprogramming*
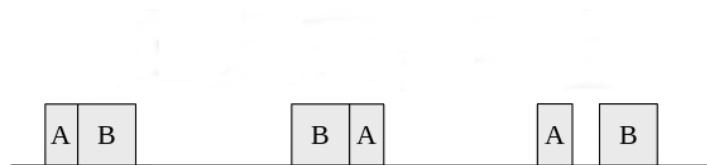
Multiprocessing vs. Multiprogramming
Multiprocessing

- Parallelism – ability of a computational system to simultaneously run two or more programs
  - more than one processor is required (one for each simultaneous execution)
- The operating systems of such computational systems supports multiprocessing

symetric processing
(SMP)

planar mesh
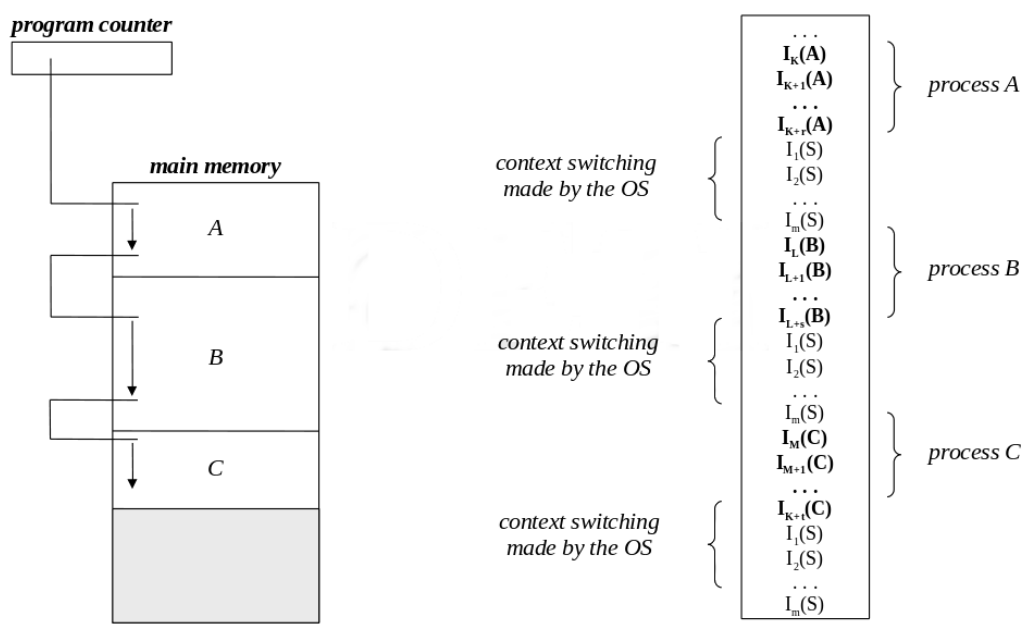
Multiprocessing vs. Multiprogramming
Multiprogramming

- Concurrency – illusion created by a computational system of apparently being able to simultaneously run more programs than the number of existing processors
  - The existing processor(s) must be assigned to the different programs in a time multiplexed way
- The operating systems of such computational systems supports multiprogramming

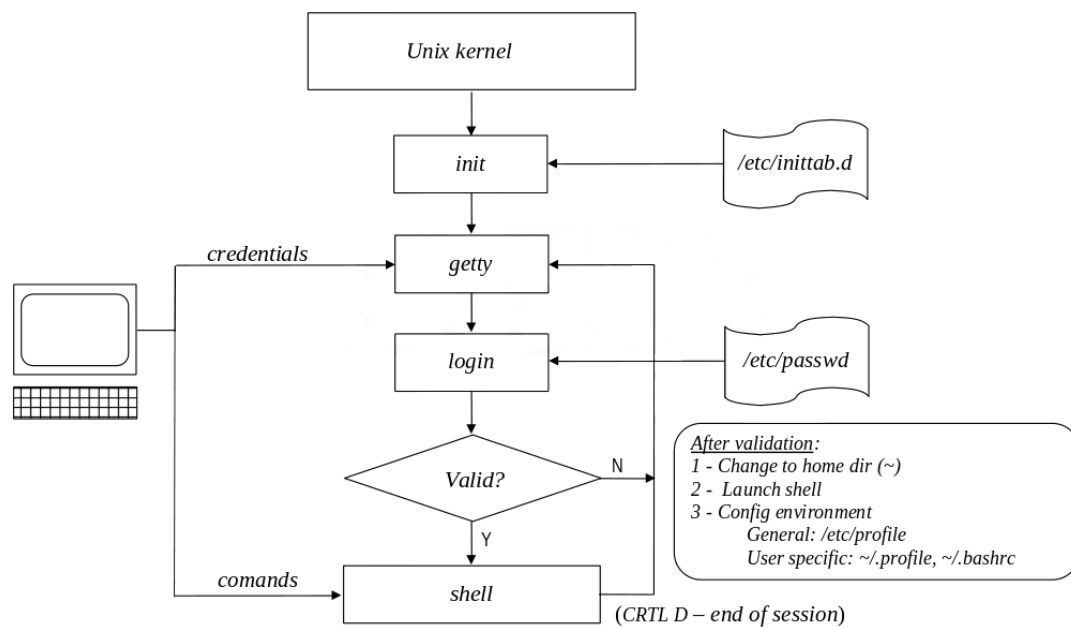- Programs A and B are executing concurrently in a single processor computational system

Process
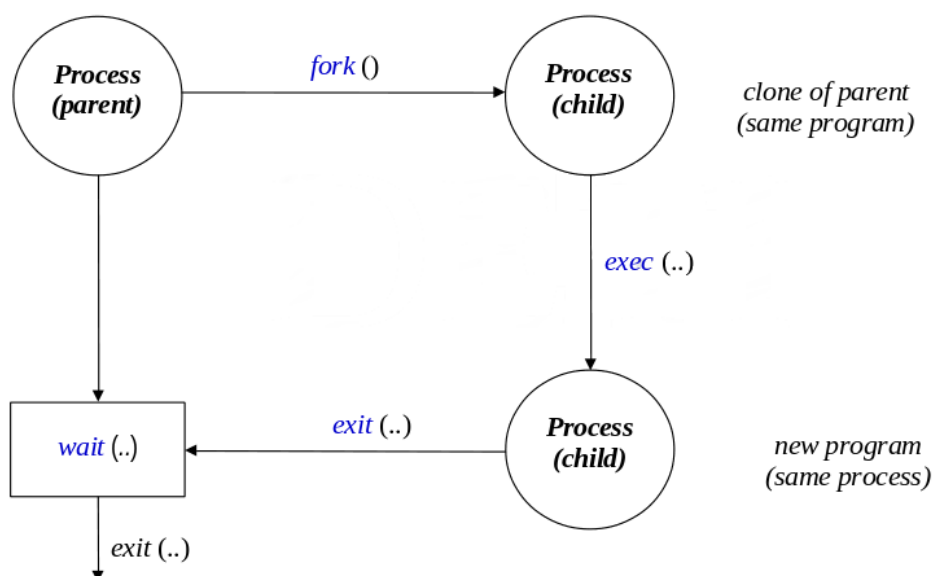Execution in a multiprogrammed environment

Processes in Unix
Traditional login


```
                    Unix kernel

                       init  ◄────── /etc/inittab.d

   credentials
         ┌─────────► getty ◄──────┐
  ▭▭     │              │         │
  [term] │            login ◄──── /etc/passwd
         │              │         │
         │          ◇ Valid? ◇──N─┤
         │              │Y        │
  comands│            shell ──────┘
         └──────────►         (CRTL D – end of session)
```

After validation:
1 - Change to home dir (~)
2 - Launch shell
3 - Config environment
      General: /etc/profile
      User specific: ~/.profile, ~/.bashrc

Processes in Unix
Creation by cloning

Process (parent) — *fork* () → Process (child) — *clone of parent (same program)*

exec (..)

wait (..) ← *exit* (..) — Process (child) — *new program (same process)*

exit (..)

Processes in Unix
Process creation: `fork1`

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

int main(void)
{
    printf("Before the fork:\n");
    printf("  PID = %d, PPID = %d.\n",
        getpid(), getppid());

    fork();

    printf("After the fork:\n");
    printf("  PID = %d, PPID = %d.\n"
        " Am I the parent or the child?"
        " How can I know it?\n",
            getpid(), getppid());

    return EXIT_SUCCESS;
}
```

- The fork clones the executing process, creating a replica of it
- The address spaces of the two processes are equal
  - actually, just after the fork, they are the same
  - typically, a copy on write approach is followed
- The states of execution are the same
  - including the program counter
- Some process variables are different (PID, PPID, ...)

- What can we do with this?

Processes in Unix
Process creation: `fork2` and `fork3`

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

int main(void)
{
    printf("Before the fork:\n");
    printf("  PID = %d, PPID = %d.\n",
        getpid(), getppid());

    int ret = fork();

    printf("After the fork:\n");
    printf("  PID = %d, PPID = %d.\n",
        getpid(), getppid());
    printf("  ret = %d\n", ret);

    return EXIT_SUCCESS;
}
```

- The value returned by the fork is different in parent and child processes
  - in the parent, it is the PID of the child
  - in the child, it is always 0

Processes in Unix
Process creation: `fork2` and `fork3`

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

int main(void)
{
  printf("Before the fork:\n");
  printf("  PID = %d, PPID = %d.\n",
      getpid(), getppid());

  int ret = fork();

  if (ret == 0)
  {
    printf("I'm the child:\n");
    printf("  PID = %d, PPID = %d\n",
        getpid(), getppid());
  }
  else
  {
    printf("I'm the parent:\n");
    printf("  PID = %d, PPID = %d\n",
        getpid(), getppid());
  }

  return EXIT_SUCCESS;
}
```

- The value returned by the fork is different in parent and child processes
  - in the parent, it is the PID of the child
  - in the child, it is always 0
- This return value can be used as a boolean variable
  - so we can distinguish the code running on child and parent

- Still, what can we do with it?

Processes in Unix
Process creation: `fork3`

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

int main(void)
{
  printf("Before the fork:\n");
  printf("  PID = %d, PPID = %d.\n",
      getpid(), getppid());

  int ret = fork();

  if (ret == 0)
  {
    printf("I'm the child:\n");
    printf("  PID = %d, PPID = %d\n",
        getpid(), getppid());
  }
  else
  {
    printf("I'm the parent:\n");
    printf("  PID = %d, PPID = %d\n",
        getpid(), getppid());
  }

  return EXIT_SUCCESS;
}
```

- In general, used alone, the fork is of little interest
- In general, we want to run a different program in the child
  - exec system call
  - there are different versions of exec
- Sometimes, we want the parent to wait for the conclusion of the program running in the child
  - wait system call

- *In this code, we are assuming the* `fork` *doesn't fail*
  - *in case of an error, it returns* -1

Process creation in Unix
Launching a program: `fork + exec`

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

int main(int argc, char *argv[])
{
  /* check arguments */
  if (argc != 2)
  {
    fprintf(stderr, "launch <<cmd>>\n");
    exit(EXIT_FAILURE);
  }
  char *aplic = argv[1];

  printf("====================\n");

  /* clone phase */
  int pid;
  if ((pid = fork()) < 0)
  {
    perror("Fail cloning process");
    exit(EXIT_FAILURE);
  }
```

```c
/* exec and wait phases */
if (pid != 0) // only runs in parent process
{
  int status;
  while (wait(&status) == -1);
  printf("=================\n");
  printf("Process %d (child of %d)"
          " ends with status %d\n",
          pid, getpid(), WEXITSTATUS(status));
}
else // this only runs in the child process
{
  execl(aplic, aplic, NULL);
  perror("Fail launching program");
  exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS); // or return EXIT_SUCCESS
}
```

Processes in Unix
Execution of a C/C++ program
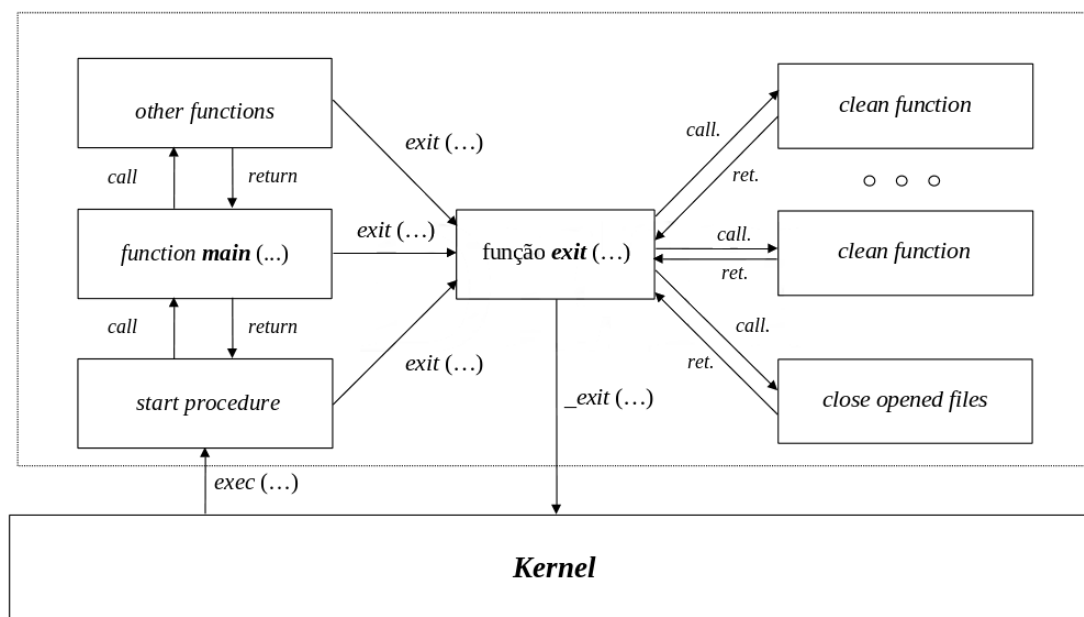
Processes in Unix
Executing a C/C++ program: `atexit`

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

/* cleaning functions */
static void atexit_1(void)
{
    printf("atexit 1\n");
}

static void atexit_2(void)
{
    printf("atexit 2\n");
}

/* main programa */
int main(void)
{
    /* registering at exit functions */
    assert(atexit(atexit_1) == 0);
    assert(atexit(atexit_2) == 0);

    /* normal work */
    printf("hello world 1!\n");

    for (int i = 0; i < 5; i++) sleep(1);

    return EXIT_SUCCESS;
}
```

- The atexit function allows to register a function to be called at the program's normal termination
- They are called in reverse order relative to their register

- *What happens if the termination is forced?*

Processes in Unix
Command line arguments and environment variables

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

int main(int argc, char *argv[], char *env[])
{
    /* printing command line arguments */
    printf("Command line arguments:\n");
    for (int i = 0; argv[i] != NULL; i++)
    {
        printf(" %s\n", argv[i]);
    }

    /* printing all environment variables */
    printf("\nEnvironment variables:\n");
    for (int i = 0; env[i] != NULL; i++)
    {
        printf(" %s\n", env[i]);
    }

    /* printing a specific environment variable */
    printf("\nEnvironment variable:\n");
    printf("  env[\"HOME\"] = \"%s\"\n", getenv("HOME"));
    printf("  env[\"zzz\"] = \"%s\"\n", getenv("zzz"));

    return EXIT_SUCCESS;
}
```
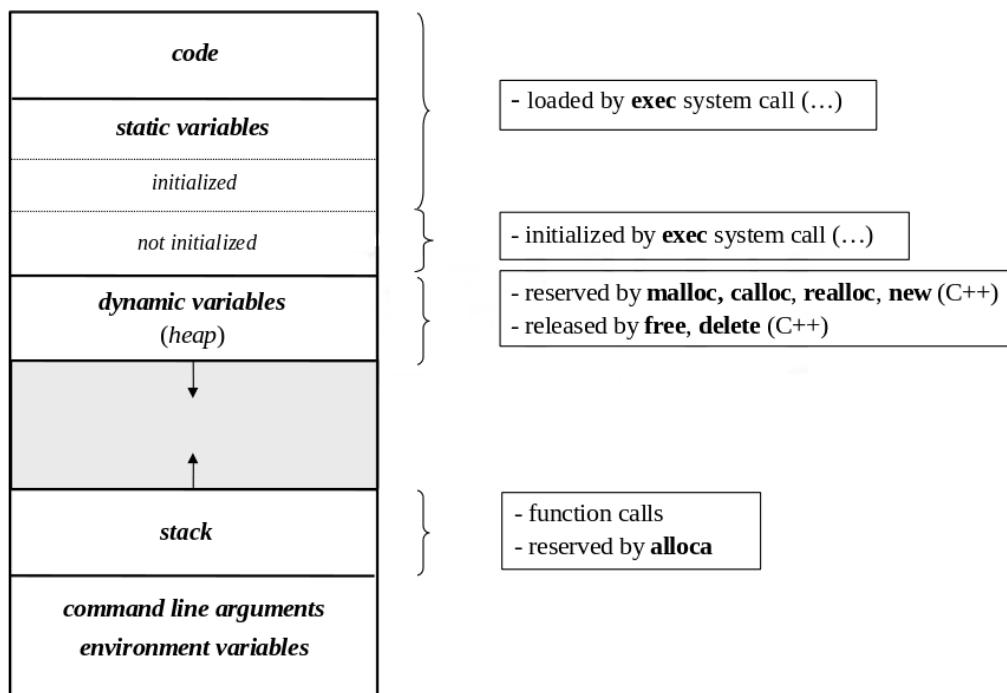
- argv is an array of strings
- argv[0] is the program reference
- env is an array of strings, each representing a variable, in the form name-value pair
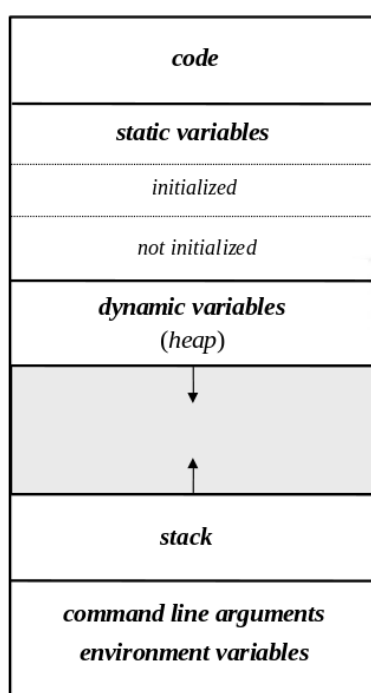- getenv returns the value of a variable name

Processes in Unix
Address space of a Unix process

code	- loaded by **exec** system call (…)
static variables	
initialized	- initialized by **exec** system call (…)
not initialized	
dynamic variables (*heap*)	- reserved by **malloc, calloc, realloc, new** (C++) - released by **free, delete** (C++)
↓ ↑	
stack	- function calls - reserved by **alloca**
command line arguments environment variables	

Processes in Unix
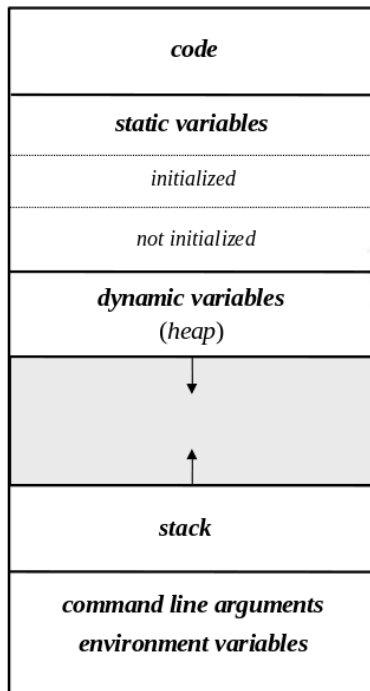## Address space of a Unix process (2)


```
int n1 = 1;
static int n2 = 2;
int n3;
static int n4;
int n5;
static int n6 = 6;

int main(int argc, char *argv[], char *env[])
{
    extern char** environ;
    static int n7;
    static int n8 = 8;
    int *p9 = (int*)malloc(sizeof(int));
    int *p10 = new int;
    int *p11 = (int*)alloca(sizeof(int));
    int n12;
    int n13 = 13;
    int n14;
    printf("\ngetenv(n0): %p\n", getenv("n0"));
    printf("\nargv: %p\nenviron: %p\nenv: %p\nmain: %p\n\n",
            argv, environ, env, main);
    printf("\n&argc: %p\n&argv: %p\n&env: %p\n",
            &argc, &argv, &env);
    printf("&n1: %p\n&n2: %p\n&n3: %p\n&n4: %p\n&n5: %p\n"
            "&n6: %p\n&n7: %p\n&n8: %p\nn9: %p\nn10: %p\n"
            "p11: %p\n&n12: %p\n&n13: %p\n&n14: %p\n",
            &n1, &n2, &n3, &n4, &n5, &n6, &n7, &n8,
            p9, p10, p11, &n12, &n13, &n14);
```

Processes in Unix
Address space of a Unix process (3)

code	
static variables	
initialized	
not initialized	
dynamic variables (heap)	
↓	
↑	
stack	
command line arguments	
environment variables	

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

int n01 = 1;

int main(int argc, char *argv[], char *env[])
{
    int pid = fork();
    if (pid != 0)
    {
        fprintf(stderr, "%5d: n01 = %-5d (%p)\n",
                pid, n01, &n01);
        wait(NULL);
        fprintf(stderr, "%5d: n01 = %-5d (%p)\n",
                pid, n01, &n01);
    }
    else
    {
        fprintf(stderr, "%5d: n01 = %-5d (%p)\n",
                pid, n01, &n01);
        n01 = 1111;
        fprintf(stderr, "%5d: n01 = %-5d (%p)\n",
                pid, n01, &n01);
    }
    return 0;
}
```