Sistemas de Operação

Processes in Unix/Linux

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

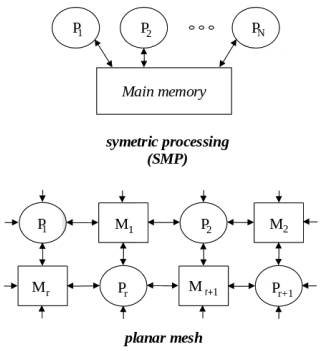
Outline

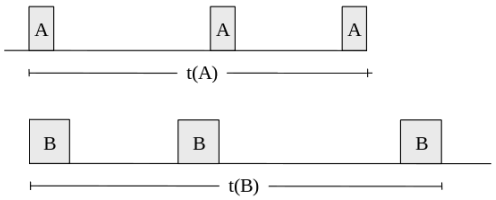
Process
Program vs. process

- 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 transferred 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

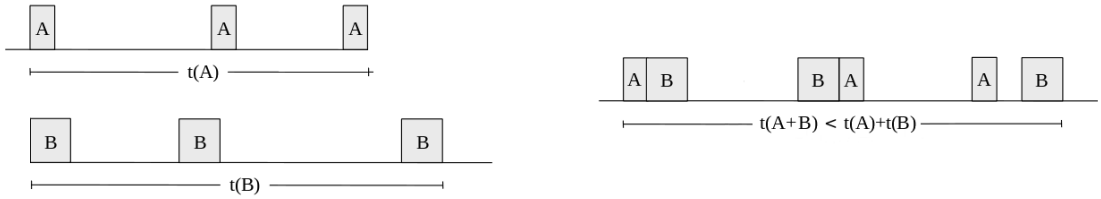




- Programs A and B are executing parallelly in at least two-processors computational system

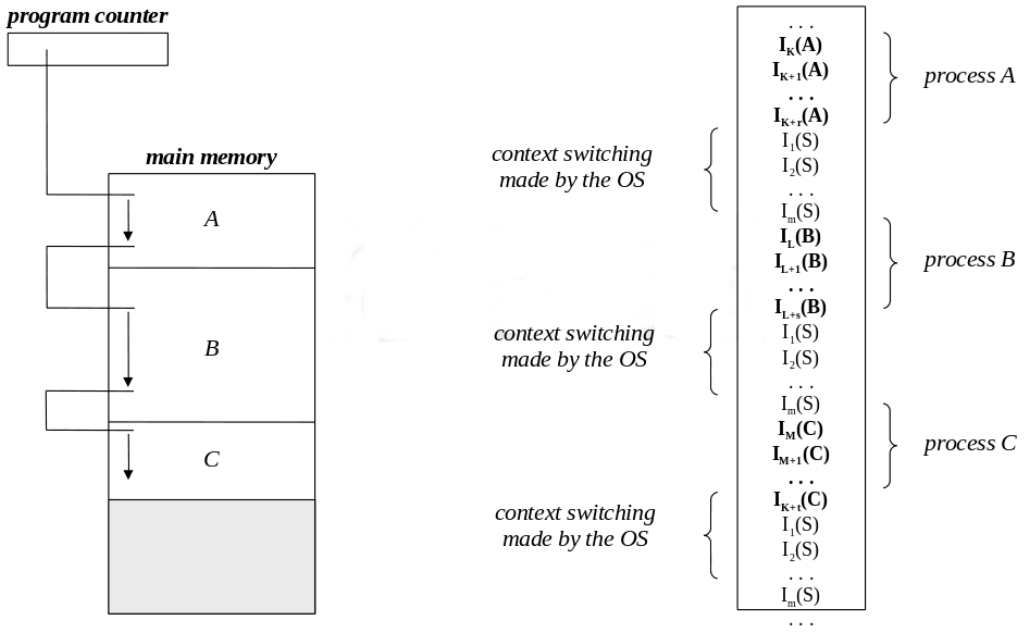
Multiprocessing vs. Multiprogramming
Multiprogramming

- 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

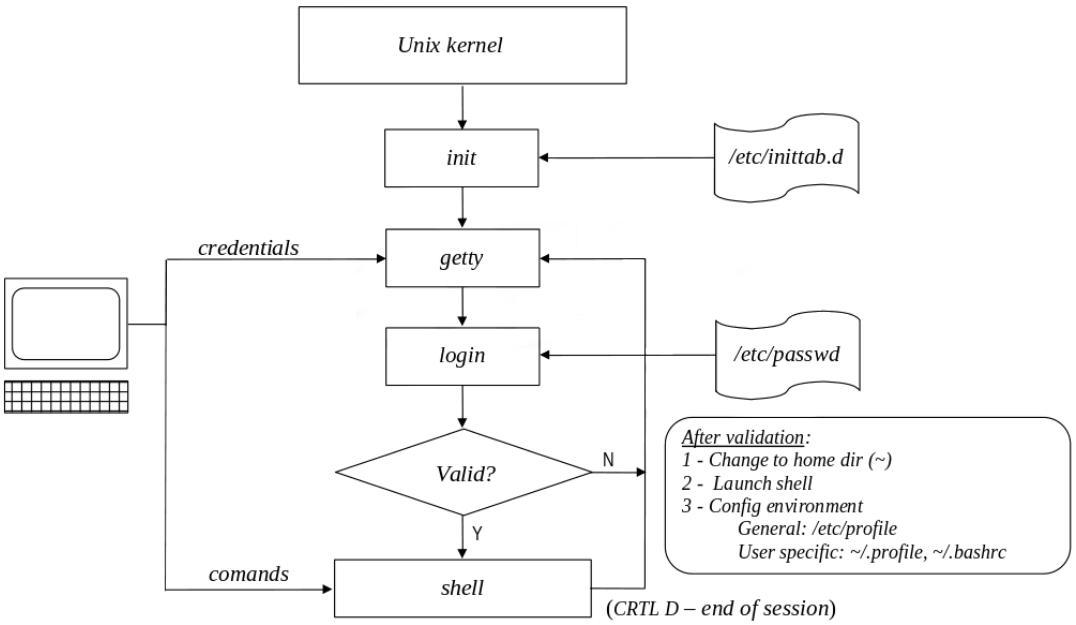


- Programs A and B are executing concurrently in a single processor computational system

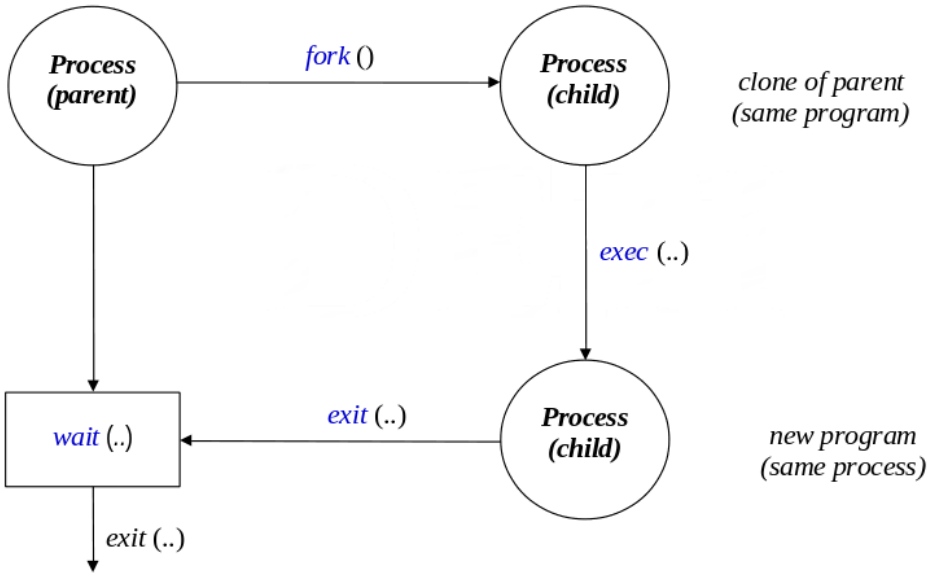
Process
Execution in a multiprogrammed environment



Processes in Unix
Traditional login



Processes in Unix
Creation by cloning



Processes in Unix
Process creation: *syscall* `fork`

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

int main(void)
{
    printf("Hello, World!\n");
    fork();
    printf("Hello, World! Again\n");
    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 value of the program counter

Processes in Unix
Process creation: *syscall* `fork`

```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 value of the program counter
- Some process variables are different (PID, PPID, ...)
- What can we do with this?

Processes in Unix
Process creation: the return of the `fork` system call

```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: the return of the `fork` system call

```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
Program execution: the `exec` family of system calls; the `wait` system call

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

int main(void)
{
  printf("Before the fork\n");

  int ret = fork();

  if (ret == 0)
  {
    execlp("ls", "ls", "-l", NULL);
    return EXIT_FAILURE;
  }
  else
  {
    printf("I'm the parent\n");
    printf("And I will finish, "
        "without waiting for the child to finish\n");
  }

  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 family of system calls
  - 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 system calls don't fail*
  - *in case of failure, they return* `-1`

Processes in Unix
Program execution: the `exec` family of system calls; the `wait` system call

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

int main(void)
{
  printf("Before the fork\n");

  int ret = fork();

  if (ret == 0)
  {
    execlp("ls", "ls", "-l", NULL);
    return EXIT_FAILURE;
  }
  else
  {
    printf("I'm the parent\n");
    printf("And I will finish, "
        "after waiting for the child to finish\n");
    wait(NULL);
  }

  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 family of system calls
  - 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 system calls don't fail*
  - *in case of failure, they return* `-1`

ACP (UA/DETI) SO-2025/2026 September, 2025 15 / 26

Processes in Unix
`fork` illustration program

- What are the possible outputs of the execution of this program?

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```


Processes in Unix
`fork` illustration program

parent process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

- At the beginning, there is only the parent process

Processes in Unix
`fork` illustration program

parent process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

- So, the first `printf` only occurs in the parent

Processes in Unix
`fork` illustration program

parent process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

- When the `fork` starts, still only the parent process exists

Processes in Unix
`fork` illustration program

parent process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

child process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

- But when it ends, the child exists, so there is a return value on both parent and child

Processes in Unix
`fork` illustration program

parent process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

child process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

- The value of `pid` is 0 in child and different from 0 in parent

Processes in Unix
`fork` illustration program

parent process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

child process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

- So, parent and child follow different paths

Processes in Unix
`fork` illustration program

parent process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

child process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

- Parent and child are concurrent and the delay may influence multiprogramming

Processes in Unix
`fork` illustration program

parent process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

child process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

- So, no way to know who prints first

Processes in Unix
`fork` illustration program

parent process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

child process

```
int main(int argc, char *argv[])
{
    uint32_t t = 1000;

    printf("1"); fflush(stdout);
    int pid = fork();
    switch (pid)
    {
        case 0:
            bwDelay(t);
            printf("2"); fflush(stdout);
            bwDelay(t);
            printf("3"); fflush(stdout);
            break;
        default:
            bwDelay(t);
            printf("4"); fflush(stdout);
            bwDelay(t);
            printf("5"); fflush(stdout);
            wait(NULL);
            printf("\n");
    }
    return EXIT_SUCCESS;
}
```

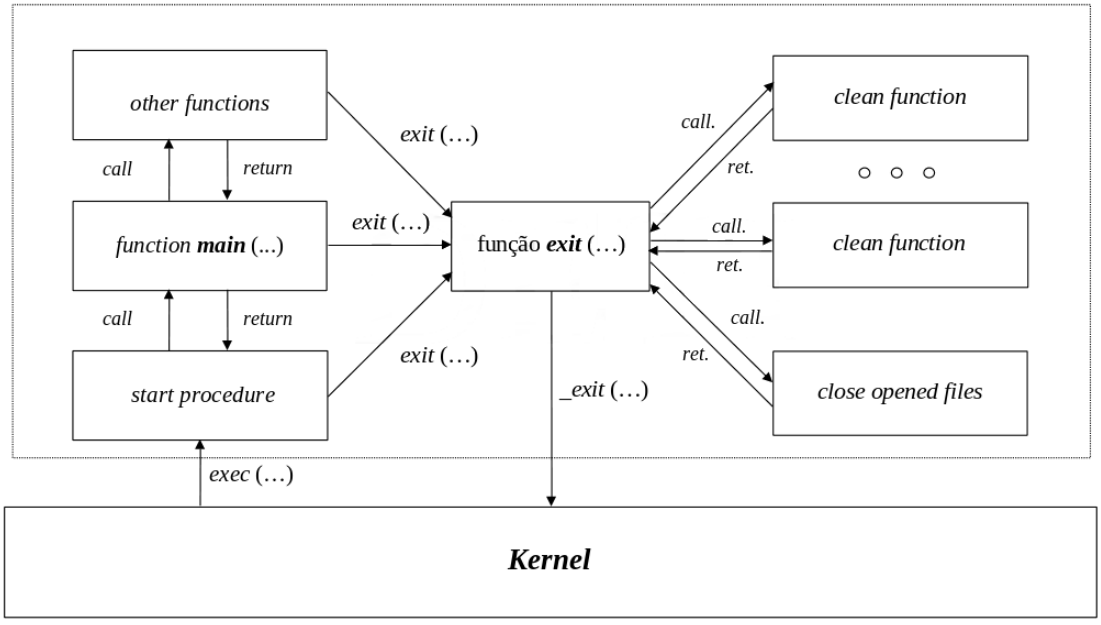
- Because code is sequential: the 5 is always after the 4; the 3 is always after the 2

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"));
    unsetenv("HOME");
    printf("  env[\"HOME\"] = \"%s\"\n", getenv("HOME"));

    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
Execution of a C/C++ program

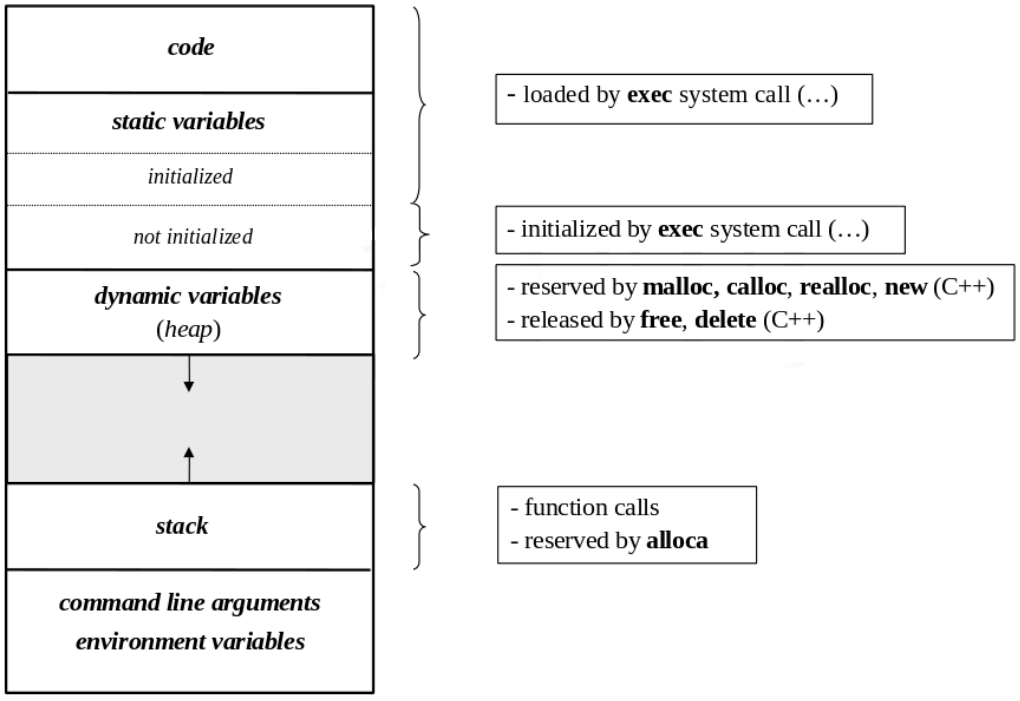


Processes in Unix
Executing a C/C++ program: `atexit`

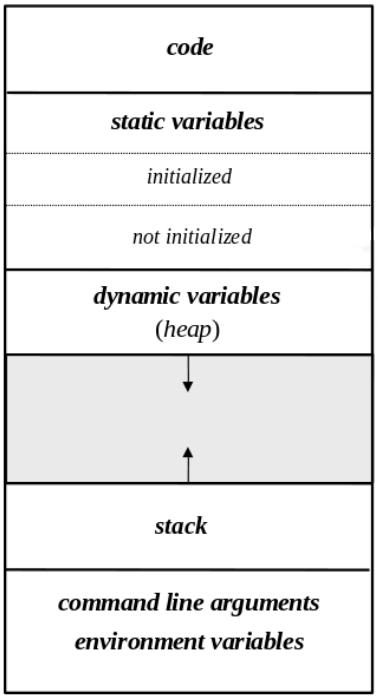
```c
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <assert.h>

/* 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
Address space of a Unix process

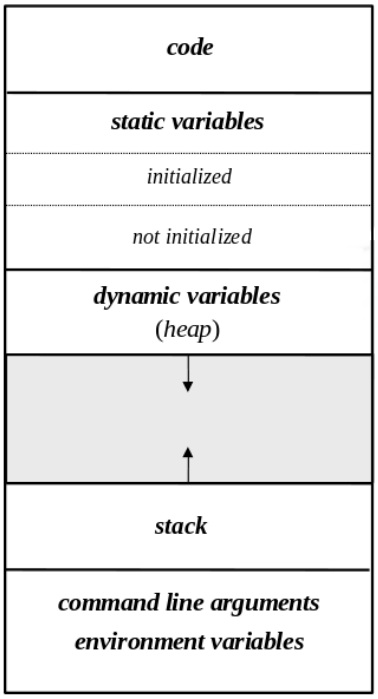


Processes in Unix
Address space of a Unix process (2)




```c
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)




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

int n = 1;

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