



Sistemas de Operação / Fundamentos de Sistemas Operativos

Processes and threads

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

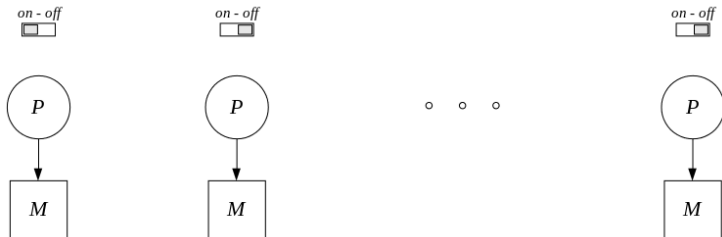
Contents

- ➊ Processes
- ➋ Threads and multithreading
- ➌ Process in Unix
- ➍ Threads in Linux
- ➎ Bibliography

Process

Process model

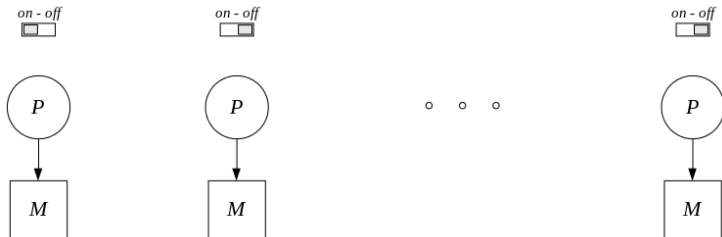
- In **multiprogramming** the activity of the processor, because it is switching back and forth from process to process, is hard to perceive
- Thus, it is better to assume the existence of a number of virtual processors, one per existing process
 - Turning off one virtual processor and on another, corresponds to a process switching
 - number of active virtual processors \leq number of real processors



Process

Process model

- The switching between processes, and thus the switching between virtual processors, can occur for different reasons, possibly not controlled by the running program
- Thus, to be viable, this process model requires that
 - the execution of any process is not affected by the **instant in time** or the **location in the code** where the switching takes place
 - no restrictions are imposed on the total or partial execution times of any process



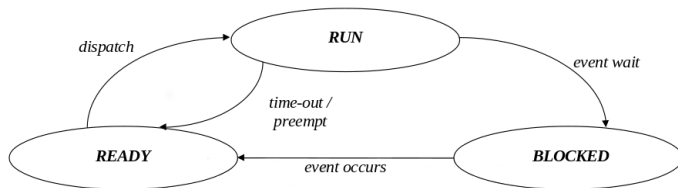
Process

(Short-term) process states

- A process can be **not running** for different reasons
 - so, one should identify the possible process **states**
- The most important are:
 - **run** – the process is in possession of a processor, and thus running
 - **blocked** – the process is waiting for the occurrence of an external event (access to a resource, end of an input/output operation, etc.)
 - **ready** – the process is ready to run, but waiting for the availability of a processor to start/resume its execution
- Transitions between states usually result from external intervention, but, in some cases, can be triggered by the process itself
- The part of the operating system that handles these transitions is called the (**processor**) **scheduler**, and is an integral part of its kernel
 - Different policies exist to control the firing of these transitions
 - They will be covered a few sessions later

Process

Short-term state diagram



- **event wait** – the running process is prevented to proceed, awaiting the occurrence of an external event
- **dispatch** – one of the processes ready to run is selected and is given the processor
- **event occurs** – an external event occurred and the process waiting for it must now wait for the processor
- **preempt** – a higher priority process get ready to run, so the running process is removed from the processor
- **time-out** – the time quantum assigned to the running process get to the end, so the process is removed from the processor

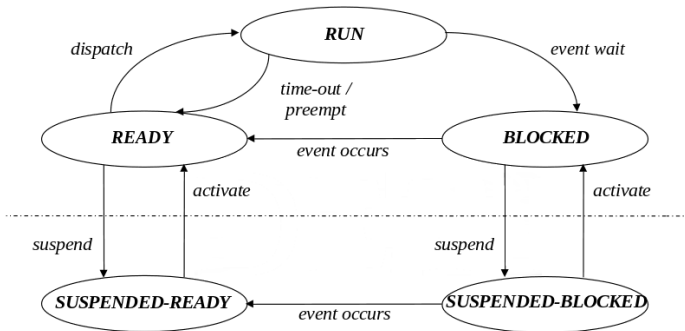
Process

Medium-term states

- The main memory is finite, which limits the number of coexisting processes
- A way to overcome this limitation is to use an area in secondary memory to extend the main memory
 - This is called *swap area* (can be a disk partition or a file)
 - A non running process, or part of it, can be *swapped out*, in order to free main memory for other processes
 - That process will be later on *swapped in*, after main memory becomes available
- Two new states should be added to the process state diagram to incorporate these situations:
 - *suspended-ready* – the process is ready but swapped out
 - *suspended-blocked* – the process is blocked and swapped out

Process

State diagram, including short- and medium-term states



- Two new transitions appear:
 - **suspend** – the process is *swapped out*
 - **activate** – the process is *swapped in*

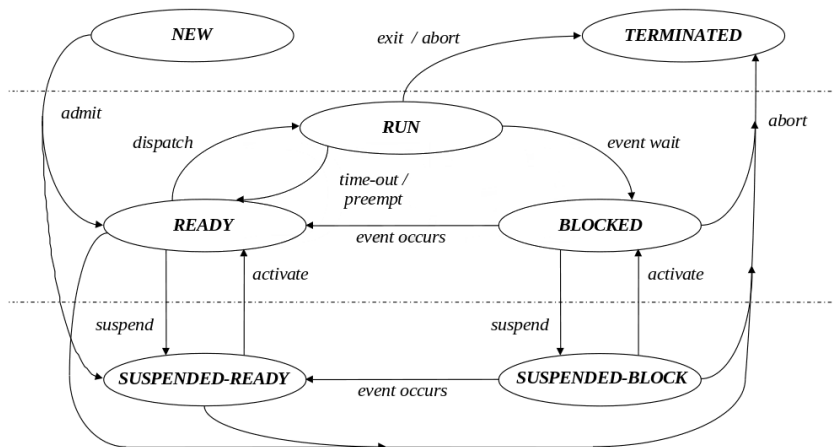
Process

Long-term states and transitions

- The previous state diagram assumes processes are timeless
 - Apart from some system processes this is not true
 - Processes are created, exist for some time, and eventually terminate
- Two new states are required to represent creation and termination
 - **new** – the process has been created but not yet admitted to the pool of executable processes (the process data structure is been initialized)
 - **terminated** – the process has been released from the pool of executable processes, but some actions are still required before the process is discarded
- three new transitions exist
 - **admit** – the process is admitted (by the OS) to the pool of executable processes
 - **exit** – the running process indicates the OS it has completed
 - **abort** – the process is forced to terminate (because of a fatal error or because an authorized process aborts its execution)

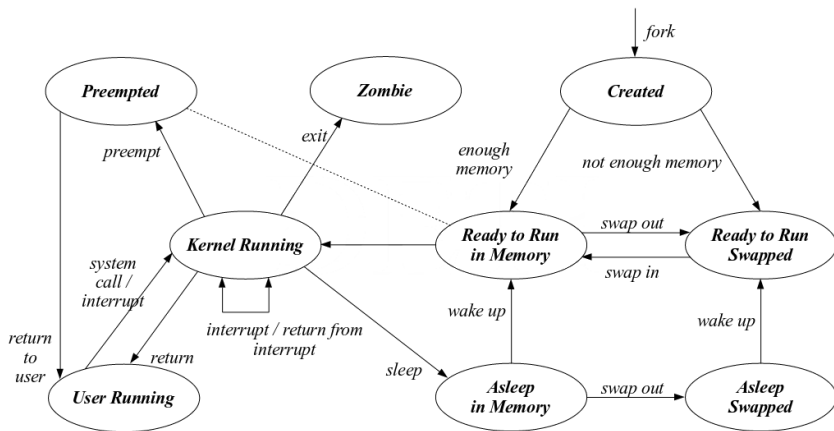
Process

Global state diagram



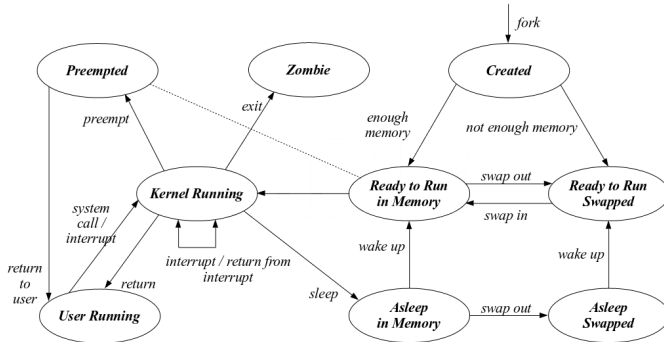
Process

Typical Unix state diagram



Process

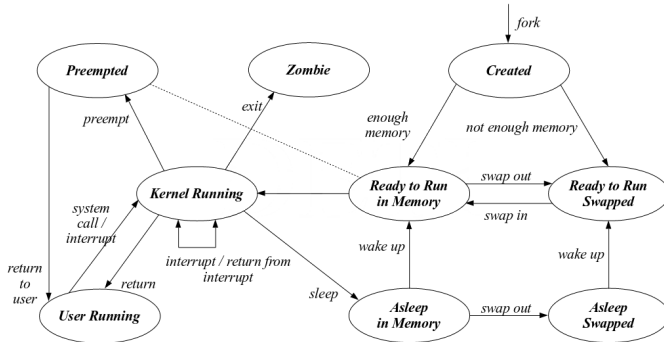
Typical Unix state diagram (2)



- There are two run states, **kernel running** and **user running**, associated to the processor running mode, supervisor and user, respectively
- The ready state is also splitted in two states, **ready to run in memory** and **preempted**, but they are equivalent, as indicated by the dashed line

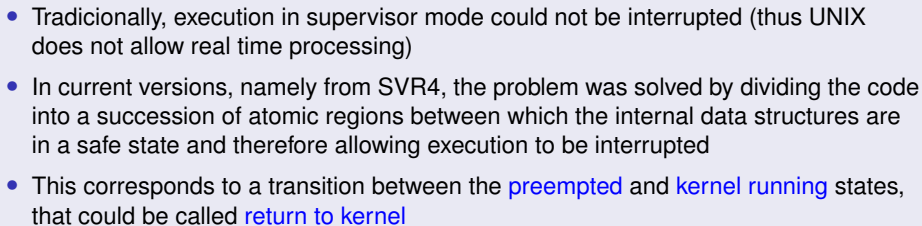
Process

Typical Unix state diagram (3)



- When a user process leaves supervisor mode, it can be preempted (because a higher priority process is ready to run)
- In practice, processes in **ready to run in memory** and **preempted** shared the same queue, thus are treated as equal
- The **time-out** transition is covered by the preempt one

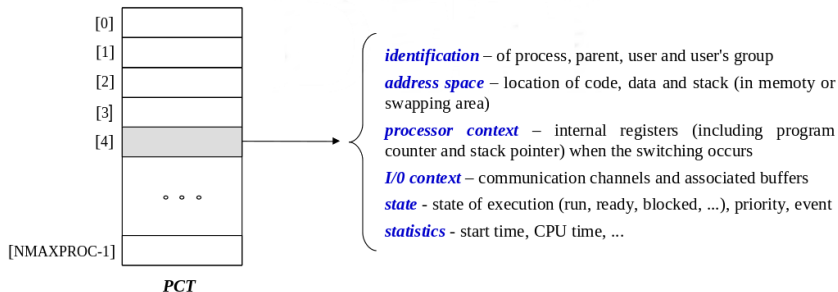
Typical Unix state diagram (4)



Process

Process control table

- To implement the process model, the operating systems needs a data structure to be used to store the information about each process – **process control block**
- The **process control table (PCT)** can be seen as an array of process control blocks, stores information about all processes



Process

Process switching

- Current processors have two functioning modes:
 - **supervisor mode** – all instruction set can be executed
 - is a privileged mode
 - **user mode** – only part of the instruction set can be executed
 - input/output instructions are excluded as well as those that modify control registers
 - it is the normal mode of operation
- Switching from user mode to supervisor mode is only possible through an **exception** (for security reasons)
- An exception can be caused by:
 - I/O interrupt
 - illegal instruction (division by zero, bus error)
 - trap instruction (software interruption)

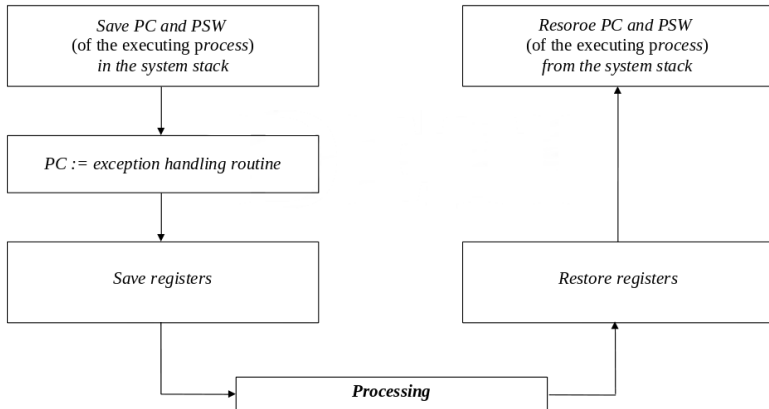
Process

Process switching (2)

- The operating system should function in supervisor mode
 - in order to have access to all the functionalities of the processor
- Thus kernel functions (including system calls) must be fired by
 - hardware (interrupt)
 - trap (software interruption)
- There is a uniform operation environment: **exception handling**
- Process switching can be seen the same way, with a small difference

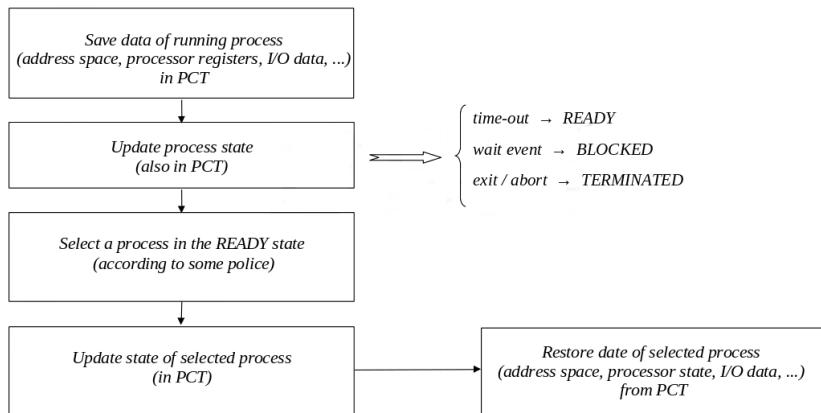
Process

Processing a (normal) exception



Process

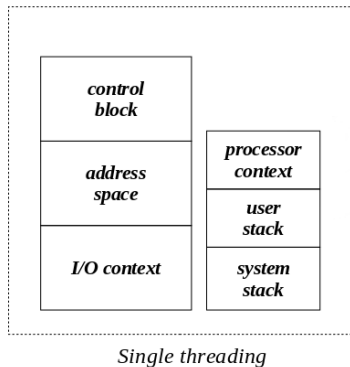
Processing a process switching



Threads

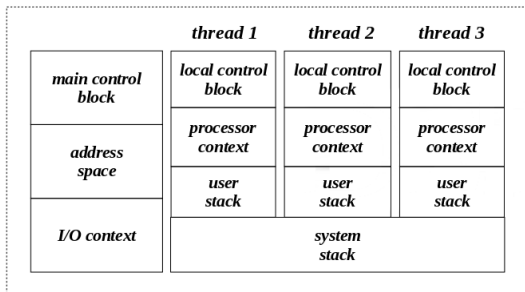
Single threading

- In traditional operating system, a process includes:
 - an address space (code and data of the associated program)
 - a set of communication channels with I/O devices
 - a single thread of control, which incorporates the processor registers (including the program counter) and a stack
- However, these components can be managed separately
- In this model, **thread** appears as an execution component within a process



Threads

Multithreading

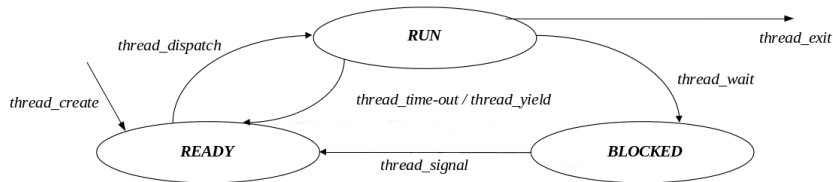


Multithreading

- Several independent threads can coexist in the same process, thus sharing the same address space and the same I/O context
 - This is referred to as *multithreading*
- Threads can be seen as *light weight processes*

Threads

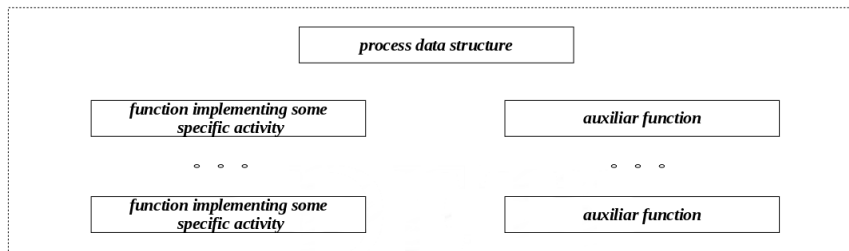
State diagram of a thread



- Only states concerning the management of the processor are considered (short-term states)
- states **suspended-ready** and **suspended-blocked** are not present:
 - they are related to the process, not to the threads
- states **new** and **terminated** are not present:
 - the management of the multiprogramming environment is basically related to restrict the number of threads that can exist within a process

Threads

Structure of a multithreaded program



- Each thread is typically associated to the execution of a function that implements some specific activity
- Communication between threads can be done through the process data structure, which is global from the threads point of view
- The main program, also represented by a function that implements a specific activity, is the first thread to be created and, in general, the last to be destroyed

Threads

Implementations of multithreading

- **user level threads** – threads are implemented by a library, at user level, which provides creation and management of threads without kernel intervention
 - versatile and portable
 - when a thread calls a blocking system call, the whole process blocks
 - because the kernel only sees the process
- **kernel level threads** – threads are implemented directly at kernel level
 - less versatile and less portable
 - when a thread calls a blocking system call, another thread can be scheduled to execution

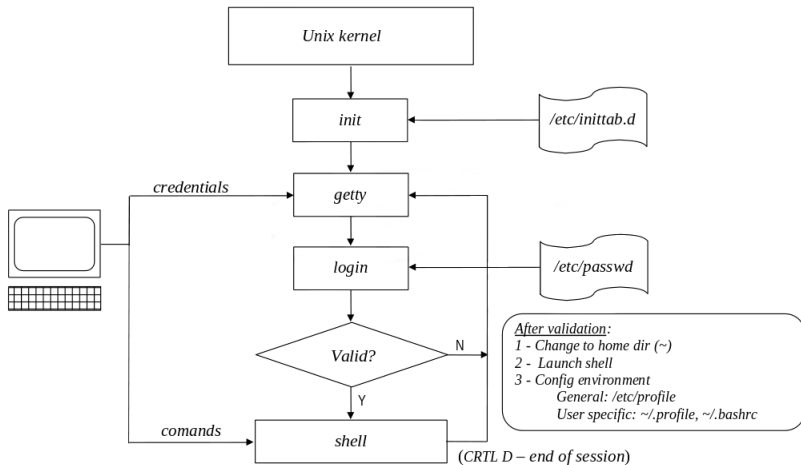
Threads

Advantages of multithreading

- **easier implementation of applications** – in many applications, decomposing the solution into a number of parallel activities makes the programming model simpler
 - since the address space and the I/O context is shared among all threads, multithreading favors this decomposition.
- **better management of computer resources** – creating, destroying and switching threads is easier then doing the same with processes
- **better performance** – when an application involves substantial I/O, multithreading allows activities to overlap, thus speeding up its execution
- **multiprocessing** – real parallelism is possible if multiples CPUs exist

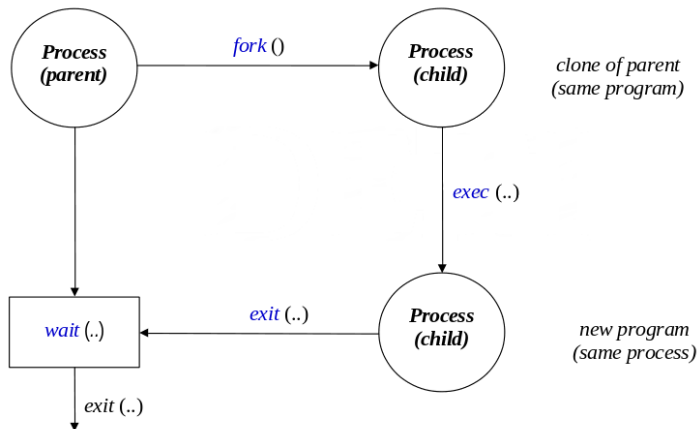
Processes in Unix

Traditional login



Processes in Unix

Creation by cloning



Processes in Unix

Process creation: `fork1`

```
#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.",
        getpid(), getppid());
    printf("  Which is mine?\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 program counter
- Some process variables are different (PID, PPID, ...)
- What can we do with this?

Processes in Unix

Process creation: `fork2`

```
#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, ret = %d\n",
        getpid(), getppid(), 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
- 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`

```
#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;
}
```

- By itself, 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`

```
#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");

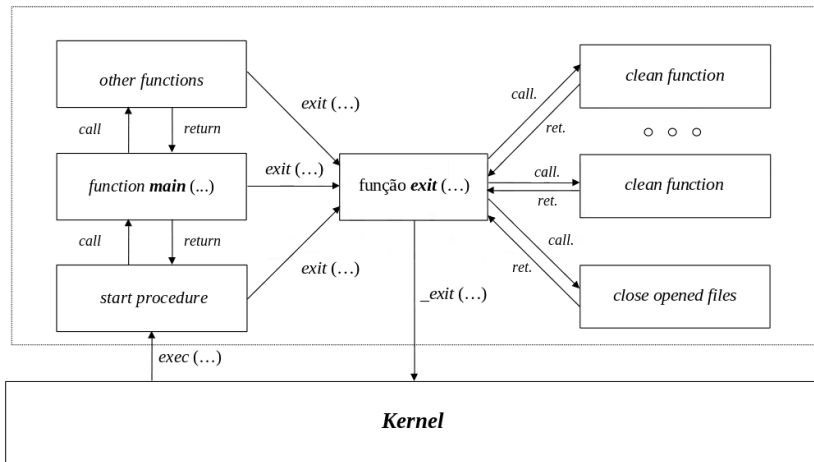
    /* clone phase */
    int pid;
    if ((pid = fork()) < 0)
    {
        perror("Fail cloning process");
        exit(EXIT_FAILURE);
    }
}
```

```
/* 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`

```
#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

```
#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]);
    }

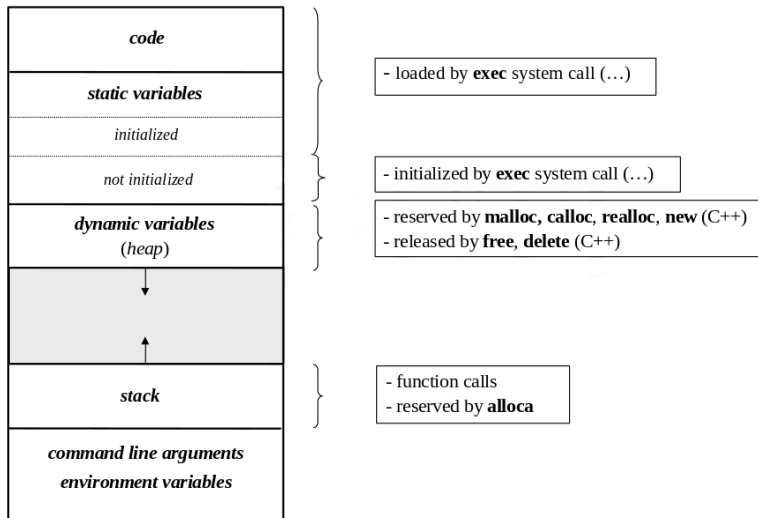
    /* printing a specific environment variable */
    printf("\nEnvironment variable:\n");
    printf(" env[\"HOME\"] = \"%s\"\n", getenv("HOME"));
    printf(" env[\"zzz\"] = \"%s\"\n", getenv("zzz"));

    return EXIT_SUCCESS;
}
```

- **argv** is an array of strings
- **argv[0]** is the program reference
- **env** is an array of strings, each one representing a variable, in the form **name-value** pair
- **getenv** returns the value of a variable name

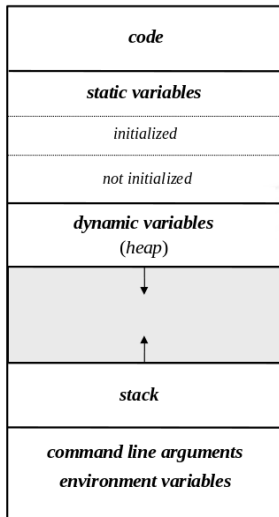
Processes in Unix

Address space of a Unix process



Processes in Unix

Address space of a Unix process (2)

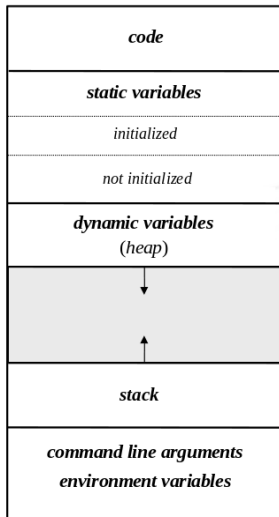


```
int n1 = 1;
static int n2 = 2;
int n3;
static int n4;
int n5;
static int n6 = 6;
```

```
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\nmain: %p\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\n&n9: %p\n&n10: %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)



```
#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;
}
```

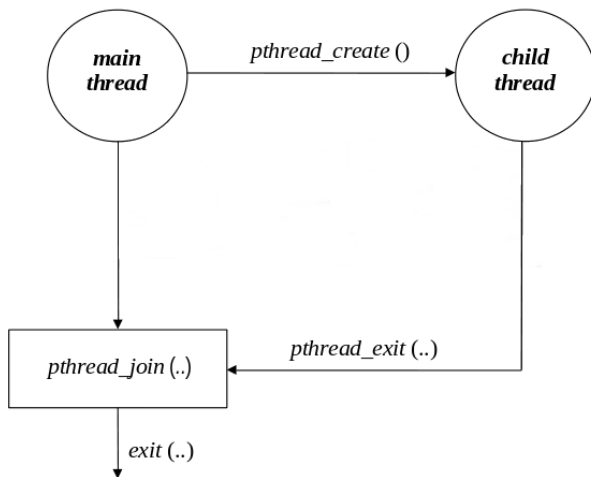
Threads in linux

The `clone` system call

- In Linux there are two system calls to create a child process:
 - **`fork`** – creates a new process that is a full copy of the current one
 - the address space and I/O context are duplicated
 - the child starts execution in the point of the fork
 - **`clone`** – creates a new process that can share elements with its parent
 - address space, table of file descriptors, and table of signal handlers, for example, are shareable.
 - the child starts execution in a given function
- Thus, from the kernel point of view, processes and threads are treated similarly
- Threads of the same process forms a thread group and have the same thread group identifier (TGID)
 - this is the value returned by system call `getpid()`
- Within a group, threads can be distinguished by their unique thread identifier (TID)
 - this value is returned by system call `gettid()`

Threads in linux

Thread creation and termination – `pthread` library



Threads in linux

Thread creation and termination – example

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

/* return status */
int status;

/* child thread */
void *threadChild (void *par)
{
    printf ("I'm the child thread!\n");

    sleep(1);
    status = EXIT_SUCCESS;
    pthread_exit (&status);
}
```

```
/* main thread */
int main (int argc, char *argv[])
{
    /* launching the child thread */
    pthread_t thr;
    if (pthread_create (&thr, NULL,
                       threadChild, NULL) != 0)
    {
        perror ("Fail launching thread");
        return EXIT_FAILURE;
    }

    /* waits for child termination */
    if (pthread_join (thr, NULL) != 0)
    {
        perror ("Fail joining child");
        return EXIT_FAILURE;
    }

    printf ("Child ends; status %d.\n", status);

    return EXIT_SUCCESS;
}
```


Bibliography

- Operating Systems: Internals and Design Principles, W. Stallings, Prentice-Hall International Editions, 7th Ed, 2012
 - Chapter 3: Process Description and Control (sections 3.1 to 3.5 and 3.7)
 - Chapter 4: Threads (sections 4.1, 4.2 and 4.6)
- Operating Systems Concepts, A. Silberschatz, P. Galvin and G. Gagne, John Wiley & Sons, 9th Ed, 2013
 - Chapter 3: Processes (sections 3.1 to 3.3)
 - Chapter 4: Threads (sections 4.1 and 4.4.1)
- Modern Operating Systems, A. Tanenbaum and H. Bos, Pearson Education Limited, 4th Ed, 2015
 - Chapter 2: Processes and Threads (sections 2.1 and 2.2)