



Sistemas de Operação / Fundamentos de Sistemas Operativos

Threads, mutexes and condition variables in Unix/Linux

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

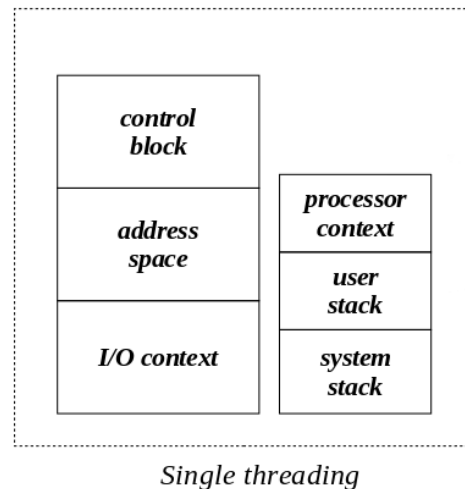
Outline

- ① Threads and multithreading
- ② Threads in Linux
- ③ Monitors
- ④ POSIX support for monitors

Threads

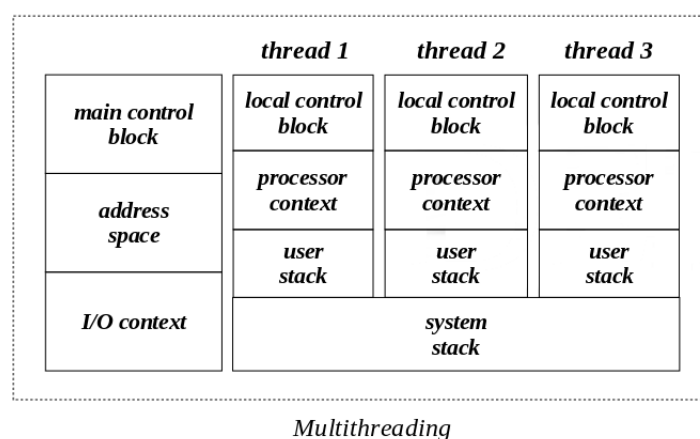
Single threading

- 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

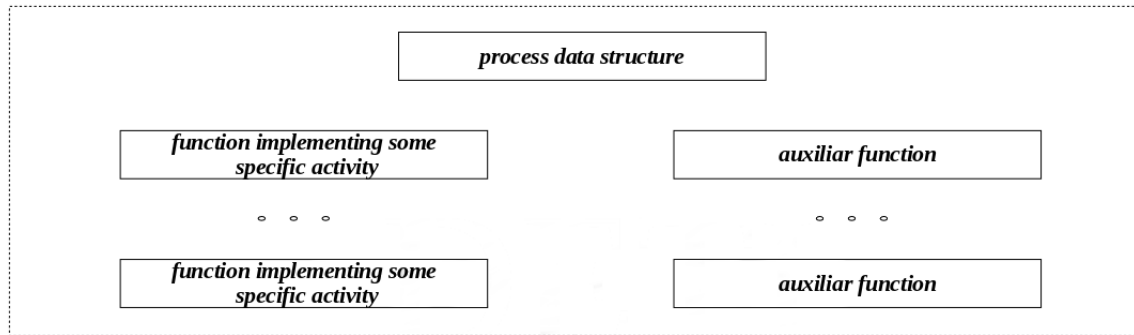
Multithreading



- 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

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
 - It includes **static** and **dynamic** variables (heap memory)
- 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 schedule 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 than 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

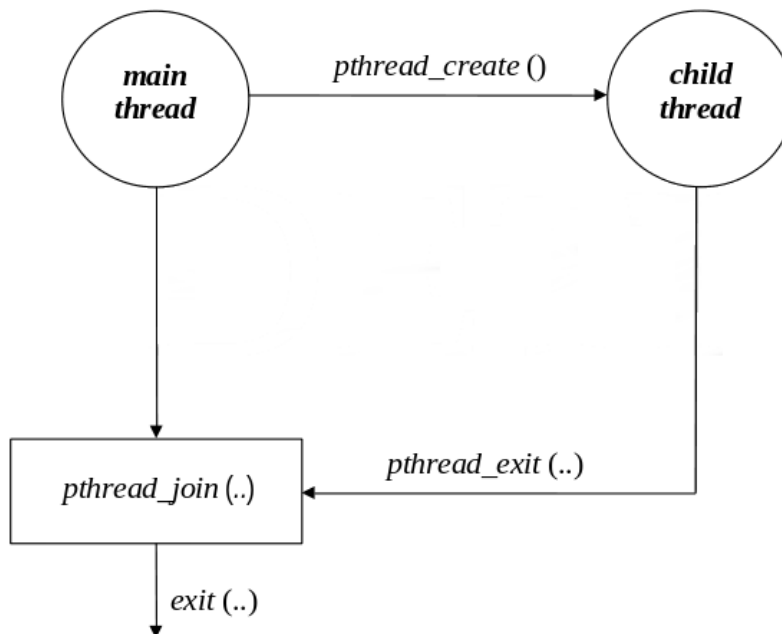
Threads in linux

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 forking
 - **`clone`** – creates a new process that can share elements with its parent
 - address space, table of file descriptors, and table of signal handlers are shareable.
 - the child starts execution in a specified 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;
}
```

Monitors

Introduction

- A problem with semaphores is that they are used both to implement **mutual exclusion** and to **synchronize** processes
 - Being low level primitives, they are applied in a **bottom-up** perspective
 - if required conditions are not satisfied, processes are blocked before they enter their critical sections
 - this approach is prone to errors, mainly in complex situations, as synchronization points can be scattered throughout the program
 - A higher level approach should followed a **top-down** perspective
 - processes must first enter their critical sections and then block if continuation conditions are not satisfied
 - A solution is to introduce a (concurrent) construction at the programming level that deals with mutual exclusion and synchronization separately
-
- A **monitor** is a synchronization mechanism, independently proposed by Hoare and Brinch Hansen, supported by a (concurrent) programming language
 - The **pthread** library provides primitives that allows to implement monitors (of the Lampson-Redell type)

Monitors

Definition

```
monitor example
{
    /* internal shared data structure */
    DATA data;

    cond c; /* condition variable */

    /* access methods */
    method_1 (...)
    {
        ...
    }

    method_2 (...)
    {
        ...
    }

    ...

    /* initialization code */
    ...
}
```

- An application is seen as a set of threads that compete to access the **shared data** structure
- This shared data can only be accessed through the access methods
- Every method is executed in **mutual exclusion**
- If a thread calls an access method while another thread is inside another access method, its execution is blocked until the other leaves
- Synchronization between threads is possible through **condition variables**
- Two operation on them are possible:
 - **wait** – the thread is blocked and put outside the monitor
 - **signal** – if there are threads blocked, one is waked up. *Which one?*

Monitors

Bounded-buffer problem – solving using monitors

```
shared FIFO fifo;      /* fixed-size FIFO memory */
shared mutex access;   /* mutex to control mutual exclusion */
shared cond nslots;    /* condition variable to control availability of slots */
shared cond nitems;    /* condition variable to control availability of items */

/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        lock(access);
        if/while (fifo.isFull())
        {
            wait(nslots, access);
        }
        fifo.insert(data);
        signal(nitems);
        unlock(access);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        lock(access);
        if/while (fifo.isEmpty())
        {
            wait(nitems, access);
        }
        fifo.retrieve(&data);
        signal(nslots);
        unlock(access);
        consume_data(data);
        do_something_else();
    }
}
```

- The **mutex** is the resource used to control mutual exclusion
- **Critical sections** are explicitly framed by the **lock** and **unlock** of a mutex

Unix IPC primitives

POSIX support for monitors

- Standard POSIX, IEEE 1003.1c, defines a programming interface (API) for the creation and synchronization of threads
 - In unix, this interface is implemented by the **pthread** library
- It allows for the implementation of monitors in C/C++
 - Using mutexes and condition variables
 - Note that they are of the **Lampson / Redell** type
- Some of the available functions:
 - **pthread_create** – creates a new thread; similar to `fork`
 - **pthread_exit** – equivalent to `exit`
 - **pthread_join** – equivalent a `waitpid`
 - **pthread_self** – equivalent a `getpid()`
 - **pthread_mutex_*** – manipulation of mutexes
 - **pthread_cond_*** – manipulation of condition variables
 - **pthread_once** – initialization