



Sistemas de Operação

Threads, mutexes and condition variables in Unix/Linux

Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

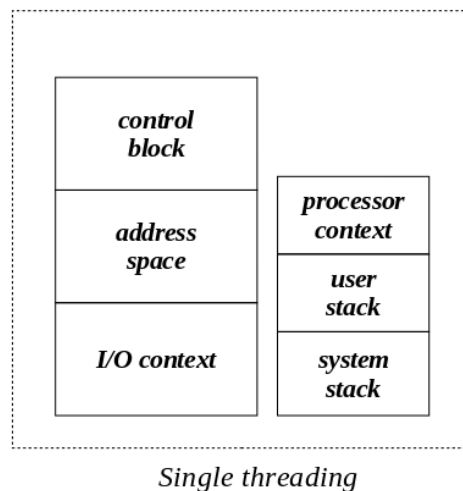
Outline

- ① Threads and multithreading
- ② Threads in Linux
- ③ Monitors
- ④ POSIX support for implementing monitors
- ⑤ Bounded-buffer problem – solving using monitors
- ⑥ Sleeping barber problem, using monitors

Threads

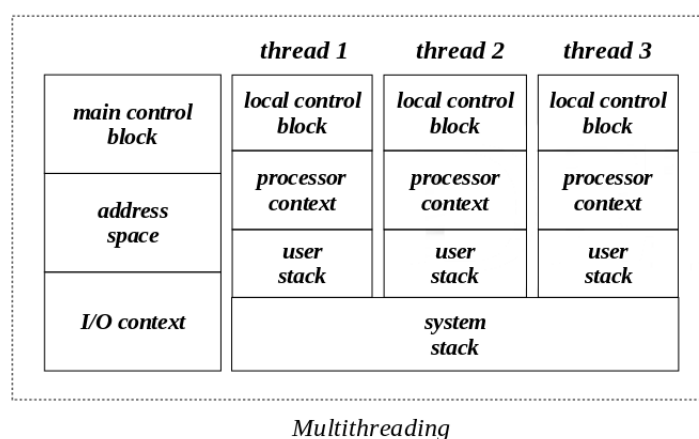
Single threading

- 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

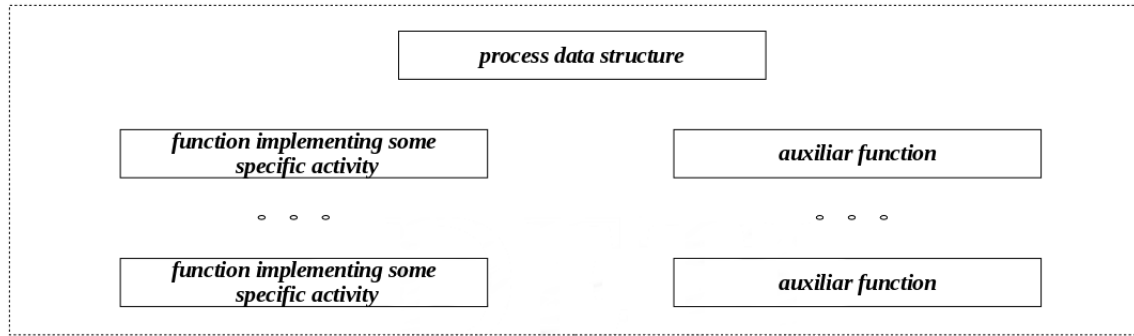
Multithreading



- 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** (data segment) and **dynamic** (heap segment) variables
- 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 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

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()`

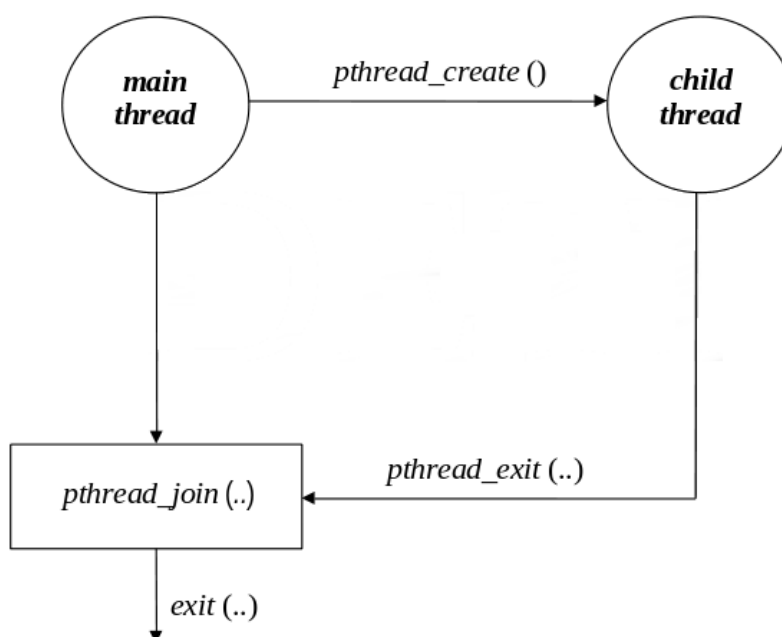
Thread in Linux

POSIX library

- Standard POSIX, IEEE 1003.1c, defines a programming interface (API) for the creation and synchronization of threads
 - In Linux, this interface is implemented by the `pthread` library
- Some of the available functions to manage threads:
 - `pthread_create` – create a new thread (corresponding to the `fork` in processes)
 - `pthread_exit` – terminate calling thread (corresponding to the `exit` in processes)
 - `pthread_join` – joint with a terminated thread (corresponding to the `waitpid` in processes)
 - `pthread_kill` – send a signal to a thread (corresponding to the `kill` in processes)
 - `pthread_cancel` – send a cancellation request to a thread
 - `pthread_self` – obtain ID of the calling thread
 - `pthread_detach` – detach a thread

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 *arg)
{
    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;
}
```

Thread synchronization

Introducing monitors

- A problem with semaphores is that they are used both to implement **mutual exclusion** and to **synchronize**
 - Also, 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 wait 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 such 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)

Thread synchronization

Monitor 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?*

Thread synchronization

POSIX support for monitors

- The `pthread` library allows for the implementation of monitors in C/C++
 - **mutexes** (mutual exclusion elements) are used to implement **mutual exclusion**
 - **condition variables** are used to implement **synchronization**
- Support for mutual exclusion:
 - `pthread_mutex_t` – the mutex data type
 - `pthread_mutex_init` – initializes a mutex object
 - `pthread_mutex_lock` – locks the given mutex
 - `pthread_mutex_unlock` – unlocks the given mutex
- Support for synchronization:
 - `pthread_cond_t` – the condition variable data type
 - `pthread_cond_init` – initializes a condition variable object
 - `pthread_cond_wait` – atomically unlocks an associated mutex and waits for the given condition variable to be signaled.
 - `pthread_cond_signal` – restarts one of the threads that are waiting on the given condition variable
 - `pthread_cond_broadcast` – restarts all of the threads that are waiting on the given condition variable

Monitor

Bounded-buffer problem – making the implementation safe

```
void fifoInsert(FIFO *f, uint32_t id, uint32_t v1, uint32_t v2)
{
    /* wait until fifo is not full */
    while (fifolsFull(f))
    {
        bwDelay(10); // wait for a while
    }

    /* make insertion */
    f->data[f->in].id = id;
    f->data[f->in].v1 = v1;
    f->data[f->in].v2 = v2;
    f->in = (f->in + 1) % f->size;
    f->full = (f->in == f->out);
}

void fifoRetrieve(FIFO *f, uint32_t *idp, uint32_t *v1p, uint32_t *v2p)
{
    /* wait until fifo is not empty */
    while (fifolsEmpty(f))
    {
        bwDelay(10); // wait for a while
    }

    /* make retrieval */
    *idp = f->data[f->out].id;
    *v1p = f->data[f->out].v1;
    *v2p = f->data[f->out].v2;
    f->out = (f->out + 1) % f->size;
    f->full = false;
}

void fifoDestroy(FIFO *f)
```

Monitor

Bounded-buffer problem – making the implementation safe

```
void fifoInsert(FIFO *f, uint32_t id, uint32_t v1, uint32_t v2)
{
    lock
    /* wait until fifo is not full */
    while (fifolsFull(f))
    {
        wait until not full (unlocking internally)
    }

    /* make insertion */
    f->data[f->in].id = id;
    f->data[f->in].v1 = v1;
    f->data[f->in].v2 = v2;
    f->in = (f->in + 1) % f->size;
    f->full = (f->in == f->out);
}
    signal not empty
    unlock

void fifoRetrieve(FIFO *f, uint32_t *idp, uint32_t *v1p, uint32_t *v2p)
{
    lock
    /* wait until fifo is not empty */
    while (fifolsEmpty(f))
    {
        wait until not empty (unlocking internally)
    }

    /* make retrieval */
    *idp = f->data[f->out].id;
    *v1p = f->data[f->out].v1;
    *v2p = f->data[f->out].v2;
    f->out = (f->out + 1) % f->size;
    f->full = false;
}
    signal not full
    unlock
```


Monitor

Bounded-buffer problem – safe implementation using monitors

```
void fifoInsert(FIFO *f, uint32_t id, uint32_t v1, uint32_t v2)
{
    /* lock access on entry */
    mutex.lock(&f->access);

    /* wait until fifo is not full */
    while (fifolsFull(f))
    {
        cond_wait(&f->notFull, &f->access);
    }

    /* make insertion */
    f->data[f->in].id = id;
    f->data[f->in].v1 = v1;
    f->data[f->in].v2 = v2;
    f->in = (f->in + 1) % f->size;
    f->full = (f->in == f->out);

    /* signal fifo is not empty */
    cond_broadcast(&f->notEmpty);

    /* unlock access before quitting */
    mutex.unlock(&f->access);
}

void fifoRetrieve(FIFO *f, uint32_t *idp, uint32_t *v1p, uint32_t *v2p)
{
    /* lock access on entry */
    mutex.lock(&f->access);

    /* wait until fifo is not full */
    while (fifolsEmpty(f))
    {
        cond_wait(&f->notEmpty, &f->access);
    }

    /* make retrieval */
    *idp = f->data[f->out].id;
    *v1p = f->data[f->out].v1;
    *v2p = f->data[f->out].v2;
    f->out = (f->out + 1) % f->size;
    f->full = false;

    /* signal fifo is not empty */
    cond_broadcast(&f->notFull);

    /* unlock access before quitting */
    mutex.unlock(&f->access);
}
```

Illustration example

Sleeping barber example – problem statement



4 waiting chairs



1 barber chair

Barber Shop

- In this problem, attributed to Dijkstra, a number of entities (customers) interact with another entity (barber) to request a service (haircut)
- Problem statement:
 - If there are no customers, the barber falls asleep in the barber chair
 - Upon entering, if the barber is sleeping, a customer must wake him up,
 - Otherwise, if the barber is working, the customer leaves if all chairs are occupied or sits if one is available
 - Upon finishing a haircut, the barber inspects the waiting room to see if there are any waiting customers, falling asleep if there are none