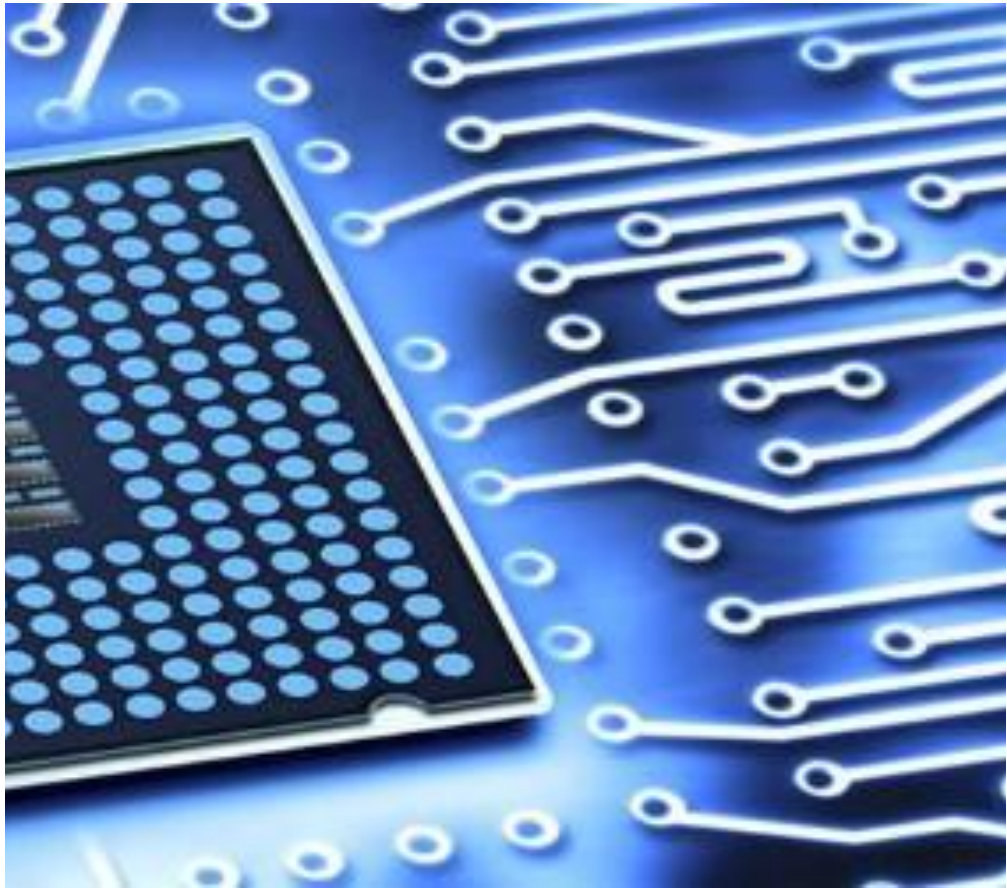
PERCORSO DI FORMAZIONE “PROGETTAZIONE FIRMWARE”



Linux IPC for embedded applications
- Part 1

Davide Baroffio

davide.baroffio@polimi.it

I edizione

Titolo Piano Formativo: PROGETTO FORMATIVO DIGITAL PLATFORMS 2025
Codice identificativo: 424945
Numero identificativo azione: 3655432
Titolo azione: PROGETTAZIONE FIRMWARE

Cefriel
POLITECNICO DI MILANO

Percorso

INTRODUCTION TO COMPUTING PLATFORMS AND OPERATING SYSTEMS	**SHELL, BASIC COMMANDS, BASH, BASH SCRIPTING**	**C/C++ USER-SPACE CODE DEVELOPMENT IN LINUX**
SECURE PROGRAMMING IN C/C++ FOR EMBEDDED	**LINUX IPC FOR EMBEDDED APPLICATIONS (USER-SPACE)**	**SYSTEM ADMINISTRATION FOR EMBEDDED SYSTEMS**
CONTAINERS	**INTRODUCTION TO EMBEDDED LINUX**	**EMBEDDED LINUX DISTRIBUTION**
KERNEL-SPACE PROGRAMMING	**REAL-TIME LINUX**	**OVERVIEW ON ADVANCED TOPICS**

Cefriel
POLITECNICO DI MILANO

Contents

Notation

In these slides we use the following notation for file contents:

```
int main() {
   printf(“This is a file\n”);
}
```

And this for commands written in the command line

```
echo “this is a shell command”
# this is a comment
```

Introduction to parallel programming

Cefriel
POLITECNICO DI MILANO

Parallel programming

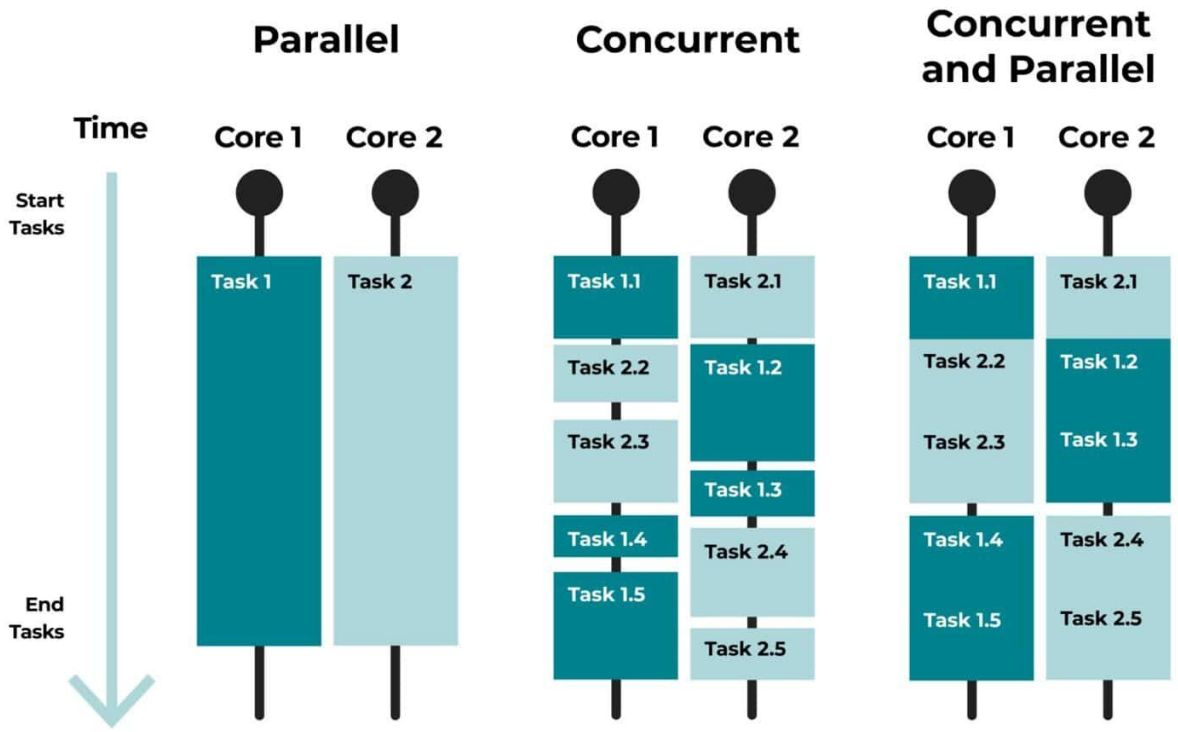
Parallel programming is a method that allows to perform multiple computations simultaneously.

For doing so, we typically rely on multiple computing elements (e.g. multi-core CPUs, GPU accelerators).

Note: Programs may also *appear* to run simultaneously, but in reality they may interleave on the same core. In this case, we talk about **concurrent** programming.

Concurrent programming

In modern operating systems running on multi-core machines, the job of the OS is to schedule the access of the tasks to the cores.

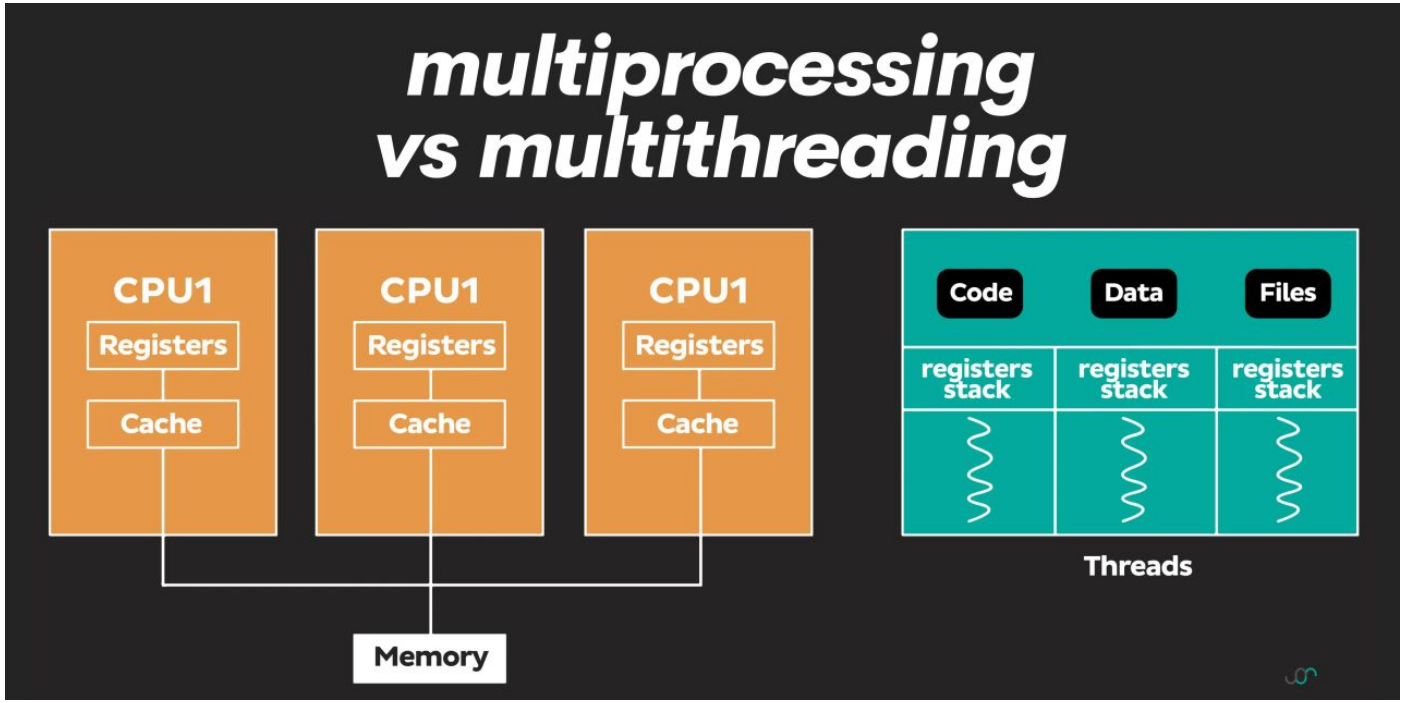


Cefriel
POLITECNICO DI MILANO

Process vs Thread

A **process** (or task) is a program in execution on the machine.

A **thread** is a basic unit of execution that lives within a process.



Objective: Multiprocessing vs Multithreading

Multiprocessing

Running multiple parallel processes independently, making them communicate and sync via the operating system.

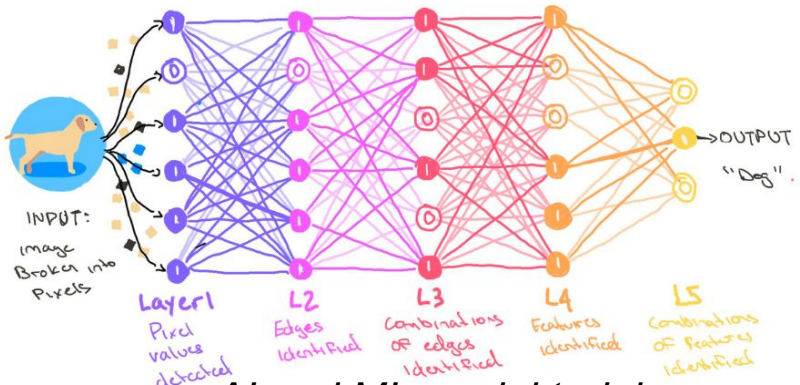
Shared memory must be explicitly set up.

Multithreading

Methods for doing parallel work inside a single process, safely sharing memory.

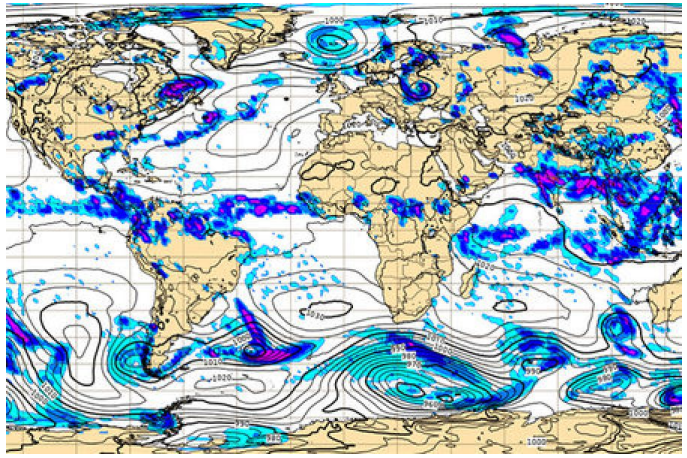
Memory is shared by default.

Examples of parallelized applications



AI and ML model training

Video rendering and encoding



Weather forecasting

Big data analysis

Cefriel
POLITECNICO DI MILANO

Code with me & Hands-on

It is recommended to follow along, we will use a running example based on the code provided in this repository:

https://github.com/davidebaroffio-polimi/cefriel-IPC-examples

```
# go in a directory of your choice
cd <path/to/your/dir>
# clone the repo and go there
git clone https://github.com/davidebaroffio-polimi/cefriel-IPC-examples
cd cefriel-IPC-examples
```

Multithreading with pthread

Cefriel
POLITECNICO DI MILANO

Pthread

Pthread (POSIX Thread) is the standard API for thread management and creation in C and C++.

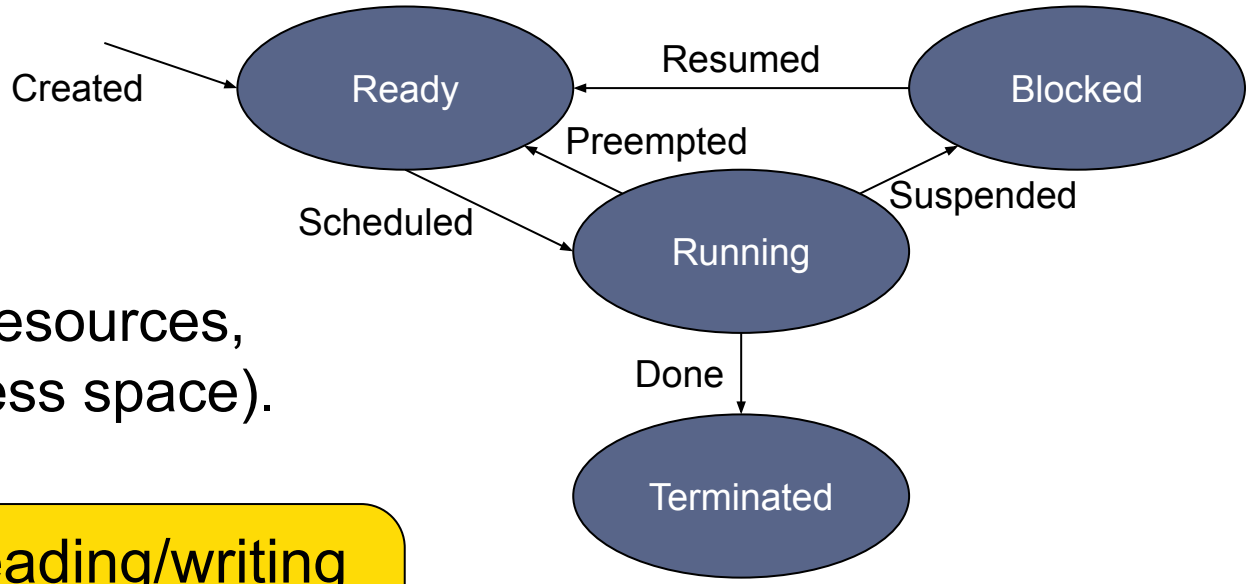
It is a standard defined by POSIX (issued by IEEE).

Pthreads can leverage multi-processor or multi-core systems for parallel execution, and different threads may run simultaneously on different processors.

Pthread threads

Each thread is independent as it maintains its own:

- Stack pointer
- Registers
- Scheduling properties
- Signals



But all threads share the same process resources, including: file descriptors, pointers (address space).

Note: shared memory entails that reading/writing the same memory location is possible, we need to synchronize threads to avoid race conditions!

Pthread overview

Pthread offers three main features:

- Thread management
- Mutexes
- Condition Variables (condvars)

To compile a threaded program with GCC we use:

```
gcc -pthread …
```

Creating a multithreaded program

We first need to include the pthread library:

```
#include <pthread.h>
```

We create a function that runs the code of the thread:

```
void *func(void *args) {...}
```

Then we create a thread with:

```
pthread_t t;
int pthread_create (&t, NULL, func, NULL);
```

>> Live example

Terminating threads

A thread can terminate in four main ways:

- The threaded function returns
- The thread calls `pthread_exit()`
- The thread is canceled via `pthread_cancel()`
- The process terminates (e.g. `exit` or `exec`)

Note: if the main returns before the threads terminate, all the threads are killed. We can use pthread_exit in the main as well, so that the process will be kept alive.

>> Live example

Joining threads

We may need to wait for threads to terminate in the parent thread.

A typical example is a master-slave pattern in which the master waits for the slave to compute the result.
We can set a thread to “sleep” until another thread terminates by joining them:

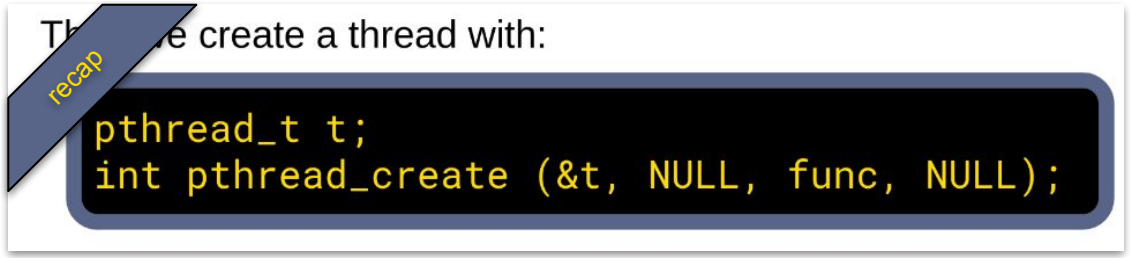
Stores a pointer to the exit status of the thread.

```
pthread_join (&t, <thread_ret>);
```

>> Live example

Argument passing

Previously, we put some of the arguments of pthread_create conveniently to NULL, but what do they do?



The last argument is a pointer to the arguments of the threaded routine.

```
pthread_t t;
int arg = 0xC1A0;
pthread_create (&t, NULL, func, (void*) &arg);
```

>> Live example

Cefriel
POLITECNICO DI MILANO

Hands-on

Recap:

```
# go in a directory of your choice
cd <path/to/your/dir>
# clone the repo and go there
git clone https://github.com/davidebaroffio-polimi/cefriel-IPC-examples
cd cefriel-IPC-examples
```

In the root of the repo, there is a file named file_analyzer.c, that computes the number of lines in a file.

Task 1

Your task is to parallelize the application using pthread, creating a thread for each chunk and accumulating the results in the main.

>> Live example

Cefriel
POLITECNICO DI MILANO

Attributes

What about the other NULL argument of pthread_create?

It is used for attributes, which provide specific get/set APIs we can use for setting the attributes of the thread.

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_set<attr_name>(&attr, …);
int pthread_create (&t, &attr, func, NULL);
```

Interesting thread attributes

Here is a list of pthread thread attributes that may be useful:

- `detachstate`
- `guardsize`
- `inheritsched`
- `schedparam`
- `schedpolicy`
- `scope`
- `stackaddr`
- `stacksize`

We will not look into them for timing reasons.

Pthread useful APIs

- `pthread_self()`
  - Returns the unique thread id (pthread_t) of the calling thread.
- pthread_equal(t1, t2)
  - Returns 0 if t1 and t2 are different, otherwise a non-zero value is returned.
- pthread_yield()
  - Forces the calling thread to relinquish the use of the processor and wait for the thread to be rescheduled.

Mutexes

A mutex (short for *Mutual Exclusion*), is a powerful instrument that can be used to perform thread synchronization and protecting concurrent data access.

A mutex can be used to prevent *race conditions*:

Thread 1	**Thread 2**	**Balance**
Read 10		10
	Read 10	10
Write 10 + 5		15
	Write 10 + 5	15

Should be 20!

Mutexes in pthread

We can create a mutex in pthread statically…

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
```

… or dynamically:

```
pthread_mutex_t *my_mutex;
pthread_mutex_init(my_mutex, NULL)
```

This argument can be used to set mutex attributes, similarly to pthread attributes

Mutual exclusion of resources

With a mutex we can implicitly lock a resource by locking the mutex, other threads trying to access the resource will wait for the mutex to unlock before accessing the variable.

```
pthread_mutex_t mymutex = …;

pthread_mutex_lock(&mymutex);
balance+=deposit; //              ← critical instruction
pthread_mutex_unlock(&mymutex);
```

We can also say that the critical instruction is atomic, since it cannot* be interrupted in the middle.

>> Live example

*Mutex cautions

Caution 1

You are not locking a resource! If other threads lock it because the programmer misused locks, atomicity is not guaranteed.

Thread 1	Thread 2	Thread 3
Lock	Lock	
balance = balance + 5;	balance = 20;	balance = balance + 30;
Unlock	Unlock	

*Mutex cautions

Caution 2

Locking a mutex causes all other threads trying to access the same resource to block. Consider using the non-blocking version:

```
if(!pthread_mutex_trylock(&mymutex))
{
  // do stuff
}
```

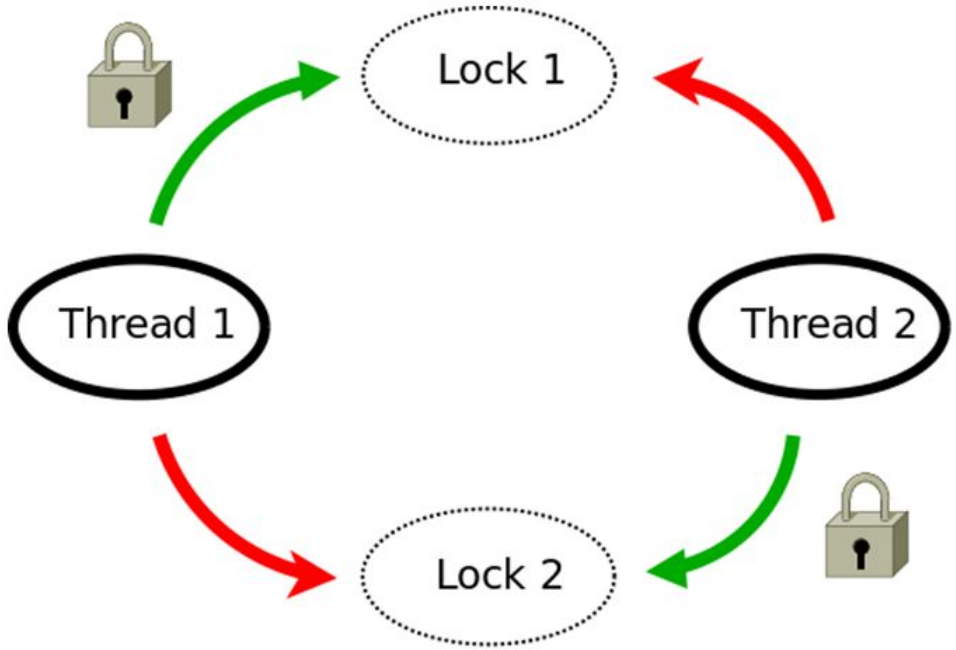
>> Live example

*Mutex cautions

Caution 3

Beware of deadlocks!

Circular lock dependencies may make your application stuck in an infinite loop.



Hands-on

Task 2

Following task 1, update the `res` global variable using mutexes, instead of making the main accumulate the results.

Bonus

Update `res` at each iteration (every fgetc) with a best effort approach using try locks.

>> Live example

Condition variables

Condition variables are another way to perform thread synchronization. They can be seen as a way to send notifications between threads.

If we wait for a condition to be satisfied before running a critical section, instead of polling, we can signal via condition variables.

Condvars in pthread

Similarly to mutexes, we can create a condition variable in pthread statically…

```
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;
```

… or dynamically:

```
pthread_cond_t *condvar;
pthread_cond_init(condvar, NULL)
```

This argument can be used to set cond attributes, similarly to mutexes and thread attributes.

Condvars waiting and signaling

>> Live example

We can block a thread until a specified condvar is signaled:

```
pthread_cond_wait(&condvar, &some_mutex);
```

A condvar is signaled in two ways, depending on the number of threads waiting on the condvar:

```
pthread_cond_signal(&condvar);      // 1 thread waiting
pthread_cond_broadcast(&condvar);   // >1 threads waiting
```

Note: condition variables are always used in conjunction with mutexes. Routines for wait and signal must be preceded and followed by mutex locks and unlocks, respectively.

Hands-on

Task 3

Following task 2, we want to use a condition variable in the main thread that waits until all the threads have terminated the computation.

>> Live example

Inter-Process Communication

Cefriel
POLITECNICO DI MILANO

Multiprocessing

In contrast with multithreading, multiprocessing is a technique that spawns different programs, eventually coordinating them via Inter-Process Communication mechanisms.

Bash

We can multiprocess in bash using the & character:

```
./process1 & ./process2 & …
```

C

We can multiprocess in C using `fork()` with (optionally) `execve()`:

```
pid_t pid = fork();
if (pid == 0) execve(...); // child
else {
   // body
}
```

Forking

Forking a program in C means creating an exact copy of it in the memory.
We distinguish between parent and child of a forked process:

- The **parent** is the original forked process, that generated the new process
- The **child** is the newborn process, identical to the original process

At runtime:

- The forking procedure returns the pid of the child if the current process is the parent.
- The forking procedure returns 0 if the current process is the child.

Other pid utilities:

```
#include <unistd.h>
…
pid_t my_pid = getpid();
pid_t parent_pid = getppid();
```

Inter-Process Communication

By Inter-Process Communication we refer to the set of functionalities provided by the operating system that enable coordination and data exchange among different processes.

We will see:

- Signals
- Pipes & FIFOs
- Message queues
- Shared memory
- Semaphores

Signals

The use of signals is the oldest and simplest IPC method.

They allow to send asynchronous events to other processes and are also typically used by the kernel to indicate an error (e.g. segmentation fault) or I/O status.

Note: Non-root users are usually not permitted to send signals to processes of other users.

Unix signals

Signals cannot transport data, but have an identifier that defines the “type” of signal.

1:	SIGHUP	14:	SIGALRM	*:	SIGTTOU	*:	SIGXFSZ
2:	SIGINT	15:	SIGTERM	*:	SIGBUS	*:	SIGEMT
3:	SIGQUIT	*:	SIGUSR1	*:	SIGPOLL	*:	SIGSTKFLT
4:	SIGILL	*:	SIGUSR2	*:	SIGPROF	*:	SIGIO
6:	SIGABRT	*:	SIGCHLD	*:	SIGSYS	*:	SIGPWR
8:	SIGFPE	*:	SIGCONT	5:	SIGTRAP	*:	SIGLOST
9:	**SIGKILL**	*:	**SIGSTOP**	*:	SIGURG	*:	SIGWINCH
11:	SIGSEGV	*:	SIGTSTP	*:	SIGVTALRM	*:	SIGUNUSED
13:	SIGPIPE	*:	SIGTTIN	*:	SIGXCPU		

Developers can design handlers for all signals, except SIGKILL and SIGSTOP.

Sending signals from bash

In bash, you can send signals via the kill command:

```
kill -<signal> <PID>
```

Where <signal> is either the identifier (number) of the signal, or the name of the signal (e.g. SIGKILL).

We can also use killall to send signals to all processes given the program name:

```
killall -<signal> <name>
```

Managing signals in C

To send a signal to a process from C, we can use the kill() function of signal.h:

```
kill(pid, signum);
```

We can also define custom handlers as follows:

```
void handler(int signum) {
   // handle signal
}

…
   signal(SIGUSR1, handler);
```

>> Live example

Hands-on

Task 4

Given the multithreaded program of Task 3, can we use the user-defined SIGUSR1 signal rather than condition variables to notify the main that the tasks have finished?

>> Live example

Pipes

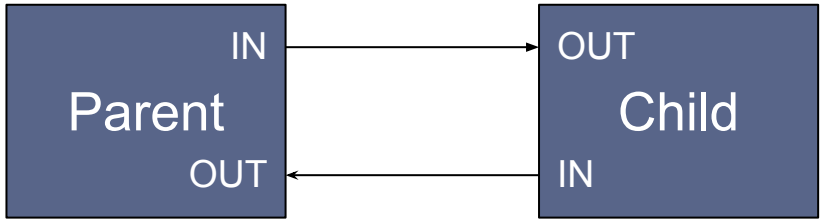
We have already seen pipes in bash: `strings program.elf | grep password`

that maps the stdout of the first program to the stdin of the second.

C pipes work similarly, creating two in-memory file descriptors, one for reading and one for writing, used for communication between parent-child (forked) processes.

Note: Pipes can **only** be used by processes that share a related ancestry (e.g. parent/child)

Pipes are unidirectional, so we need to create a pipe in each direction.



Cefriel
POLITECNICO DI MILANO

Creating pipes in C

A pipe is created using the pipe() procedure:

```
int fd[2];
pipe(fd);
```

After calling pipe:

- fd[0] contains the *reading* end of the pipe
- fd[1] contains the *writing* end of the pipe

Parent and child have to close the unused file descriptors.

Bidirectional pipes

The standard procedure for bidirectional pipes is:

- Create two pipes fd1 and fd2
- Fork the process
- One process closes the read fd of fd1 and the write fd of fd2
- The other process closes the read fd of fd2 and the write fd of fd1
- Both processes read and write on the remaining file descriptors

```
int fd1[2]; int fd2[2];
pipe(fd1); pipe(fd2);

pid_t pid = fork();
if (pid == 0) {
    close(fd1[0]);
    close(fd2[1]);
    // write on fd1 and read on fd2
} else {
    close(fd1[1]);
    close(fd2[0]);
    // read on fd1 and write on fd2
}
```

Code with me

There is a file called `message_pipes.c` that sends messages from the child process to its parent.

Let’s take a look at that to see an example of pipes :)

>> Live example

FIFO, a.k.a. named pipes

FIFO (named pipes) are similar to pipes, with the following differences:

- They exist in the filesystem
- Ancestry relation is no longer mandatory
- They are not destroyed when they are closed

We can create a fifo from bash using the mkfifo command

>> Live example

```
$ mkfifo my_fifo
$ ls -l my_fifo
prw-rw-r-- 1 buoffio buoffio 0 apr 18 13:58 my_fifo
```

Note: Although it looks like a regular file, it has the `p` flag, telling us that it is a named pipe.

FIFOs in C

We can create a FIFO in C with mkfifo().

```
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
…
if (mkfifo(fifo_path, 0666) == -1) {
    // check if fifo already exists
    if (errno != EEXIST) {
        perror("mkfifo failed");
        return 1;
    }
}
// do stuff
```

Coincides with permissions: `rw-rw-rw-`

FIFO sender code

The sender opens the fifo as a standard file and writes the message on the queue:

```
int fd = open(FIFO_NAME, O_WRONLY);

char *msg = “hello world\n”;

// Write message to FIFO
write(fd, msg, strlen(msg));
```

By default, `open()` blocks if there is no reading process.
If you want to have a non-blocking opening, use:

```
int fd = open(FIFO_NAME, O_WRONLY | O_NONBLOCK);
```

FIFO receiver code

The receiver opens the fifo as a standard file and reads the message from the queue:

```
#include <fcntl.h>

FILE *fd = open(FIFO_NAME, O_RDONLY);

char buffer[100];

// Read message from FIFO
int bytesRead = read(fd, buffer, sizeof(buffer)-1);
```

Note: We are using open/write/read instead of fopen/fwrite/fread because they are not buffered.

Hands-on

Task 5

Implement `message_pipes.c` using FIFO.

>> Live example

FIFO concurrent access

Granting access to concurrent processes to FIFO queues is done by the OS according to a set of rules:

- Multiple readers can open a FIFO
- Multiple writers can open a FIFO
- Bytes are passed from writers to readers, reading bytes from the queue deletes them
- The kernel guarantees atomicity for writing less than PIPE_BUF bytes at a time

```
#include <linux/limits.h>
```

Rule of thumbs

- Write small messages
- If you write big messages, consider structuring the data (e.g. XML)

Message queues

Message queues provide a higher level message passing interface with respect to FIFO. We can create a message queue as follows:

```
#include <mqueue.h>

mqd_t mq = mq_open("/mymq", O_CREAT | O_RDWR, 0666, NULL);
```

File permissions (only used for O_CREAT)

Queue attributes (nullable, and only used for O_CREAT)

Or just open it if it already exists:

```
mqd_t mq = mq_open("/mymq", O_RDWR);
```

Note: Message queues have a naming convention similar to files and must begin with a “/”.
Message queues can be found at `/dev/mqueue/`

Message queue attributes

Message queues have a set of useful attributes that can be set upon queue creation.

```
struct mq_attr attr;

attr.mq_flags = 0;          // 0 or O_NONBLOCK
attr.mq_maxmsg = 10;        // max #messages
attr.mq_msgsize = 100;      // size of each message
attr.mq_curmsgs = 0;        // current #messages

mq = mq_open(QUEUE_NAME, O_CREAT | O_RDWR, 0644, &attr);
```

Developers may also get them using the appropriate getter:

```
struct mq_attr attr;
mq_getattr(mq, &attr);
```

Message exchange with message queues

With message queue:

- We have no directions and file descriptors
- We can send bytes as messages in the queue and read/write them atomically
- We have message priorities

```
mq_send(mq, "Hello MQ", 8, 0);

char buf[100];
mq_receive(mq, buf, 100, NULL);
```

Note: Message queues are good for nice, structured data, streams are better handled with FIFOs.

Managing message queues

Once a queue is open, we can close it (similarly to what we do with files) using mq_close:

```
mq_close(mq);
```

We can also delete a queue with mq_unlink, which tells the kernel to delete a queue when all processes stop using it:

```
mq_unlink("/mymq");
```

Hands-on

Task 6

Implement `message_pipes.c` using message queues.

>> Live example

Shared memory

Shared memory allows two or more processes to access the same memory region. The implementation in the kernel (from Linux 2.4) works as a memory-mapped file located in `/dev/shm/`

After creation, shared memory can be managed via mmap, munmap, and so on…

Shared memory is created and unlinked similarly to message queues:

```
#include <sys/mman.h>
int shm_fd = shm_open(“/myshm”, <o_flags>, <permissions>);
shm_unlink("/myshm");
```

Mapping shared memory

Once a shared memory object has been created, it can be resized with `ftruncate`. For instance, assuming we want to create a shared memory region for sharing the content of an integer across multiple processes:

```
// we want to share an integer among
ftruncate(shm_fd, sizeof(int));
```

We can then access the shared memory mapping it to an address of our process with `mmap`:

```
int *shared_int = mmap(
    NULL, sizeof(int), PROT_READ | PROT_WRITE,
    MAP_SHARED, shm_fd, 0
);
```

Synchronizing shared memory

To manage shared memory access between processes, we need mutual exclusion.

Linux provides POSIX **semaphores** (a generalization of mutexes) allowing up to N processes to coordinate access.

Semaphores are ideal for managing **resource pools** (e.g., threads, connections, buffers), controlling how many processes can access resources at once.

POSIX Semaphores

Semaphores are another way for processes to synchronize their actions.

They are based on **wait** and **post** actions:

- sem_wait(): "I want a resource. If none are available, I'll wait."
- sem_post(): "I'm done with the resource. Here, someone else can have it."

The semaphore is initialized with a value that represents the size of available resources in the resource pool.

At each wait and post, the semaphore value is decremented or incremented.

Semaphore management

The semaphore API for creation and deletion is similar to the one of message queues and shared memory.

```
#include <semaphore.h>

// create a semaphore for 10 resources
sem_t *my_sem = sem_open(“/my_sem”, O_CREAT, 0666, 10);

// delete the semaphore
sem_unlink(“/my_sem”);
```

Set this to 0 if the semaphore already exists (the other two arguments will be ignored)

Synchronizing w/ semaphores

Each process must access a shared resource via the semaphore, surrounding critical sections with wait/post actions:

```
sem_wait(my_sem);   // lock

// begin critical section
…
// end critical section

sem_post(my_sem);   // unlock
```

Hands-on

Task 7

Implement the file analyzer example (line count) via multiprocessing and IPC such that:

- we fork the original process NUM_CHUNKS times
- res is in a shared memory location
- processes can modify res acquiring the lock via a semaphore